

Basic Preprocessed Library: try to have a STL--

Marc Boyer
IRIT - TésA
2, rue Camichel
31000 Toulouse
France
Marc.Boyer@enseeiht.fr

April 1, 2004

Abstract

Generic programming is a old issue in programming. This paper presents a way to get a macro based version of the STL for C programmers, designed to be usable for beginners also, without sacrificing type-checking neither performance issues. The result is the (under development) BPL: Basic Preprocessed Library. It offers container for build-in or user-defined types. It is robust enough to support itself (it allows vector of vector of `int` for example).

Contents

1	Motivations	2
1.1	Other approaches	3
1.1.1	Genericity trough <code>void*</code>	3
1.1.2	Genericity with macros	3
1.2	Quick overview	4
1.3	Why STL?	5
2	User manual	5
2.1	Compilation process step by step	6
2.1.1	Instantiating it	6
2.1.2	Compiling it	7
2.1.3	Using it	8
2.2	The BPL data model	9
2.2.1	Error handling	9
2.2.2	What if build failed?	11
2.3	Iterators and algorithms	11
2.4	Container of container	11

3	Implementation	12
3.1	Having a generic container	12
3.1.1	Avoiding the use of macro while coding a container	13
3.1.2	Cleaning the macro name space	13
3.2	Having generic iterators	14
3.3	Having generic algorithms	14
3.4	Specialisation	15
3.5	Inlining	15
3.6	Summary	15
4	Conclusion	17
A	Macro concatenation	17
B	Under development	18
C	Copyright policy	18

1 Motivations

I am amazed by the number of C programmers that rewrite from scratch some very basic (generic) containers, lists, stacks and so on. Even worst, some are using unsuitable solution when the good one is too hard to develop (using linked lists when rb-trees would be better for example).

It contrasts with languages where genericity is a built-in feature (like Eiffel, C++), and the language includes a collection of generic containers.

So, it seems that there is a need of a library offering generic containers. In fact, there are some, but, as discussed in section 1.1, I think that there is a need of a generic library, ensuring type-checking, and simple to use, even for beginners.

The ideal way, to get genericity, is to take a language that offers it to the programmer, like Eiffel, Ada, C++. Nevertheless, sometimes you can't. In C, there are basically two ways:

- using the preprocessor,
- casting (in general to `void*`)

Using the preprocessor has some well known writing and debugging drawbacks. But casting to `void*` can be worst, because you lose the type checking and some optimisations.

So, I was wondering if it was possible to write a macro-based library, that offers the same kind of functionality that the C++ STL. The result is the BPL, presented here.

Notice that solving the C generic programming problem is out of the scope of this paper. It only combines some well-known techniques to offer a “clean” set of containers, easy to use.

Another (more personal) goal was to have a “clean C” code. I tried to get the less warning messages with `gcc -Wall -ansi -pedantic`.

1.1 Other approaches

I am not the first at all to want to have genericity in C, but the other solutions does not satisfy me, as user of the C++ STL and teacher for beginners with C.

1.1.1 Genericity trough void*

One common way to get genericity is to write a library handling some `void*` pointer, and to let the programmer gives you the size of the object (to be able to store the object and not just a pointer on it), the useful functions (comparison), and so on.

For example, if you are writing a list container, it should be like:

```
void insert(List* list, void* element, size_t elemSize);
...
char name[64];
insert(li, name, sizeof name);
sorted_insert(li, name, sizeof name, strcmp);
```

You can do a bit better, storing the size of handled elements, the useful functions in the container itself, giving it only once, at initialisation time.

```
List stringList;
initList(&stringList, 64, strcmp, strcpy);
```

But, you loose *type checking* and perhaps *some optimisations*.

The worst is the type checking problem. Because of the `void*` interface, you can add anything in any such generic container. While reading it, you should think that you are a good professional C programmer, not one of my student, and you never add an `int` (or `int*`) into a list of strings. Yes, but did you never confused a `char*` and a `char**`, or a `list*` and a `node*` (furthermore if `list` is a simple typedef to `node*`)?

The other problem is a performance problem. If you are handling some small type, did-you need the `memcpy` to be called each time one of your type is copied? Is there no difference between `a=b` and `memcpy(&a, &b, sizeof b)` with you compiler? The same way, if you are using a numerical type, (you are handling a sorted list of `pid`, that are typedef to `int`), did you want to write a `pid_lesser` function, or just `<`, and are you sure your compiler is clever enough to replace every call of `pid_lesser(a,b)` (given as a function pointer at list initialisation), by `a<b`?

1.1.2 Genericity with macros

To keep type checking at compilation time, I see no other solution that the use of the preprocessor and macros¹.

Nevertheless, writing (and debugging) code using macros is sometime a hard task. So, I was trying to use the less macros as possible, to avoid the 10-lines long macros

¹Even if, I could make mine the words of B. Stroustrup “The irony is that I dislike most forms of preprocessors and macros” [The Design and Evolution of C++, § 3.2.1].

(or worst), to keep the “macro name space” as small as possible, and to be simple enough to be used by beginners.

This work is of course inspired by the well known `generic.h` file used in old `libg++` distributions, and also by the `00PC` project of Laurent Deniaud².

In the `generic.h` approach, when writing a generic code, you have to write all your code as a (huge) macro. I try to avoid it. Moreover, if you are a simple user of the generic code, you can declare a queue of double just with `queue(double) Q_double`, and insert elements into a queue with code like `Q_double.insert(1.0)`. In C, you should write something like `insert(&Q_double, 1.0)`, but, as there is no overload of function, this is allowed only if you have one single instance of the code of `insert`. If you want to be able to get a queue of double and a queue of int, you have to do some mangling (and my solution is to write something like `insert_dbl(&Q_double, 1.0)`).

In the `00PC` approach, the genericity is included in a more general object-oriented framework. My purpose was to keep it simple, and accessible for beginners. If you want a framework with objects, inheritance, polymorphism, and exceptions, you should have a look on `00PC`.

The data model of BPL is simpler, in order to be accessible for beginners, just assuming that a type can be initialised, destroyed and copied (see sections 2.2 for details). Error handling is simply done by return value of functions.

1.2 Quick overview

Let gives you a quick overview of how to use the BPL. If you want to declare a vector of int, the user should write a `VectorInt.h` header file, that looks like:

```
#ifndef VECTOR_INT_H
#define VECTOR_INT_H

#define BPL_TYPE          int
#define BPL_CONT_SUFFIX  Vect_
#define BPL_TYPE_SUFFIX  int

#define COPY_TYPE(X,Y)    ((X)=(Y))
#define INIT_TYPE(X)      NULL
#define DESTROY_TYPE(X)   NULL
#define INIT_COPY(X,Y)    COPY_TYPE(X,Y)

#include "VectorGen.h"
#endif
```

Then, here is a simple code that use this vector of integers.

```
#include "VectorInt.h"
#define TEST_SIZE (16U)
void foo(){
    Vector_int vi;
    int i;
    initWithSizeVect_int(&vi, TEST_SIZE);
    for(i=0; i<TEST_SIZE; ++i){
        putVect_int(&vi, i, i);
```

²<http://ldeniau.home.cern.ch/ldeniau/html/oopc/oopc.html>

```

}
for(i=0;i<TEST_SIZE;++i){
    printf("%d :", getVect_int(vi,i) );
}
destroyVect_int(&vi);
}

```

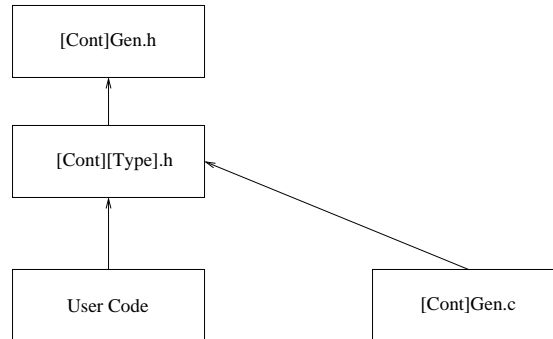


Figure 1: Inclusion architecture

The inclusion architecture is described in Figure 1. The user instantiates a `[Cont] [Type] .h` file, defining the handled type and the associated operations. This file includes the `[Cont]Gen.h` which defines the type(s) and the functions associated to this container. As usual, the user code includes this header, and the generic code also does.

1.3 Why STL?

BPL is directly inspired by the STL. Even the name. BPL can be read as “Basic Preprocessor Library” (as STL is the Standard Template Library), or “Boyer Practice Library” (as STL was also Stepanov and Lee Library, and “Practice” comes from the fact that, after one year of teaching C to very first beginners, and I was wondering if I was still able to code a tree), or “Beginner Purpose Library”, since, as a teacher, my goal was to make a library that can be used by beginners.

But why did I chose to make a clone of the STL and not another set of container with another interface?

The first reason is that, as C++ user, I am used to use the STL. The second is that, because STL is standard, given with every compiler, it is widely used and well known.

2 User manual

Given that this documentation is beginner-oriented, the user manual begins with a short tutorial-like (2.1), that is sufficient to have a basic use of the BPL (like vector of `int`).

2.1 Compilation process step by step

To use (*i.e.* instantiate) a generic container, the first step is to instantiate the generic parameters, the second is to compile it, and the third is to use it.

2.1.1 Instantiating it

Instantiating the generic is simply done by copying the file `[Container]Type.h`, into `[Container][Instance].h`. Then, in the file `[Container][Instance].h`, the user should give the definition of some macros.

There are three types of input macros used to instantiate a generic container:

1. The *type* macros, defining the types handled by the container. A simple container (*sequence* in the STL vocabulary), like a vector, just needs one type, (the contained type) as an association container, like a map, needs two types (the index and the contained types).
2. The *functions* macros, that are a minimal interface to handle the types. If you want to have a sorted list, you must give the comparison criterion. Some default values are provided for these macros.
3. The *naming* macros, are used to give the user the choice of the mangling of functions names. Let me first describe the problem: in the same code, you can have a vector of `int` and a list of `float`, what could be the name of the inserting function? In C, they can not both be called `insert`. One could be called `insertVect_int`, and the other `insertList_float`. But, how to name a vector of `char*`? `insertVect_char*` is not a valid name for a function. So, you have to define the suffix that will be used. Moreover, if you have not automatic completion on your editor, you perhaps prefer writing `insertL_float`, or `fl_insertV` instead of `insertList_float`. That is the role of the naming macros.

Of course, if your types or the functions used are defined in other headers, you have to include these headers.

example *The Vector container.* To use the vector container, the user should instantiate:

There is only one type macro:

`BPL_TYPE` The type that will be handled, like `int`, `unsigned long`, `char*` or any user-defined type. No default value.

There are two naming macros³:

`BPL_TYPE_SUFFIX` The suffix used to mark out the type. Remember that this value must be a possible suffix of a C identifier. It means that you must not define `BPL_TYPE_SUFFIX` as `char*` for example. If you need to handle pointers on `char`, you can define `BPL_TYPE` as `char*` but `BPL_TYPE_SUFFIX` should then be `pchar`, `charPtr`, or anything else that means to you “pointer on `char`”, but not `char*`! No default value.

³Actually, there is a third naming macro, `INST`. See page 13

`BPL_CONT_SUFFIX` The suffix used to mark out the container. Default value is `Vect_`

And there are some (?) functions macros. If you are handling a simple build-in type, all default values should be correct. Either, have a look on the section 2.2. Here is just a short description of some of them.

`INIT_TYPE(X)`, `COPY_TYPE(X,Y)`, `DESTROY_TYPE(X)` Should be defined to have respectively the semantic of constructor, assignment, destructor of the type. Returns 0 if fails. Default values are:

```
#define COPY_TYPE(X,Y)      ((X)=(Y),1)
#define INIT_TYPE(X)        (1)
#define DESTROY_TYPE(X)     (1)
```

`CMP_TYPE(X)` Returns the comparison of two data of the handled type, with the same conventions that `strcmp`: value is 0 if `X == Y`, `< 0` if `X < Y` and `> 0` if `X > Y`. Default value is `((X)-(Y))`

2.1.2 Compiling it

Here is the Achilles' heel of the BPL. The standard solution is not very sexy (but there is one with the `-include` option of `gcc`).

For a given container, the generic code is in a file named `[Container]Gen.c`. The problem is: how to compile the generic code with a given instantiation?

One solution (it is the way it is done in most C++ compilers) is to include the code from the header: just adding a `#include "[Container]Gen.c"` in the files `[Container][Type].h`. But I try to avoid. I like to have a file `[Container][Type].o` that is the compiled code relates to the `[Container][Type].h` instantiation. Moreover, I use a lot of `static` functions, which is incompatible with multiple inclusion of the same generic code⁴.

So, the question is, how to get a several `[Container][Type].o` compiled file, from several `[Container][Type].h` (user defined instantiation) and a single `[Container]Gen.c` (the generic code)?

The solution is to use the `-D` option and a `[Container]TypeSwitch.h` file. The implementation `[Container]Gen.c` includes this switch include, where are registered all the instantiation you are using of the container. The command line option is used to select the right instantiation, as presented in figure 2. The switch file looks like:

```
#ifndef VECTOR_INT
#include "VectorInt.h"
#else
#ifdef VECTOR_STRING
#include "VectorString.h"
#endif
#endif
#endif
```

and the compilation line looks like:

```
VectorInt.o: VectorInt.h VectorGen.c
$(CC) $(CFLAGS) -c -DKEEP_INTERNAL_BPL_MACROS \
-DVECTOR_INT VectorGen.c -o $@
```

⁴Nevertheless, as presented in Section 3.5, for optimisation reasons, some code is inserted...

Be aware that, with help of the `-I` option, you are not forced to have one single `[Container]TypeSwitch.h` on all you system. You can have one per projet, or one per subpart of the project⁵.

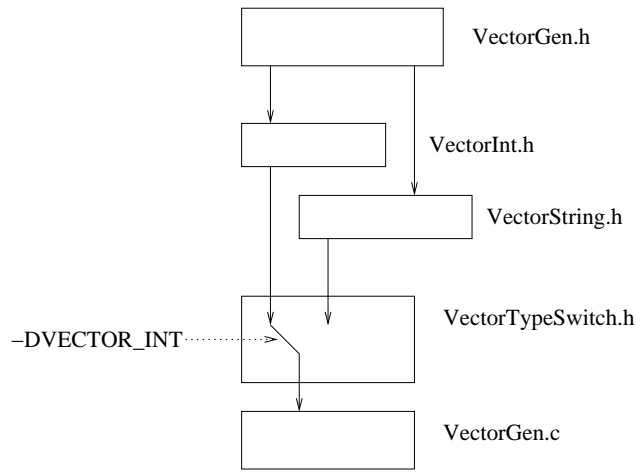


Figure 2: The switch of headers with the `-D` option

The drawback of this solution is to force the programmer to write code (the `[Container]TypeSwitch.h` file) for solving a compilation process problem.

Relying on some compiler or platform specific tool, you can keep an empty `[Container]TypeSwitch.h`.

For example, one solution is to use a gcc option: `-include`. With that, my compilation rules in makefile are:

```

VectorInt.o: VectorInt.h VectorGen.c
    $(CC) $(CFLAGS) -c -DKEEP_INTERNAL_BPL_MACROS \
    -include VectorInt.h VectorGen.c -o $@
  
```

Another solution, if you are in a Unix-like environment, is to use the `sed` tool. In the `[Container]Gen.c` code, there is one tag `/*sed-tag-for-include*/`. You can call `sed` with command like

```
s/\/*sed-tag-for-include*\/*\/*#include "[Container] [Type].h"/
```

2.1.3 Using it

To use an instantiation of a generic container, it is very simple. You just have to include the instantiation headers you need. Then, you can use all the interface of the container, *i.e.* the type and all the methods defined in the `[Container]Gen.h`, with the naming conventions described bellow.

As presented in section 2.1.1, in the instantiation process, you have defined two naming macros: `BPL_CONT_SUFFIX` and `BPL_TYPE_SUFFIX`. Then, for each function (or

⁵BPL is a prototype for the moment, and all files are in a single directory, but, in a real developing environment, the generic codes should be in a “common” directory, and only the instantiation files are in the project directories. The switch file is one of the instantiation files.

type) `INST(toto)foo` defined in the interface of the container (or in files `iterator.h` or `algorithms.h`, see section 2.3) you can use the function `foo##BPL_CONT_SUFFIX##BPL_TYPE_SUFFIX` (`##` is the concatenation operator), and the same for the types. The name of the container type is the concatenation of the kind of container (`Vector`, `Map`) and the `BPL_TYPE_SUFFIX`⁶.

example If you are using a vector of `int`, keeping the default definition for `BPL_CONT_SUFFIX` (`Vect_`), and giving value `Int` for `BPL_TYPE_SUFFIX`, you can define variables of type `Vector_Int` (the container) and `itVect_Int` (iterator), and call functions `initVect_Int(Vector_Int*)`, `int getVect_Int(Vector_Int, size_t)`...

You can also define a vector of `char*`, with value `V` for `BPL_CONT_SUFFIX` and `pc` for `BPL_TYPE_SUFFIX`. Then, the types are `Vector_pc` and `itVpc`, and the functions signatures are `initVpc(Vector_pc*)`, `char* getVpc(Vector_ps, size_t)`...

Notice there also the interest of having a naming convention that is not only based of the instantiation types. If you use a map as a phone book, with `char*` as key type and `struct address` as data type, the suffix `PhBook` can be simpler than `pChar_sAddress`.

Of course, to be able to run the program, you should link it with the `[Container] [Type] .o`.

2.2 The BPL data model

The BPL offers a set of container for build-in or user-defined types. The problem was: how to keep general enough to allow advanced use (object or abstract data type model) without frightening beginners or forcing the user to write tens lines of macros to have a vector of `int`?

The BPL data model is a kind of “minimal” data model. The user is not forced to gives a full interface, but just the minimum needed to initialise, copy and destroy the data. For build-in types, initialisations and destruction are `nop` and the copy is just `=`. But for more elaborated ones, you can need to get or free ressources, count references and so on.

This interface should be given by defining the macros in the instantiation file `[Cont]Type.h` like `INIT_TYPE`, `DESTROY_TYPE`...

A set of default value is given, that should be sufficient for build-in types.

2.2.1 Error handling

The notion of construction, destruction, copy are basic ones nowadays. The only thing to be clarified is the error handling. What if the construction of an object fails? In C++, such kind of error is managed by exceptions. The historical point of view in C seems to be “return value” based.

It was a design choice: exception *vs.* return value.

⁶This mangling sheme can be changed, by defining your own `INST(toto)` macro. See section 3.1 for details.

For sake of simplicity, to keep it accessible for beginners, there is no exception support in the BPL. The constructor, copy method, are expressions that should return a value, 0 if the operation has failed, something else otherwise⁷. Notice that the destruction has no return value, it should not fail. I have followed the C++point of view that forbid the thrown of exceptions by destructors.

example What could be the value of `INIT_TYPE(X)`? Here are some example.

```
#define INIT_TYPE(X)      (1)                /* 1 */
#define INIT_TYPE(X)      (X)=malloc(128U)   /* 2 */
#define INIT_TYPE(X)      (X)=NULL,1         /* 3 */
#define INIT_TYPE(X)      initVect_int(&(X)) /* 4 */
```

The default value, on the first line, is just the value 1. It is a nop, that always succeeds.

If you are handling some buffer, you can make a call to `malloc` (see 2). The value of the expression is the return value of `malloc`, `NULL` if `malloc` fails (convertible to 0).

If you are handling some pointer, assigning `NULL`, to each created pointer could be a good idea. Unlike the previous example, the value of the assignment is 0, but this is not a failure. To do so, you could write a short function that assign `NULL` to its parameter and returns 1, or use the “,” operator, like in 3.

Keep in mind that `INIT_TYPE` is a macro, that will be called with a variable name. Then, if you want to call a function that need a pointer on your variable, you can call it with `&(X)`, like in 4, that is part of the definition of a vector of `int` (yes, it works, like the `vector< vector<int> >` in C++).

The code of the copy follows the same ideas. The default value is

```
#define COPY_TYPE(X,Y)      ((X)=(Y), 1)
```

Just one particular point with the default destructor. For simple types, the destructor is a nop with no return value. What could then be the definition of `DESTROY_TYPE(X)`? I should have introduced a condition mechanism, based on `#ifdef INIT_TYPE`, but it is source of confusion. I prefer to rely on compiler optimisations. If the constructor does nothing, then, write a nul-effect instruction⁸. If your compiler does not remove this dead code, then, change you compiler, or patch the BPL code.

```
#define DESTROY_TYPE(X,Y)      ((void) NULL)
```

⁷This convention does not allow to have various error return code, but this is the C paradigm.

⁸Remind that `while(...) (void) NULL;` is a valid C code. The `void` cast is just there to tell the compiler that you want to ignore this value, and then avoid a warning message.

2.2.2 What if build failed?

The default contract with the constructor-destructor model is that, you should not use a data before its successful initialisation. Keep in mind that even destruction is not permitted before initialisation.

But what if building fails?

Using the Ressource Acquisition Is Initialisation (RAII) paradigm, the construction will try to acquire ressources, and then, can fail. In BPL, the constructor should then release the aquired ressources, and return false.

The destructor will not be called by the BPL containers if the constructor has failed. From the BPL data model point of view, the only operation that is possible on a data which initialisation has failed is another initialisation.

This was important to say because, for some programmers, if the constructor fails, it should let the data in a “default safe state”, that allows at least to call the destructor (like the couple `malloc/free` for example). BPL does not have such requirement (See “The C++programming langage”, Appendix E.3.5.2⁹ to get a more detail justification of this point of view by Bjarne Stroustrup.).

This BPL choice can have some drawbacks (the part of the code of the constructor that handle errors can be common with the destructor), but, the penalty for people handling object model with obvious default state must be balanced by the penalty for people handling object model with non obvious default state.

2.3 Iterators and algorithms

When you instantiate a container, you also instantiate two types: the forward iterator `INST(it)` and the reverse iterator `INST(rit)`, with the associates functions: incrementation (the `++` of C++), access to the value, `begin`, `end`....

They are all defined in `iterator.h`

Notice that you always can copy an iterator with `=`. The iterator have a value semantics.

With this iterators, you do not get the full power of the C++ iterators, since, for the moment, you can not use one instantiation of iterator with another instantiation of container (to copy a subpart of a `list<int>` to a `vector<int>` for example).

You also get the set of algorithms that have been defined in `algorithms.h`.

2.4 Container of container

The compilation models allow you to have a container of container (in the distribution, you can find a vector of vector of int).

Nevertheless, for some internal details, you should include `Clean[Cont]Macros.h` just after the include of the first-level container in the second-level container instantiation. That is to say, the file `VectorVectorInt.h` looks like

```
#include "VectorInt.h"
#include "CleanVectorMacros.h"
#define BPL_TYPE Vector_int
```

⁹This appendix can be downloaded from the WEB site of the author.

```

#define BPL_CONT_SUFFIX Vect_
#define BPL_TYPE_SUFFIX Vect_int
...

```

3 Implementation

Genericity can be shortly defined by writing a “partial” code, *i.e.* a code that manipulate types, functions and constants (generic parameters) that will be defined at instantiation time. Genericity is a static polymorphisms, in the sense that the instantiation is done at compilation time (dynamic polymorphisms, like inheritance, allows the code to be instantiated at run time).

In a macro-based genericity, the generic parameters are macro *names*. To instantiate a generic parameters, the user should *define* this macro with the good value.

The BPL approach is to put each generic code into a file. To instantiate it, the user has just to declare the parameters macros, and to include the matching header.

In other words, a generic code as some *input macros*, which are the generic parameters (types and functions¹⁰). To instantiate a generic code, the user should *instantiate* these macros¹¹. The instantiation is done in a single header file (one per instantiation). The generic code, header and implementation, must be compiled with these input macros.

3.1 Having a generic container

As the first step, look how to define the simplest generic thing: a type with a number as generic parameter: we would like to code something like a string of fixed size. The code of `StringGen.h` could look like:

```

/* Input generic macro:
 * STRING_SIZE: size of the string */
typedef struct {
    char val[STRING_SIZE+1];
} string;
fscan_string(FILE* f, string* s);

```

But, if we want to allow the user to have different instantiations of the string with different size, we should gives a different name to the different types and functions. It could be done with `typedef struct {...} CONCAT_MACRO(string,STRING_SIZE)` and `CONCAT_MACRO(fscan_string,STRING_SIZE)(FILE *f, string *s)`¹². Then, the following code

```

#define STRING_SIZE 64
#include "StringGen.h"
#define STRING_SIZE 128
#include "StringGen.h"

```

will define two new types: `string_64` and `string_128`, and two functions `fscan_string_64(FILE* f, string_64* s)` and `fscan_string_128(FILE* f, string_128* s)`. It works, but this is not very

¹⁰or macros with parameters

¹¹Some (a few) of these macros can have a default definition (cf. `INST`)

¹²Where `CONCAT_MACRO` is a macro that concatenates macros. Details are given in Appendix A.

user-friendly, as the user perhaps want to be able to change the size of string without changing all names. Moreover, the suffix `string_64` can be found too long (especially if you do not have automatic completion in your code editor). At least, imagine that the generic parameter is not a numerical value but a type, and you want to instantiate it with `unsigned long`. Because this type is not a single word, this naming scheme will fail. Then, in addition to the generic parameter itself, the BPL offers the user the way to chose the suffix that will be added to the function names, with two macros, that are concatenated at the end of the function names. The example given in Section 1.2 will define functions like `initVect_int`.

To add one level of generality, and to quicken writing of generic code, an instantiation macro `INST` have been defined. Its definition for vector is

```
#define INST(FCT_NAME) CONCAT_MACRO3(FCT_NAME, BPL_CONT_SUFFIX, BPL_TYPE_SUFFIX)
```

Then, each function is declared as `INST(fctName)`.

Notice that the user can also define its own `INST` macro. The definition given there is just a default value.

3.1.1 Avoiding the use of macro while coding a container

I need macros to have type-safe genericity, but I try to use it as less as possible. The BPL compilation model help to do so. Each generic implementation file has a certain number of parameter macros. Some are type names. Then, you can use `typedef` to avoid to use it. The scope of the `typedef` is local to the implementation code. Then, this code begins with some `typedef` used to rename the macros. It avoids the use of the macros in the code itself, and helps to detect a error in macro declaration.

For example, the file `VectGen.c` has one `BPL_VECTOR_TYPE` macro, which is the mangled name of this vector instantiation. Then, one of the first lines of code is:

```
typedef BPL_VECTOR_TYPE Vector;
```

3.1.2 Cleaning the macro name space

Defining macros creates a problem of name collision. This problem is both internal to the library (a vector of `int` and a vector of `char` will use the same macros names, with different values), and external to the library (what append if the macro name is already used somewhere else in the code?).

The internal collision problem is the simplest: you can undefine a macro before using it¹³. Nevertheless, it does not solve the external collision problem. If the same macro is use somewhere else with another semantics, this solution erase it, without any warning.

My solution is not to protect my code from collision, but to clean the name space of the macro I used (including generic parameters and internal macros). Then, if one macro is already used, your compiler should complain, and you will have to do some corrective action (swapping the header inclusions, changing some macro name, etc.)

```
#define BPL_TYPE int
#include "VectorGen.h"
/* Here, there is no more BPL_TYPE macro */
```

¹³The classical idiom is: `#ifdef F00 #undef F00 #endif`.

To make it more clear, for each "`[TYPE]Gen.h`" header, there is one "`Clean[TYPE]Macros.h`" file, in charge of this cleaning action.

It should be noticed that, sometime (for implementation purpose), the system need to keep the macro definitions. Then, a dedicated `KEEP_INTERNAL_BPL_MACROS` macro is defined as compilation option.

This solution as a drawback: when instantiating a container of container (or worst, container of container of container), the system should instanciate all the macros, clean it and build new ones. Then, when compiling the instantiation code of a container of container, the macro `KEEP_INTERNAL_BPL_MACROS` is defined, and then, the macro space name is not cleaned before being used once more. So, the user should itself includes the `Clean[Cont]Macros.h` file, as presented in section 2.4.

I am currently working on solution based on counting with the preprocessor, but there is nothing really clean for the moment.

3.2 Having generic iterators

One important idiom of the STL is the notion of iterator.

Then, the file `iterator.h` offers two iterators types (simple and reverse). When instantiating a container, the system also instantiates two iterators (with the same naming mangling convention) and a basic set of operations.

One translation problem from the C++ STL to the C BPL is that, the C++ iterator interface makes a wide use of operators. Then, `iterator.h` offers functions like `itInc` that increments an iterator and so on.

From implementation point of view, `iterator.h` has three input macros:

`INST(fct)` The mangling macro

`BPL_CONTAINER_TYPE` The container type name

`BPL_TYPE` The name of the type stored in the container

The mangling system is the one of `INST(fct)`.

Each container header includes the iterator header, with the right input macro values.

The container implementation is in charge of the implementation of the functions of `iterator.h`.

3.3 Having generic algorithms

With iterators, algorithms are the third pillar of the STL, the set that reveals the flavour of containers...

In BPL, algorithms are implemented using two files: one header file `algorithms.h` and one implementation file `algorithms.c`, with the same input macros than iterators (`INST(fct)`, `BPL_CONTAINER_TYPE`, `BPL_TYPE`).

One main difference between iterators and algorithms is that, for algorithms, it exists a generic implementation code (for iterators, the code was in the code of the container), which is included by all container code.

The generic code is a default implementation, and, using the specialisation mechanism, each container can implement a more efficient code for each algorithm.

3.4 Specialisation

One important feature of C++ template is the notion of specialisation. But genericity is not a built-in feature of C, and neither specialisation...

With macros and conditional compilation, a small aspect of specialisation can be done: the default code must be protected by a compilation condition (`#ifndef`) and a coherent naming convention.

Then, in the file `algorithms.c`, each function `foo` is protected by a guard:

```
#ifndef BPL_F00_IMPL
void INST(foo)(...){
    ...
}
#else
# undef BPL_F00_IMPL
#endif
```

And if a container defines a specialisation of one algorithm `foo`, it just defines the macro `BPL_F00_IMPL`. Pay attention to the `# undef BPL_F00_IMPL`, which cleans the macro space name, and allows to compile another specialisation of the same algorithm if needed.

3.5 Inlining

All what have been presented gives a functional code, respecting the type-checking constraint, but not so efficient as C programmers like a code to be. As a simple example, the access to an element in a vector calls a function `get`, but most of C programmers does not want to pay the cost of a function call when accessing an element of an array.

The classical historical solution in C is to use macro, but the use of macro is not type-safe... By chance, the C99 defines the `inline` keyword for this purpose.

The question is then: `static` or not `static`?

I have chosen `static inline` for two reasons: firstly, with my knowledge of compilers, I do not expect a lot of C compiler to be able to inline a function defined in another compilation unit; secondly, some C compilers were able to inline static functions before the introduction of the `inline` keyword, and I wish people to be able to use BPL without having a C99 compiler.

Notice that, with a simple `#define inline`, the keyword can be hidden, schitching, in this case, from C99 to C90 with the same semantics (in this example).

3.6 Summary

This section sum up all the architecture of the system.

For each container, the core of the system is the pair `[Cont]Gen.h`, which declares the generic interface of the container, and the file `[Cont]Type.h` which is a template

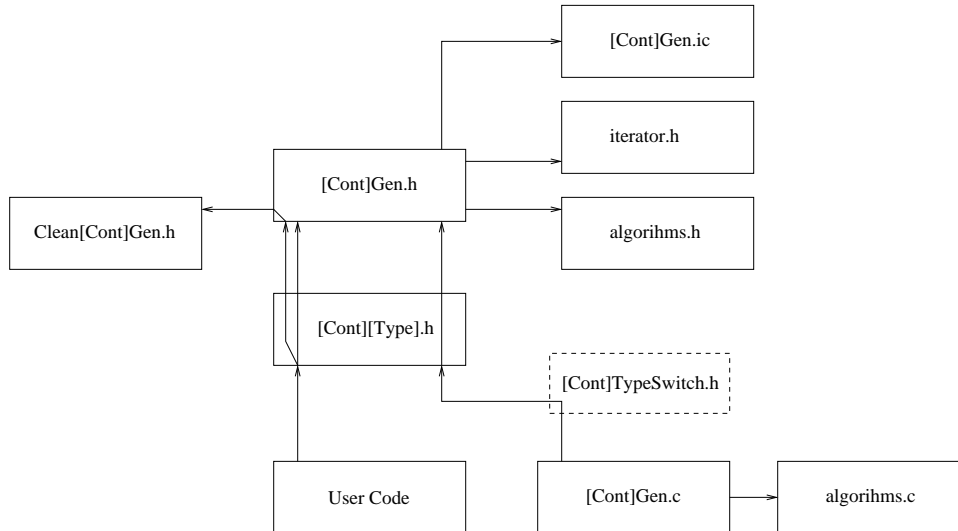


Figure 3: Global inclusion architecture

of instantiation, that must be copied into a `[Cont][Type].h` for each instantiation with type `[Type]`, and where some macros are defined and instantiated.

This `[Cont]Gen.h` file is included by the `[Cont][Type].h`, but `[Cont][Type].h` is, from logic point of view, an input for `[Cont]Gen.h`: `[Cont][Type].h` defines the macros needed by `[Cont]Gen.h`. The code `#include "[Cont]Gen.h"` is one of the last line of `[Cont][Type].h` (opposite to the common use).

There are two other special generic headers: `iterator.h` and `algorithms.h`, declaring respectively the set of functions necessary to handle an iterator (notice that the iterator type itself must be defined by the container in `[Cont]Gen.h`) and the set of algorithms. So, when declaring a container, you always get the set of algorithms¹⁴.

Then, depending on the fact that we are compiling user code or the implementation of the container (the switch is done with the `KEEP_INTERNAL_BPL_MACROS` macro), there are two cases.

When compiling user code, a file `[Cont]Gen.ic` containing the definition of function to be inlined is included, and, at last, file `Clean[Cont]Gen.h` is included to clean the macro name space (and then allow the same macros to be used to instantiate the container with another type).

When compiling the implementation, these two files are simply not included, because the implementation needs to know the instantiation macros values. Notice also that the implementation does not simply defines the functions declared in `[Cont]Gen.h`. It is also in charge of defining the functions declared in `iterator.h`. Moreover, it

¹⁴In C++, you can choose to include `algorithms.h` or not, but, with this macro-based genericity model, making the link between the generic code and the instantiation macros would have been worst than systematic include.

includes `algorithms.c` where are defined some default implementation of the algorithms, based on the iterator interface. The container can also defines a more efficient implementation, using the specialisation mechanism described in Section 3.4.

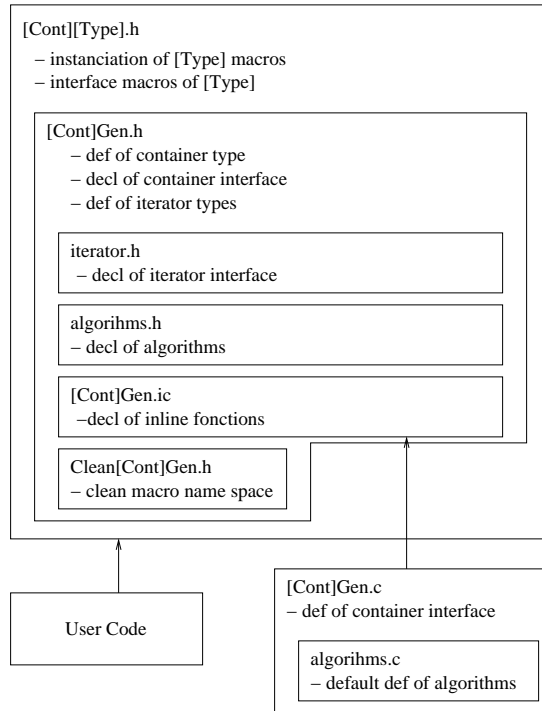


Figure 4: Global inclusion architecture – merged point of view

Figure 4 presents this architecture from the preprocessor point of view, *i.e.* once the `#include` directives have been replaced by the included file.

4 Conclusion

I was wondering if a generic clone of the STL, efficient and type-safe, was possible in C. To do so, I had to solve some technological solutions. The choices done are presented in this paper.

So, it is possible. I have written enough code to test all my ideas. Now, there is only thousands of line of code to write to get a full clone of the STL...

A Macro concatenation

To concatenate string, the preprocessor offers the operator `##`. Then, a `## b` is replaced by `ab`, even if `b` is a macro. To be able to concatenate the *value* of a macro,

you should use a two phases solution:

```
#define CONCAT(X,Y)      X ## Y
#define CONCAT_MACRO(X,Y)  CONCAT(X,Y)
```

In this solution, the call of `CONCAT_MACRO(A,B)` will be replaced by `CONCAT(VA,VB)` where `VA` is replaced by *the value* of `A` if `A` is a macro, and `A` either (the same for `VB`).

B Under development

The BPL is still under development. If you need some feature, you can either write it (all details are given here) or send me an email. I can do it on free time.

C Copyright policy

I have not any idea now of the kind of licence that would be usefull for this code. So, this is a tempory licence. If you are interested as user, please email me.

The BPL is free for educational purpose and non profit organisations. You are free to use this code, to distributute it, to change it for your own use, to distribute binaries made using part of this code or changed one.

You are not allowed to distribute a modified version of this code.