Intrusive Data Structures

Jiri Soukup, Code Farms Inc.
7214 Jock Trail, Richmond, Ont., Canada, KOA 2ZO
613-838-4829, fax 613-838-3316, jiri@codefarms.com

ABSTRACT

This article compares two styles of building data structures and data structure libraries in C++: (a) Intrusive data structures formed by pointers stored inside the application objects, (b) Containers where auxiliary objects form the required data structure, and only point to the application objects without adding any pointers or other data to them.

Each style works better in different situations, and the first half of the article discusses the impact of this choice on program performance, code maintainability, ease of use and data integrity. When working with templates, containers are easier to implement, which may be the reason why most class libraries are based on containers. The only existing library which is consistently intrusive (Code Farms) uses a code generator. If intrusive data structures could not be implemented with templates, their applicability would be severely limited. The second half of the article deals with this pivotal question, shows an elegant way of building an intrusive data structure library with templates, explains why its interface is similar to- but cannot be identical with STL, discusses the impact of the new architecture on class dependency, and compares the new approach with existing libraries. A prototype of the new library is now available under the name Pattern Template Library - see [15].

Chapter 1: Introduction

Open any book on data structures, and somewhere near the beginning you will find a linked list similar to Fig.1a. This is the type of list you would probably use in your program if you were not using any class library.
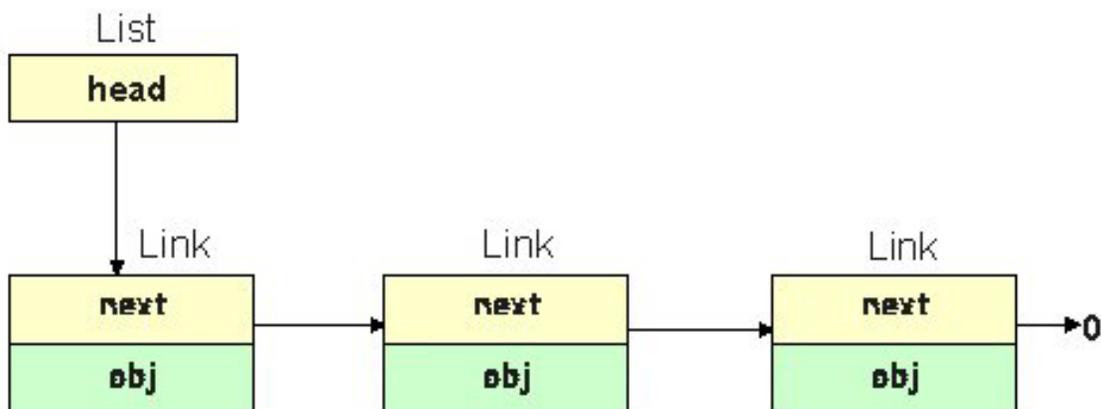

Fig.1a: Different implementations of the abstract list: intrusive linked list.
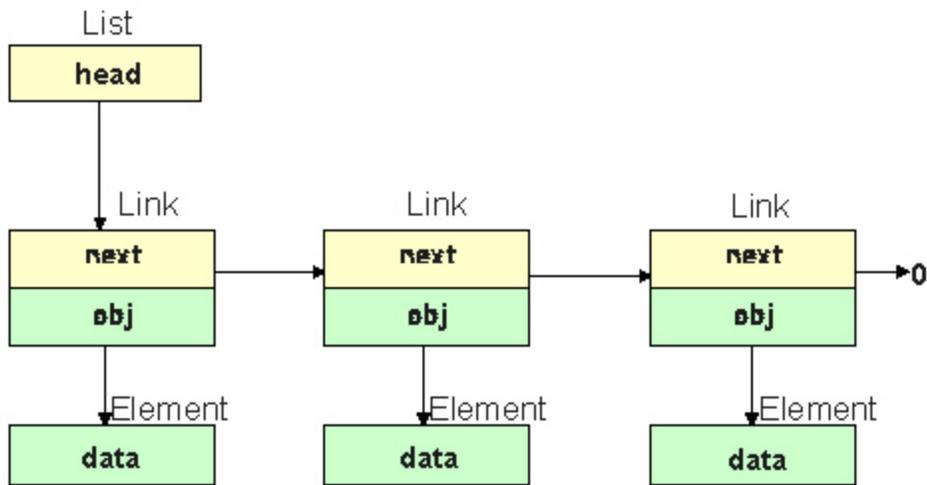
Fig.1b: Different implementations of the abstract list: non-intrusive linked list.
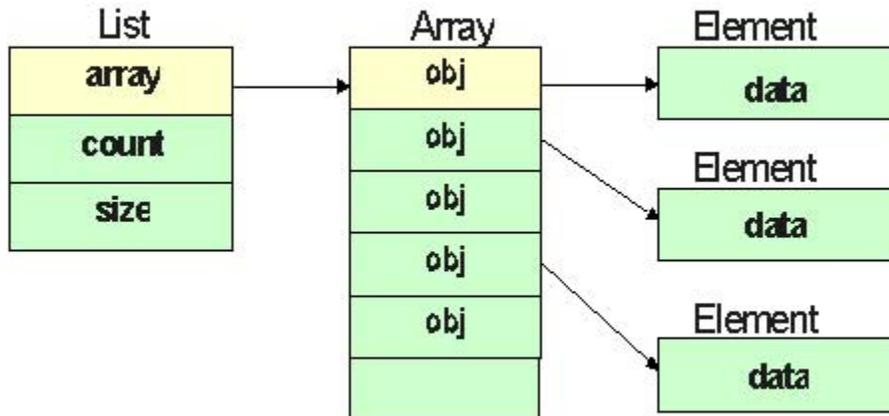

Fig.1c: Different implementations of the abstract list: array-based non-intrusive list.

For example, if each Department has Employees, and each Employee belongs to exactly one Department, you would write it like this:

```
class Employee {
  Employee *next;
  ...
};
class Department {
  Employee *listHead;
  ...
};
```

This type of list is called an "intrusive list", because the pointers that connect it are embedded in the participating objects. [According to Webster, intrude = 1. to thrust, enter or force in or upon, 2. encroach, trespass - intruder - intrusion - intrusive.]

Compare this implementation of the linked list with the one shown in Fig.1b. Here auxiliary objects of type Link form the list; each Link has a pointer that leads to the object that participates in the list. You definitely need this type of list when an Element may participate in an unknown number of lists; see Fig.2.
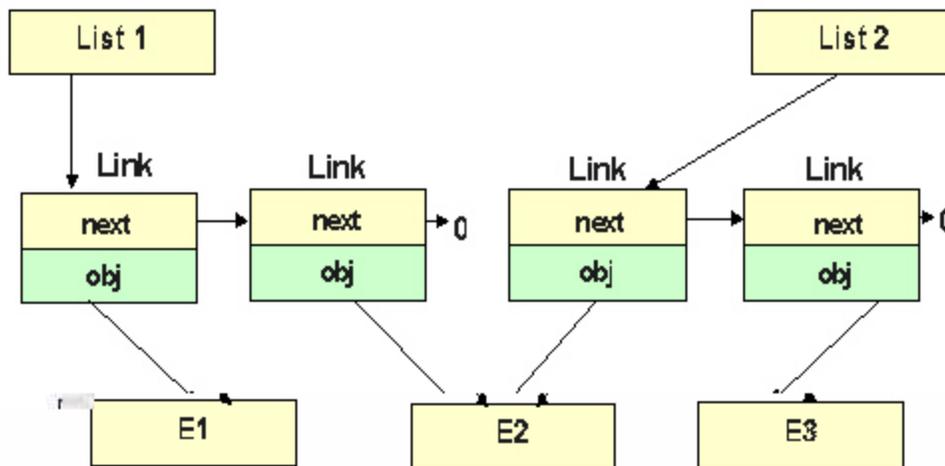

Fig.2: The same object (here Element E2) can be on more than one list.

For example, suppose a Student may take 1 to 6 Courses, and we want to maintain a list of Students for any given Course:

```
class Student { ... };
class Link {
   Student *obj;
   Link *next;
   ...
};
class Course {
   Link *listHead;
   ...
};
```

The most striking feature of this arrangement is that the class Student contains no list-related pointers or other members. You can add such a list to your program without introducing any changes to the class.

All this seems so trivial that you may wonder why we are spending so much time discussing it. However, there is more here than meets the eye. The problem is that the choice of implementation has numerous consequences which are often neglected. For example, an

intrusive implementation is computationally more efficient, and permits bi-directional access between the classes involved, as will be explained in Chap.7. Note the uni-directional access in the last code sample: Every Course knows its Students, but Students do not know the Courses they are taking. The intrusive implementation also makes it possible to perform a fast run-time check for the integrity of the list. (This may not be obvious, but we will return to this problem in Chap.4.) On the other hand, non-intrusive lists are easy to implement using templates, and they form the basis of almost every existing container class library.

What do we mean by 'container' class? We use this term when a group of objects of one class is assigned to one object of another class. For example, Course includes Students, or Department contains Employees. Containment is a general logical characteristic, and does not imply any particular implementation. For example, a container can be implemented as a linked list (Fig.1a or Fig.1b) or as an array of pointers (Fig.1c).

In most class libraries, including the STL, the containers are implemented with arrays similar to Fig.1c, and a Link class is provided rather as an exception for some limited intrusive applications. How this came about is an interesting chicken-and-the-egg story: this type of container is easy to implement (and naturally fits) the concept of the C++ template; however, among other things, templates were designed to support this type of container. There are of course other reasons: multiple lists, heterogenous containers (see Fig.3), and class dependency. We will discuss these items later. Even though this is hidden from the user, Containers in Smalltalk are also implemented in this style, and the first major C++ library [7] admittedly followed Smalltalk tradition. GNU's early Lib++ was non-intrusive. Simula, from which some of the features of C++ evolved, did have intrusive containers - you had to derive from class Link.
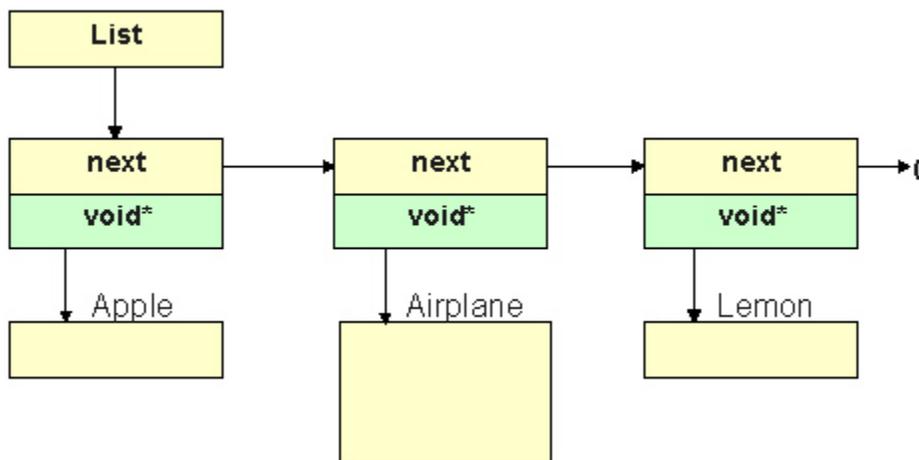

Fig.3: A container-based list may include objects of different types. However, this is a poor design. Use a list of common base objects in order to take advantage of the safe typing in C++.

Another interesting fact is that many class libraries are presented as if their containers provide all the data structures you may ever need. This of course is not true. All they give you is a few basic containers. Even the frequently encountered Aggregate (Fig.4) is not usually included, because the Child object must keep a reference to the Parent, and therefore the organization is intrusive at its very heart. Graphs, trees, many-to-many relations, and other data structures require two-way references. Even if non-intrusive containers are used, the data structure as a whole cannot be then non-intrusive.[12]
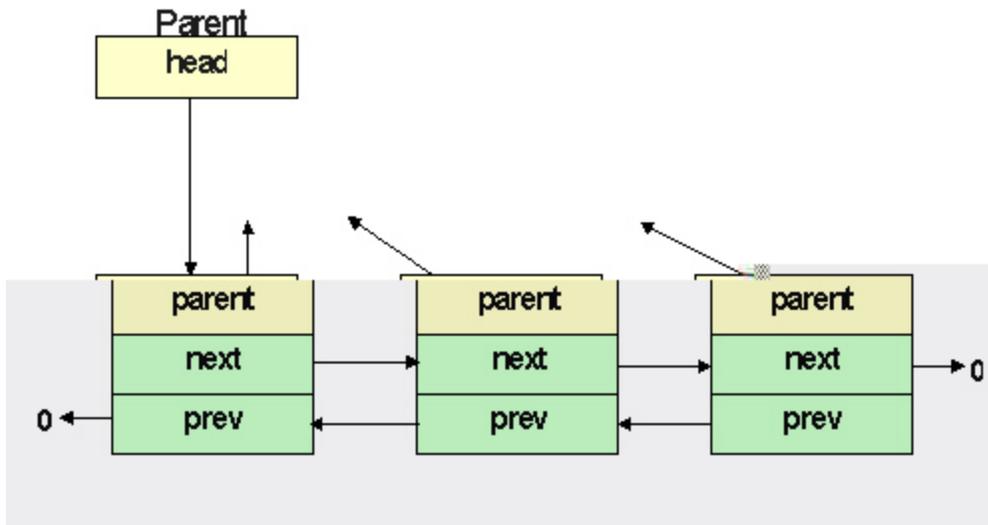
Fig.4: The intrusive implementation of Aggregate (one-to-many organization) with a doubly linked list of Children which point back to their Parent. This is one of the most frequent data organizations, and it occurs in any field of application.

Also, note the difference between the terms 'linked list' and 'list'. List is an abstract data structure in which the objects can be accessed in a sequential manner. The objects are ordered, and the list has its 'head' and 'tail'. A linked list is one special implementation of a list. Actually, all three containers shown in Fig.1 (a,b,c) are implementations of lists. For more on different implementations of list see [16].

A list is a container, but a container is not necessarily a list. The order of objects in a general container may not be defined, or you may be able to access its elements using an index or a key (an associative container). It is important that, historically, the C++ community developed a bias to represent all data structures using non-intrusive containers, such as those shown in Fig.1b and Fig.1c.

The purpose of this article is to correct this bias, and to show that in some situations the alternative---intrusive containers and data structures---are more elegant and more efficient than non-intrusive containers.

The article has two parts. The first half (sections 2,3, and 4) compares the two types of data structures (intrusive and non-intrusive) from different points of view: How does one's choice affect software architecture, program performance, and code quality? The second half (from section 5 on) discusses the implementation of intrusive data structures using C++ templates. This is equivalent to building a class library of intrusive data structures.

Chapter 2: Built-in types and polymorphic containers

There are two situations where intrusive containers cannot be used (at least not in their simple form), and where the non-intrusive form is generally preferred:

If you want the container to store objects of built-in types such as int or float.

When the container contains objects of two or more different types; see Fig.3 (polymorphic container).

Non-intrusive containers (see Fig.1b and Fig.1c) do not have these problems.

In case 1 the reason is obvious: you cannot add pointers to the built-in type of objects. However, this is an academic question, because you can always implement an intrusive container for objects of class IntWrap:

```
class IntWrap { int i;}
```

In case 2, you cannot use embedded pointers, because upon arriving at the next object, you would know neither its type nor the next pointer. You cannot have a polymorphic container without all its objects having something in common. There needs to be something for the polymorphic mechanism to work on, such as a Link or Object base. In that sense every polymorphic container is homogenous. For example, Smalltalk has only homogenous containers of Objects.

Also, in case 1 the array-based container (Fig.1c) can be modified to store a full copy of the objects, not just pointer references. For small-sized objects (whether built-in or not) this results in substantial memory and run-time savings:

```
template <class T> class List {
  T *array;
  int count, size;
  ...
};
List myList;
```

instead of:

```
template <class T> class List {
  T **array;
  int count, size;
  ...
};
class Student { ... };
List<Student> cList;
```

Note that it is not generally recognized (both by individual programmers or by the available publications) that every container must be internally implemented by an intrusive homogenous container. Non-intrusive containers are implemented as intrusive containers of links.

In situations such as that shown in Fig.3 (the heterogeneous container) the presence of the void* pointers adds a potential hazard. As you see in the figure, it is possible to add an airplane to a container which, perhaps, was originally meant just to store various fruits. In such situations, it is better to use the homogeneous container for the common base class:

```
class Fruit { ... };
class Apple : public Fruit { ... };
class Orange : public Fruit { ... };
List<Fruit> myList;
```

Note that the same approach now works with the intrusive linked list:

```
template <class T> class List {
   T *listHead;
public:
   void add(T *t){t->next=listHead; listHead=t;}
};

class Fruit {
friend class List<Fruit>;
   Fruit *next;
   ...
};
class Apple : public Fruit { ...};
class Orange : public Fruit { ... };

int main(){
   List<Fruit> myList;
   Apple* a=new Apple; myList.add(a);
   Orange* o=new Orange; myList.add(o);
   ...
}
```

Chapter 3: Performance

When comparing list implementations from Figs.1a, 1b, and 1c, the intrusive list (Fig.1a) is the most memory efficient. It needs only 4 bytes (pointer 'next') for each Element on the list. Fig.1b needs 8 bytes (pointers 'next' and 'obj'). It may appear that Fig.1c requires the same space as Fig.1a (one pointer, 'obj', for each Element on the list), but when growing a list such as this dynamically, the array is reallocated when it runs over its current size, and is often bigger than is actually needed.

Whether the list uses one or two pointers is usually insignificant even in applications where memory space is at a premium, because the objects have other data members, and the addition of one pointer represents only a relatively small increase in object size.

Traversing non-intrusive lists requires an additional pointer jump, which again is insignificant compared to other processing the application would do while traversing a list. However, in spite of all this theory, I encountered an interesting situation when optimizing the performance of a large telephone switch. We replaced non-intrusive data structures (such as in Fig.1b) by intrusive ones, and the speed almost doubled. Why?

In applications which frequently create and destroy objects, the main overhead comes from the allocation, initialization, and destruction of objects, not from adding and deleting items from the

list. Non-intrusive lists include twice as many objects (additional Link in Fig.1b), or require periodic reallocation and copying of the array (Fig.1c).

If you need non-intrusive lists, and still want to get good performance, the best strategy is to maintain a list of free Links, instead of creating and destroying them on demand. This strategy is also recommended by Stroustrup (see [14] Chap.5.5.6).

Another reason why intrusive data structures are sometimes more efficient is that random insert or remove operations from a doubly linked list are almost instant, but a linear search is necessary for the non-intrusive list. Here are some examples:

Example 1: List based on a doubly-linked intrusive ring.
(We will use a ring instead of NULL ending list. As you will see later, rings can be protected against an accidental corruption of the list.)

```
template<class T> class List {
  T *tail;
public:
  void insert(T *a,T *b){ // insert a before b
        a->prev=b->prev; b->prev=a;
        b->prev->next=a; a->next=b;
  }
  void remove(T *a){
        a->prev->next=a->next;
        a->next->prev=a->prev;
  }
  ...
};
class Employee {
friend class List<Employee>;
  Employee *next;
  Employee *prev;
  ...
};
```

Example 2: Non-intrusive list, using array of pointers.

```
template<class T> class List {
  T **obj;
  int size,count;
public:
  void increaseSize();
  void insert(T *a,T *b){ // insert a before b
        for(int i=0; i<count; i++) if(obj[i]==b)break;
        if(count+1>size)increaseSize();
        for(int k=count; k>i; k--)obj[k]=obj[k-1];
        obj[k]=a;
        count++;
  }
  void remove(T*a){
        for(int i=0; i<count; i++)if(obj[i]==a)break;
        for(; i<count-1; i++)obj[i]=obj[i+1];
        count--;
  }
  ...
};
```

When saving a list of objects to disk (persistent data), you can flatten the list to an array and store it on disk in that form, avoiding the notion of pointers. This works fine for a simple, isolated list, but in real life situations, we often have classes that participate in two or more lists. When storing such designs, we need some object ID's or, what is essentially the same thing, references or pointers, and it is the storage and recovery of these pointers which makes data persistence such a difficult problem.

Intrusive data structures have advantages when saving data to disk (persistent data):

The most expensive operation when retrieving data from disk is the recalculation of pointers (sometimes called pointer swizzling). If the list from Fig.1a uses only half as many pointers as the list from Fig.1b, the processing of pointers is cut to one half.

After recovery of objects from the disk, the program must convert them to valid C++ objects, essentially to reset the hidden virtual function pointers managed by the compiler. This is usually done by calling a special constructor. Since Fig.1a uses only half as many objects as Fig.1b, therefore there are only half as many calls to this special constructor.

These conclusions about performance apply not only to lists (Fig.1) but also to other data structures, because most data structures are composed of lists and pointers. For example, the Aggregate (one-to-many relation) is usually implemented as a list of Children that each have a pointer to their common Parent (Fig.4), the GRAPH keeps a list of Edges adjacent to every Node, and the HASH table keeps, for every bucket, a list of Entries that hash to the same value.

Stroustrup [14] comes to the same conclusions:

The simple intrusive list is close to optimal in time, space, and data hiding, and it provides great flexibility of expression.

We cannot have an intrusive list of ints.

The allocation of Links for non-intrusive lists creates overhead.

A circular list enables simple implementation of both append() and insert().

However, Stroustrup does not mention the advantage of checking the NULL pointers, which we will discuss in Section 4. As disadvantages of the intrusive list, he mentions that placing one object on two lists requires additional work. Even though this is true for the implementation shown in [14], it does not apply to the template implementation with manager classes which we will describe Chap.10.

Chapter 4: Data Integrity

Several years ago, before designing a library [13], I made a private survey in which I asked software managers what would help them most. Many of them said that they would like to eliminate pointer errors caused by wrong deallocation of objects or by messed up data structures. Such errors can be quite nasty---they may crash a program a long time after commiting the error, causing a major debugging headache. For a true story, see [1], p.91.

The fully typed containers prevent you from including a wrong type in a container, but they do not protect you against deallocating an object stored in the container without remove it from the container. Look for lines marked with "!!!" in main() below. The program crashes because pointer 'obj' on one of the Links points to a deallocated object. With some compilers, the program may

not crash and will only print an useless name, or you may not see any error at all until the memory used for the deallocated Student is reused for some other object.

```cpp
#include <iostream.h>

template<class T> class List;
template<class T> class Link {
friend class List<T>;
  T *obj;
  Link *next;
  // ...
};

template<class T> class List {
typedef void (T::*funType)(void);
  Link<T> *listHead;
public:
  List(){listHead=NULL;}
  void addHead(T *t){Link<T>* p=new Link<T>;
          p->next=listHead; listHead=p; p->obj=t;}
  void iterate(funType f); // apply fun to the list
  // ...
};

template<class T> void List<T>::iterate(funType f){
  for(Link<T>* p=listHead; p; p=p->next){
          (p->obj->*f)();
  }
}

class Student {
  char*name;
public:
  Student(char *n){name=n;}
  void prt(){cout << name << "\n";}
};
class Orange { int i;};

int main(){
  List<Student> myList;
  Orange* o=new Orange;
  myList.addHead(o); // compiler detects error !!!
  // ...
  Student* s=new Student("J.Brown");
  myList.addHead(s);
  // ... delete s; // accidentally destroying object from the list !!!
  // ...
  // much later on, traversing the list
  myList.iterate(&Student::prt); // !!! crash
  return 0;
}
```

Both the non-intrusive and intrusive lists shown above would crash in this example, but the difference is that this type of error cannot be prevented with the non-intrusive list, but it can be prevented for the intrusive list. The remaining part of this Section will explain how to do this.

Consider a design of the non-intrusive list where any object can be included only once; such an organization is usually called a set (*). Functions that add objects to the list must traverse the entire list just to verify that the new object is not on the list yet. As you will see, this is not necessary for the intrusive list.

Footnote (*): Note the different definitions of sets. Gorlen at al. [7] define a set as a collection were duplicates are not allowed. In STL, Stepanov [5] defines a set as an associative container (essentially a hash table) where duplicates are not allowed. A linked list with an inefficient inserting operation fits Gorlen's definition but not Stepanov's. I am guessing that Stepanov saw this problem and restricted sets to associative containers, where the uniqueness test is fast and efficient.

Intrusive data structure can be protected against most pointer errors by adhering to the following simple rules:

Implement linked lists as rings, not as NULL ending lists.

Initialize all pointers to NULL, and set them again to NULL when the object is disconnected from the data structure.

An object cannot be destroyed before all its pointers are NULL. If any pointer is not NULL, it means that the object is still connected to some data structure.

An object cannot be added to a data structure if the pointers implementing this data structure are not NULL, which means the object is not free.

Note that I am not absorbing garbage collection into the services of the list. This is only an increased control over the list operations, and blocking operations which would mess up the list.

Personally, I find it amazing that by automatically checking just a few pointers, you can guarantee the list integrity, and prevent all pointer errors including errors caused by inadvertent deallocation of a valid member of the list. The only situation where this does not protect you is if you attempt to work with invalid objects.

For example

```
List<Student> myList;
Orange* o=new Orange;
myList.addHead((Student*)o);
```

or

```
List<Student> myList;
Student* s;
// ...
delete s;
myList.addHead(s);
```

However, these are gross errors that can be detected by other means: The first error can be detected by automatic type detection, and better compilers or software tools such as Purify detect the access outside of valid memory in the second situation.

Obviously, I am relying here on the fact that, even though one object can be simultaneously on several intrusive lists, it can be at most just once on any particular list.

This checking can have three different levels of thoroughness:

The application code can check whether all intrusive pointers are NULL before any call to delete().

The same checking can be performed within the destructor, and when an error is detected, the destructor prints an error message.

When the destructor detects an error, it throws an exception and stops the destruction process.

As shown in the following example, strategy (2) does not prevent a crash, but it immediately gives you all the clues that indicate what went wrong, thus eliminating the tedious part of the debugging.

```c
#include <stdio.h>
class A {
  A *next;
  A *prev;
  int objNo;
public:
  A(int i){next=prev=NULL; objNo=i;}
  void append(A *n){
    if(!(n->next)){n->next=n->prev=n;}
    if(next)printf("append error: %d already there\n",objNo);
    else {
      next=n->next; n->next=this;
          prev=n; next->prev=this;
    }
  }
  void remove(){
    if(next){
              if(next!=prev){prev->next=next; next->prev=prev;}
              next=prev=NULL;
        } // does nothing when already disconnected
  }
  ~A(){if(next){
        printf("destruction error: %d still in use\n",objNo);}
  }
};
int main(){
  A a1(1);
  A* a2=new A(2); a2->append(&a1);
  A* a3=new A(3); a3->append(a2);
  A* a4=new A(4); a4->append(a3);
  a4->append(a2); // error message, adding a4 twice
  // ...
  delete a3; // error message, still in use
```

```
    a2->remove();
    delete a2; // no problem, correct
    // ...
    return 0;
    // error message when destroying a1
}
```

This program generates two destructor messages---one when destroying a3, the other one when destroying the automatically generated a1 on exiting function main(). Object a4 is never destroyed. It lives until the end of the run and then simply fades away.

Note that we do not have to check for both 'next' and 'prev' to be NULL. Since only functions of class A can set these pointers, either both pointers are NULL or both are not.

This program also demonstrates how treacherous pointer errors can be. Under some compilers, this program would compile and run without a crash. If you don't check pointers 'next' and 'prev', you'll think everything is correct, and you may expand the code until you reach the point when the compiler decides to recycle the memory for a2, which is still plugged in the list. The program will crash, and by that time it may not be easy to find that you forgot to call a2->remove() before destroying a2.

The following code fragment illustrates how to actually stop the destruction process by throwing an exception in the destructor. Be warned: this works only for objects allocated from the heap, for example, for a2,a3, and a4 from the previous example, but not for a1 which was automatically allocated.

```
    #include <stdio.h>
    class A {
A *next;
A *prev;
    ...
    public:
...
~A(){if(next)throw "A_still_in_use";}
    };
    int main(){
try {
  A* a1=new A(1);
  A* a2=new A(2); a2->append(a1);
  A* a1=new A(3); a3->append(a1);
  delete a2;      // throws exception
  delete a3;      // never gets here
      }
    catch(const char* p) { printf("error:%s\n",p);}
    return 0;
}
```

For automatically allocated objects, the program does not do what you may intuitively expect. For example, if you modify the last program like this

```
...
try {
   A a1(1);
   A* a2=new A(2); a2->append(&a1);
   A a3(3);        a3.append(a2);
   delete a2;
   delete &a3
}
...
```

and you compile using VC++, you do not get the expected message. Instead, the program terminates with message "abnormal program termination." There is a special rule that, in case of an exception, the program first destroys all objects that were automatically allocated within the last try{ ... } block; see [14], p.603 (Sect.r.15.3). When destroying a2, the program first destroys a1 and a3 before it starts to execute the 'catch' part, which leads to more throws, and the program exits without actually printing any message.

Comment: Under most circumstances, putting an automatic object on a list is dangerous, and should be avoided. However there are situations where a temporary list is needed within the scope of a single function.

Note that checking for pointers being NULL works only for intrusive data structures. In non-intrusive data structures, objects have no relational pointers.

One way to accomplish similar checking for non-intrusive data is to keep a reference counter on each object, as is done in Smalltalk. This makes the link non-intrusive, but the reference counter intrusive. The advantage would be that each object would have only one reference counter, regardless of the number of data structures in which it participates, and the counter would prevent us from destroying an object which is still in use (only objects having counter==0 may be destroyed). The disadvantage would be that this arrangement would not protect us from an accidental insertion of an object into the same list twice. The reference counter provides some checking, but not as much as described above for the intrusive pointers.

Here is an interesting twist: intrusive data structures permit better integrity checking, but at the same time, integrity checking is more important for intrusive data than it is for non-intrusive containers.

The integrity checking described above has been consistently used throughout the library in [13], and it is the main reason for the 2-3 times faster debugging reported by the users of that library.

While compiling, the protection is similar to the non-intrusive library. For example, the compiler catches errors such as adding an object to the wrong type of list. However, when you start running the program, the library applies the second level of checking, and pinpoints right away additional errors in your data structure handling. Debugging is reduced to a minimum, and once the program runs, it is quite safe to use.

Chapter 5: Inside or Outside?

Careful inspection of Figs. 1a and 1b reveals that you can derive Fig.1a from Fig.1b by merging the Link into the Element object. How can we implement this in code? There are two ways in which two objects can merge---either through inheritance or by making one of the objects a member of the other. Instead of a long abstract discussion, I will show this in code. We can have the following configurations:

```
template<class T> class Link<T>;
class Employee : public Link<Employee> {
  ...
};


template<class T> class Link<T>;
class Employee {
  Link<Employee>link;
  ...
};


class Employee;
template<class T> class Link<T> : T {
  ...
};


class Employee;
template<class T> class Link<T> {
  T t;
  ...
};


class Employee;
template<class T> class Link<T> {
  T& t;
  ...
};
```

Cases (3), (4), and (5) are not very practical, because the application program would not allocate and work with objects like Employee, but instead would use Link<Employee>. This would tremendously complicate the use of the Employee class methods, in particular constructors. The constructs are fine, but applying them as basic building blocks for Lists doesn't seem like a good idea.

Internally, the C++ compiler handles cases (1) and (2) in an almost identical way---I am relying here on a personal assurance from Stroustrup. The only difference is that the handling of functions is more graceful in case (1); also, inheritance lets us move freely between the two parts of the composite object.
For example, inside class Link<T>, we can do this:

```
template<class T> class Link<T>{
        ...
        void f1(T *t){
          Link<T>* lnk=(Link<T>*)t; // move to the inside object
          ...
        }
        void f2(Link<T> *lnk){
```

```
            T* t=(T*)lnk; // move to the outside object
            ...
         }
};
Employee *ep;
ep=new Employee;
Link<T>
```

Chapter 6: Templates, manager classes, general data structures

To explain how to implement a generic library of intrusive data structures, we have to make a big logical jump now. Everything what I have said so far will be useful, but discussing simple containers and lists cannot show the complexity of interaction which we want to handle. I will introduce three new concepts at once:

Advanced templates for intrusive data structures which can handle multiple lists gracefully.

Traditionally, one class holds the list, and manages its operation. I will separate this into two classes: One for the parent (holder) of the list, the second for managing it . This is a new concept specific to intrusive data structures, and this article may be the first time the reader has heard about them.

More complex data structures, such as graphs.

This section will probably require more effort to read, but it is the crux of the article. If you understand this section, you will be able to build your own library of intrusive data structures.

So far, we have been discussing only one aspect of the classes that form the data structures: where to store the relational pointers or arrays. Naturally, the next step is to look at the functions (methods) that manage the data.

When working with non-intrusive containers, this is a simple matter. Only the container class and its iterator are visible to the user; therefore the functions that control the data structure are naturally assigned to this class. The interface can differ, depending on the implementation, but generally follows the same style:

```
class Employee {
                // ...
};

class Department {
public:
   List<Employee> mylist; // non-intrusive
   // ...
};

void main(void) {
   Department* dp=new Department;
   Employee*   ep=new Employee;
   dp->mylist.insert(ep);
   // ...
   // traverse all employees in dp
```

```
    ListIterator<Employee> it(dp->mylist);
    while(ep= it++){
// ... ep is the next Employee
    }
  }
```

When working with intrusive data structures, the situation is different. Typically, there are two or more classes that form the data structure, and they interact with each other. We have a situation which Booch in [9] calls a 'mechanism'. Let's look at the design of the intrusive Graph, where Nodes are connected by Edges. One possible approach would be to assign functions to the class where they can be easily (and most efficiently) implemented, which means some functions are assigned to class Node and some to class Edge:

```
class Node {
        Edge *adjacent;
public:
        Node(){adjacent=NULL;}
        void addEdge(Edge *e,Node *t){
                e->set(adjacent,this,t); adjacent=e;}
        // ...
};
class Edge {
        Edge *next;
        Node *source, *target;
public:
        Edge *nextEdge(void){return next;}
        void remove(void); //
        void set(Edge *n,Node *s,Node *t){
                next=n; source=s; target=t;}
        // ...
};
```

When working with this graph, there is no class that represents the data structure. The graph is an implied behavior of Node and Edge, and is not clearly visible when reading the program. This makes the program more difficult to maintain and debug, especially for a person not familiar with the code. Also, any time you change either Node or Edge, both classes must be recompiled. The graph ties them together, and makes them dependent on each other.

```
Node* n1=new Node;
Node* n2=new Node;
Edge* e=new Edge;
n1->addEdge(e,n2); // add edge from n1 to n2
e=e->nextEdge();   // move to the next edge
```

For complex data structures and frameworks having many aggregates, graphs, and trees, this is a serious problem. Instead of spreading the functionality among the participating classes, it is better to represent each data structure by a special manager class, and to assign all the functions that control the data structure to this class. As you will see, the resulting interface will be similar to non-intrusive containers, such as those used in STL. In a way, this new class is a generalization of the container class: instead of containing objects of a single class, it contains several interacting types.

```
graph<Node,Edge> myGraph; // intrusive
Node* n1=new Node;
Node* n2=new Node;
Edge* e=new Edge;
myGraph.addEdge(e,n1,n2); // add edge from n1 to n2
e=myGraph.nextEdge(e);   // next edge on the same node
```

This is not a new technique; the iterators for non-intrusive containers are usually implemented in this style. Note that when using a manager class, classes like Node and Edge carry the required pointers or arrays, and the manager class (class Graph) represents the data structure. Then we have one or several iterators, coded in the same style as the manager class. In [1], the manager class was called a 'pattern class', because it can represent not only a data structure, but also some design patterns.

For the graph design to be efficient, it is important to avoid unnecessary function calls between Graph, Node, and Edge. If Node and Edge are friends of class Graph, Graph can access pointers and other data that form the graph directly. The following code is still without templates, but we will get there soon:

```
class Graph;
class Node {
friend class Graph;
        Edge *adjacent;
public:

        Node(){adjacent=NULL;}
};
class Edge {
friend class Graph;
        Edge *next;
        Node *source, *target;
public:
        Edge(){next=NULL; source=target=NULL;}
};
class Graph{
public:
        Edge *nextEdge(Edge *e){return e->next;}
        void addEdge(Edge *e,Node *s,Node *t){
                e->source=s; e->target=t;
                e->next=s->adjacent; s->adjacent=e;
        }
};
```

To make this design reusable in different applications, we will code it using templates, where N and E are types of application objects to be used as nodes and graphs. Parameter i helps handle multiple, parallel organizations, as will be shown later on in this Chapter. I have simplified the classes: there are no destructors, no functions to remove Edges or Nodes, but the internal list is circular, and function addEdge() checks if the Edge pointers are NULL:

```
template<class N,class E,int i=0> class Graph;
```

```
template<class N,class E,int i=0> class Node {
   typedef Node<N,E,i> nType;
   typedef Edge<N,E,i> eType;
   friend class Graph<N,E,i>;
   friend class GraphIterator<N,E,i>;
protected:
   eType *adjacent;
public:
   Node(){adjacent=NULL;}
};

template<class N,class E,int i=0> class Edge {
   typedef Node<N,E,i> nType;
   typedef Edge<N,E,i> eType;
   friend class Graph<N,E,i>;
   friend class GraphIterator<N,E,i>;
protected:
   eType *next;
   nType *target;
public:
   Edge(){next=NULL; target=NULL;}
};

template<class N,class E,int i=0> class Graph{
   typedef Node<N,E,i> nType;
   typedef Edge<N,E,i> eType;
public:
   E *nextEdge(eType *e){return (E*)(e->next);}
   void addEdge(eType *e,nType *s,nType *t){
      if(e->next)cout << "error: cannot add graph edge\n";
      else {
         e->target=t;
         if(s->adjacent==NULL)s->adjacent=e;
         e->next=s->adjacent; s->adjacent=e;
      }
   }
};

template<class N,class E,int i=0> class GraphIterator{
   typedef Node<N,E,i> nType;
   typedef Edge<N,E,i> eType;
protected:
   eType *beg, *nxt;
public:
   GraphIterator(const nType *n){beg=n->adjacent;
      if(beg)nxt=beg->next; else nxt=NULL;}
   E* const operator++(){eType* e=nxt;
      if(e==beg)nxt=beg=NULL; else nxt=e->next; return e;}
}
```

The following example shows how the user can implement a graph of airline flights which connect various towns. Note that if there is only one graph connecting Towns with Flights, there is no need to specify parameter i in all the templates, and the default of i=0 is used:

```
#include <iostream.h>
class Town;
class Flight;

class Town : public Node<Town,Flight> {
   // ...
};
class Flight : public Edge<Town,Flight> {
   int flightNo;
public:
   Flight(int f){flightNo=f;}
   void prt(void){cout << flightNo << "\n";}
};
void main(void){
   Flight *f;
   Graph<Town,Flight> network;
   Town* t1=new Town;
   Town* t2=new Town;
   Town* t3=new Town;

   f=new Flight(128);
   network.addEdge(f,t1,t2);
   f=new Flight(312);
   network.addEdge(f,t1,t3);
   // ...
   // traverse flights starting at t1
   GraphIterator<Town,Flight> it(t1);
   while(f= ++it){
      f->prt();
   }
}
```

Parameter i permits us to implement multiple copies of the same data structure. For example, we can keep two graphs that connect Towns with Flight: Graph 1 will keep the Flights departing from each Town; Graph 2 will keep the Flights arriving at each Town.

```
class Town : public Node<Town,Flight,1>,
         public Node<Town,Flight,2>{
   // ...
};
class Flight : public Edge<Town,Flight,1>,
         public Edge<Town,Flight,2> {
   int flightNo;
public:
   Flight(int f){flightNo=f;}
   void prt(void){cout << flightNo << "\n";}
};
void main(void){
   Flight *f;
   Graph<Town,Flight,1> from;
   Graph<Town,Flight,2> to;
   Town* t1=new Town;
   Town* t2=new Town;
```

```
f=new Flight(128);
from.addEdge(f,t1,t2);
to.addEdge(f,t2,t1);
// ...
// traverse flights originating at t1
GraphIterator<Town,Flight,1> fromIter(t1);
while(f= ++fromIter){ ... }
// traverse flights arriving at t2
GraphIterator<Town,Flight,2> toIter(t2);
while(f= ++toIter){ ... }
```

This is the style in which the new Intrusive Template Library from Code Farms is implemented, except that internal lists are not hard coded, but are inherited from the List<> manager class.

The big advantage of manager classes is that they permit the reuse of basic data structures through inheritance, when creating more complex data organizations, but this is beyond the scope of this article. For example, when constructing the intrusive Graph in the previous examples, we can use the previously designed manager class List, which controls classes Parent and Child (Parent has multiple Children). Instead of pointers Node::adjacent and Edge::next, we allow Node to inherit the Parent, Edge to inherit the Child, and Graph to inherit the List; see Fig.5.
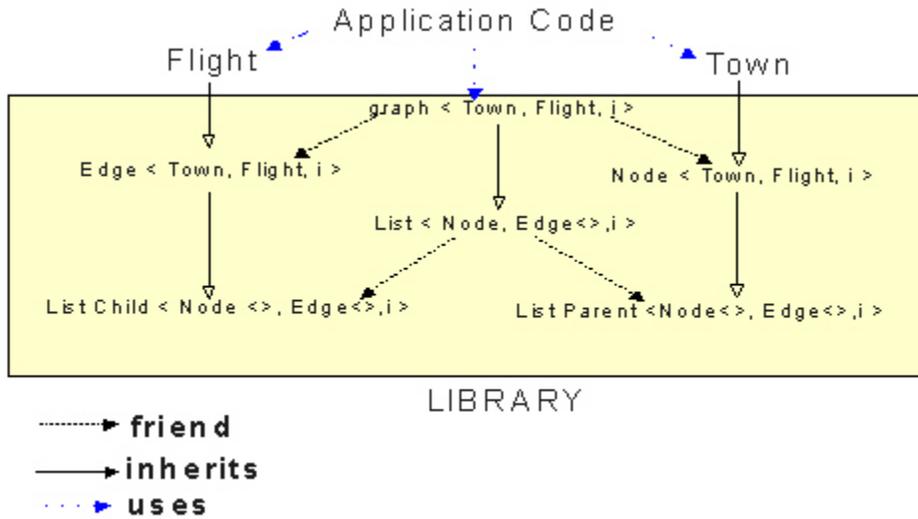

Fig.5: Class diagram for the Town-Flight example. This implementation style of intrusive Graph is used in the Intrusive Template Library by Code Farms; in the code presented in this article, Graph does not inherit from List.

Chapter 7: Interface standards, STL, and bi-directional access

When Alex Stepanov [5] and I discussed my intrusive library, in Fall 1995, his first reaction was: "Your interface is very similar to STL. We both have similar iterators and similar functions that manipulate the data structures. Could you hide your library under the STL interface?"

I have tried hard, and I can implement the same iterators and operators, but I cannot make the interface identical. The two concepts are far too different. When using manager classes, data and their relations are orthogonal. In STL, relations are interwoven with the classes that carry the data.

For example, when implementing an application where Company keeps a list of Employees, STL can be used in two styles, using one parameter in either case.

STYLE 1:

This is a plain bad design, because a Company is not a list of Employees. All Object-Oriented Design books (e.g. [9]) warn against this mistake.

```
class Employee { ... };
class Company : public List<Employee> {
   ...
};
```

STYLE 2:

This is a better style:

```
class Company { // version B
   List<Employee> cList;
   ...
};
```

The intrusive library needs two parameters, and possibly the third one, i:

```
class Company : public ListParent<Company,Employee> { ... };
class Employee : public ListChild<Company,Employee> { ... };
List<Company,Employee> eList;
```

or

```
class Company : public ListParent<Company,Employee,0> { ... };
class Employee : public ListChild<Company,Employee,0> { ... };
List<Company,Employee,0> eList;
```

This design is not identical with STYLE 1, because the list is represented by a new class, List<...>, which stands by itself. It would be nicer if we could hide the ListParent<> and ListChild<> completely like this:

```
class Company : private ListParent<Company,Employee> { ... };
```

but that would prevent easy access to the list, as presented below. If the user accepts classes ListParent<> and ListChild<> as an untouchable part of the library, and uses them only in the inheritance statement, there is no danger in inheriting these classes as 'public'.

Also when using the data structures, the logic (and syntax) of the function calls is different. With STL, we have

```
Company *c;
Employee *e1, *e2;
...
((List<Employee>*)c)->insert(e1,e2); // version A
c->cList->insert(e1,e2);          // version B
```

while with the intrusive data structures we have

```
Company *c;
Employee *e1, *e2;
...
eList.add(c,e1,e2);
```

Note the difference in the logic behind the syntax:
STL: Go to Company c, get its Employee list, and insert e2 before e1. Object c is primary, and the data organization (list) is secondary.
Intrusive: Under the data organization eList, insert e2 in Company c so that it is before e1. The data organization is primary, c is only one of the participants.

An important quality of all non-intrusive containers is that they are uni-directional. For example, if Company contains Departments, and Department contains Employees, the Department does not know its Company, and the Employee does not know its Department. If your application needs to know the parent of each object, you have to insert an additional pointer or pointers into Department and Employee, thus destroying the non-intrusiveness of your data structure.

In real life, bi-directional data organizations are very common, and more than 50% of projects involve at least some bi-directional data. Intrusive data structures naturally handle bi-directional access, and the design of general hierarchies, trees, graphs, or many-to-many relations are logical and simple. For example, the relation between Department and Employee is not represented as a Department containing Employees, plus a pointer from the Employee to its parent, but it is all one data structure called Aggregate (Fig.4). Aggregate can be controlled by functions which are similar to those for the Collection, except that there is also the parent() function which, for a given Employee, returns its parent Department:

```
AggregateParent* Aggregate::parent(AggregateChild*),
```

Aggregate is one of the most frequently used data structures in practical applications, but its concept is foreign to STL. The parent() function cannot exist in a non-intrusive library.

What is wrong with using a non-intrusive container and a pointer? The two form one data organization which is not handled as such, because the library cannot handle it. When treated as

one unit, its interface is simpler (fewer parameters required for some functions), and its implementation more efficient.

Existing STL containers and the new intrusive data structures can co-exist, and it would make sense to place them both into the same library, rather than to maintain two libraries---one intrusive and one non-intrusive.

Note that the intrusive data structures are more general, because they can easily emulate the non-intrusive form, but the opposite is not true. For example, if we need a list where one object can appear several times (a bag), we make an intrusive list of objects Link, with a pointer to the actual object. To be consistent, we consider even a single pointer to be a data organization, called SingleLink:

```
class Company : public ListParent<Company,Link,0> { ... };
class Link : public ListChild<Company,Link,0>,
        public SingleLinkSource<Link,Employee,0> { ... };
class Employee : public SingleLinkTarget<Link,Employee,0> { ... };
List<Company,Link,0) eList;
SingleLink<Link,Employee,0> eLink;

Company *cp; Employee *ep; Link *lnk;
...
// adding Employee e to the list
lnk=new Link; eLink.add(lnk,e); eList.add(c,lnk);
```

Chapter 8: Class Dependency

The implementation of the intrusive data structures introduced above leads to some class dependencies. The application classes like Company or Employee must be derived from certain library classes to be included in the data structures. Each manager class is a friend of several other classes. Won't all this create an environment where changes in one header file would trigger recompilation of the entire system? Isn't a program like that difficult to maintain and debug?

When you introduce a new data structure, you have to modify the inheritance of all the participating classes, and recompile them. This relatively small work can be avoided by using a simple code generator, which is described in the next chapter. The classes that represent the data organization, such as List<Company,Employee,1> or Graph<Town,Flight,1> will recompile only when the header file of one of the participating classes is changed, which is just a local dependency.

When working with these single data organizations, one important fact does not come out clearly enough: the manager classes actually relax dependencies among application classes, and generally improve software architecture. Let us expand the Town-Flight example to four classes: Town, Flight, Pilot, and Root. Root class is the root of all the data. We want to store a list of Towns, a list of Pilots, a graph of Flights connecting Towns, and a double link between Pilot and Flight. DoubleLink is a data organization where the two classes point to each other (one-to-one relation). Fig.6 shows the mutual decoupling of application classes. They don't invoke each other's methods, at least not those methods needed for the manipulation of the data structures. This is precisely the reason why I claim that manager classes create orthogonality between data

and their relations. In Fig.6, the top row shows the manager classes provided by the library; they represent the data relations. The middle row shows the application classes, with their relations hidden. The bottom row shows the classes needed for the implementation of the relations, and those provided by the library.
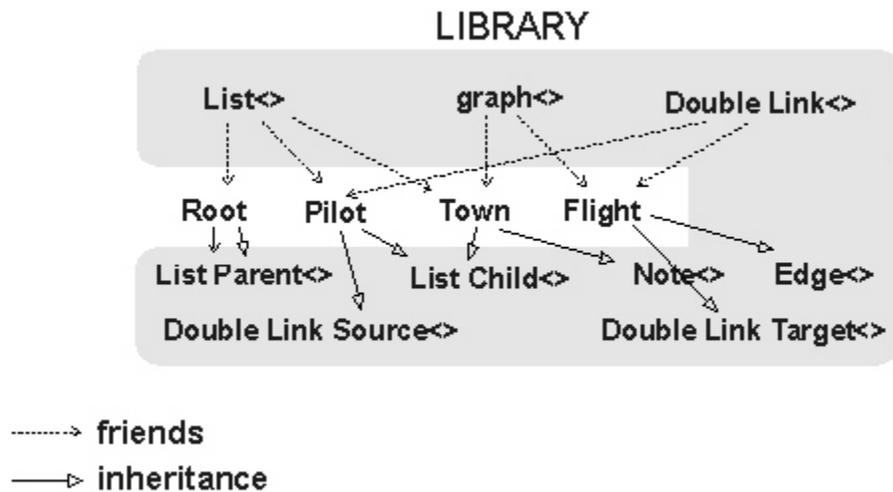

Fig.6: Dependency among classes in a more complex example. Note the three layer architecture, with application classes in the middle layer. Application classes are mutually independent; changes in one class will not trigger recompilation of other classes. [Internal relations among the library classes are not shown.]

For more reading on manager classes and class dependency, see [12] and [1].

Chapter 9: Existing Libraries

As I mentioned before, practically all C++ class libraries today are based on non-intrusive containers. This has two historical reasons:

The first large, widely-distributed, public-domain C++ library [7] was modeled after Smalltalk.

Without templates, it was impossible to implement generic containers in the intrusive form.

The existing libraries mostly follow this tradition [5, 6], and many programmers falsely believe that non-intrusive containers are the only correct way of designing data structures in C++, in spite of Stroustrup presenting both the intrusive and non-intrusive lists ([14], Chap.8.3).

Note that neither tools.h++ [6] nor STL [5] include more advanced data structures such as aggregates, graphs, or many-to-many relations, because such relations are inherently intrusive.

The only commercial library which provides intrusive data structures was developed by Code Farms [8]. This library was designed before templates became a part of the C++ language, and it uses the code generator described in the next paragraph, part (B). The library has been on the

market since 1991, and has been used successfully in numerous industrial applications, including communication, business applications, and entire CAD and CASE systems.

An important feature provided by some class libraries, and this applies to both non-intrusive and intrusive libraries, is data persistency --- the ability to store entire data structures to disk, and restore them back to memory. STL [5] does not provide persistency. tools.h++ [6] and MFC from Microsoft require the user to code a serialization function for each class. Code Farms library [13] makes data persistent automatically. Whatever method is used, when data is restored back to memory, objects move to different memory locations, and all pointers have to be recalculated. For this reason, implementation of persistency is closely related to what we do with pointers inside the data structures.

The code examples in this article completely ignore persistency. For example, if we used smart pointers, certain methods of persistency could be transparently added to our classes. For more details on how to implement persistent data, see [1], Chap.8.

For details on the concepts of STL (Standard Template Library), see [10] and [11].

Chapter 10: Code Generators

(A) SIMPLE CODING TOOL

Let us assume that our library provides the intrusive Aggregate based on the classes AggregateParent and AggregateChild, and we want to implement the organization shown in Fig.6. We assume that each Employee is a member of exactly one Department, and works exactly on one project. Using the style which we have introduced above, we can write:

```
class Company;
class Project;
class Department;
class Employee;

class Company : public AggregateParent<Company,Project,0>,
                        public AggregateParent<Company,Department,0> {
// ...
};
class Project : public AggregateChild<Company,Project,0>,
                        public AggregateParent<Project,Employee,0> {
// ...
};
class Department : public AggregateChild<Company,Department,0>,
                          public AggregateParent<Department,Employee,0> {
// ...
};
class Employee : public AggregateChild<Project,Employee,0>,
                        public AggregateChild<Department,Employee,0> {
// ...
};

// .....

// --------------------------------------------------------
```

```
// See Fig.7 and its direct mapping to the following 4 lines
// ----------------------------------------------------------
Aggregate<Company,Project,0>     projects;
Aggregate<Company,Department,0>  departments;
Aggregate<Project,Employee,0>    byProject;
Aggregate<Department,Employee,0> byDepartment;
// ----------------------------------------------------------

Project*    p=new Project;
Department* d=new Department;
Employee*   e=new Employee;

byProject.add(p,e);
byDepartment.add(d,e);
```
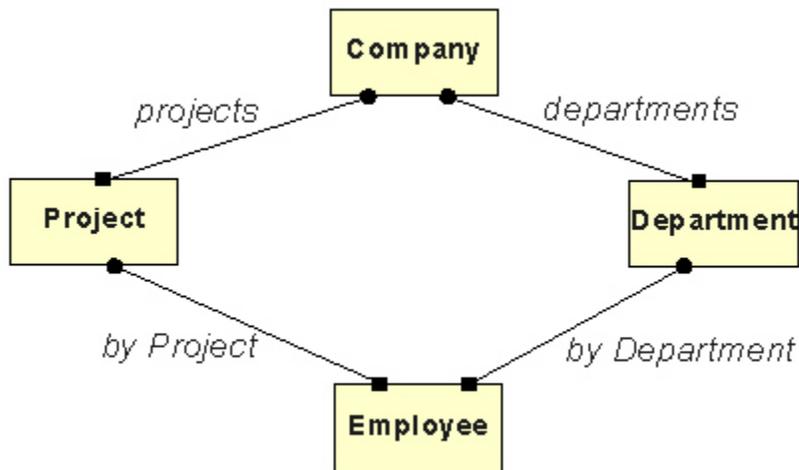

Fig.7: Example with 4 classes and 4 aggregates.

What complicates the use of these data structures are the complex inheritance statements. Fortunately, all the information about which classes must be inherited is already contained in the Aggregate<..> statements, and can be derived automatically.

In addition to the basic, pure template form, the Intrusive Template Library from Code Farms also provides a simple template manager, which reads data structure declarations (they must be marked with keyword 'pattern'), and generates macros that supply inheritance statements. This template manager only simplifies the use of templates; it does not change or generate the actual code. Programming with the template manager is much simpler, reducing the previous example to this:

```
class Company;
class Project;
class Department;
class Employee;

class Company : Pattern(company) {
```

```
// ...
};
class Project : Pattern(Project) {
// ...
};
class Department : Pattern(Department) {
// ...
};
class Employee : Pattern(Employee) {
// ...
};

// .....

// -------------------------------------------------
// Next 4 lines are the input for Template Generator
// Again, they map directly to Fig.7
// -------------------------------------------------
pattern Aggregate<Company,Project>     projects;
pattern Aggregate<Company,Department>  departments;
pattern Aggregate<Project,Employee>    byProject;
pattern Aggregate<Department,Employee> byDepartment;
// -------------------------------------------------

Project*    p=new Project;
Department* d=new Department;
Employee*   e=new Employee;

byProject.add(p,e);
byDepartment.add(d,e);
```

Every application class which is involved in data structures must have the Pattern(...) statement. All data structure declarations must start with the keyword 'pattern'.

(B) BY-PASSING TEMPLATES AND INHERITANCE

Once you decide to use a code generator, you can avoid templates altogether. This is the approach taken by the original Code Farms library [8], and we present it here mostly for completeness and historical reasons. It was developed before templates were available, and it still has advantages in some situations. The Code Farms' code generator [13] builds a meta model of all application classes, and it inserts the required pointers using generated macros, not inheritance. This is a dirty but efficient method, which results in faster code, but is not as elegant as templates. Its disadvantages are the difficulty of restricting the scope of data organizations (typically assumed to be global), and difficulties with internal reuse when building new data structure libraries. Its advantages are that the code generator knows the locations of all the pointers, and makes the data structures automatically persistent, without the user coding and supporting serialization functions for every class.

With this library, the example from Fig.7 takes on the following form. Note the ZZ_EXT_.. statements that effectively replace the Pattern() macro, and the ZZ_HYPER_.. statements that replace the 'pattern' statements from the last implementation:

```
#include "zzincl.h"
```

```
class Company {
ZZ_EXT_Company

 ...
};
class Project {
ZZ_EXT_Project

 ...
};
class Department {
ZZ_EXT_Department

 ...
};
class Employee {
ZZ_EXT_Employee

 ...
};


.....


// ------------------------------------------------
// Again, these 4 lines map directly to Fig.7
// ------------------------------------------------
ZZ_HYPER_AGGREGATE(projects,Company,Project);
ZZ_HYPER_AGGREGATE(departments,Company,Department);
ZZ_HYPER_AGGREGATE(byProject,Project,Employee);
ZZ_HYPER_AGGREGATE(byDepartment,Department,Employee);
// ------------------------------------------------

Project*   p=new Project;
Department* d=new Department;
Employee*   e=new Employee;

byProject.add(p,e);
byDepartment.add(d,e);

#include "zzfunc.c"
```

ZZ_EXT is the abbreviation for 'the extension of the object'. ZZ_HYPER (the hyper-class) is a historical term originally used for what we call the manager class now. There is no need for an integer parameter such as we needed for templates: the instance name provides the parametrization. For example, if you want to establish two parallel, independent aggregates between Company and Project, you declare:

```
 ZZ_HYPER_AGGREGATE(projects1,Company,Project);
 ZZ_HYPER_AGGREGATE(projects2,Company,Project);
```

Here, projects1.add(c,p) adds p to c under project1, while projects2.add(c,p) adds the same p to c under project2.

The code generator reads the code, detects all ZZ_.. statements, generates the manager classes and the pointers to be inserted under ZZ_EXT_.. statements, and deposits them into the file 'zzincl.h'. Then it generates all of the access functions, and deposits them into the file 'zzfunc.c'. The Prefix ZZ is used to minimize the chances of name collisions, and to speed up the code

generation. The user compiles the original code with files zzincl.h and zzfunc.c, and within the realm of his or her own code, can use their regular debugger:

FILE ZZINCL.H:

```
#define ZZ_EXT_Department \
  Company *ZZparent_departments; \
  Department *ZZsibling_departments; \
  Employee *ZZchild_byDepartment;

#define ZZ_HYPER_AGGREGATE(id,parent,child) \
class ZZ_##id { \
friend class parent; \
friend class child;  \
public: \
  void  add(parent *d,child *e); \
   ... \
} id;
  ...
```

FILE ZZFUNC.C

```
void ZZH_byDepartment::add(Department *d,Employee *e){
 e->ZZparent_byDepartment=d;
 e->ZZsibling_byDepartment=d->ZZchild_byDepartment;
 d->ZZchild_byDepartment=e;
}
 ...
```


This older method is more efficient, because it does not involve many layers of inheritance as when using templates, and it supports automatic persistence, which the templates do not (at least not so far). However, I like the flexibilty of templates and the ability to reuse existing data structures when building new, more sophisticated library classes.

Acknowledgements

Many thanks to Dr. Stroustrup for numerous valuable commments.

Conclusions

Intrusive data structures have generally been neglected in current C++ practices, and except for the Code Farms libraries, all class libraries provide mostly non-intrusive containers. Intrusive data structures are highly efficient, and are a better choice in many situations. When implemented using manager classes, intrusive data structures also improve software architecture. Complex inheritance statements associated with the template implementation of intrusive data structures can be automatically generated by a simple template manager (Section 10, A). This makes coding with intrusive data just as easy and neat as coding with non-intrusive containers.

References:

1. Soukup J.: Taming C++: Pattern Classes and Persistence for Large Projects, Addison-Wesley 1994, ISBN 0-201-52826-6

2. Ellis M., Carroll M.: Inheritance-based vs. template-based containers, C++ Report, March-April 1993, Vol.5/No.3, pp.17-20.

3. Crawford J.: Runtime parametrization revisited, C++ Report, Nov-Dec 1994, Vol.6/No.9, pp.30-33.

4. Carroll M.: Tradeoffs of runtime parametrization, C++ Report, Nov-Dec 1995, Vol.7/No.9, pp.20-27.

5. Stepanov A., Lee M.: The Standard Template Library, Hewlett-Packard, 1501 Page Mill Rd., Palo Alto, CA 94304, Aug.18/94

6. tools.h++, class library sold by Rogue Wave, 1325 NW 9-th Str., Cornvalis, OR 97330

7. Gorlen K.E., Orlow A.M., Plexico P.S.: Data Abstraction and Object-Oriented Programming in C++, John Wiley & Sons, 1990 (The book includes a diskette with the code of the NIH library.)

8. C++ Data Object Library, class library sold by Code Farms Inc., 7214 Jock Tr., Richmond, Ont., Canada, K0A 2Z0

9. Booch G.: Object Oriented Design with Applications, Benjamin/Cummings Publ. Co., 1991

10. Vilot M.: An introduction to the STL Library, C++ Report, Oct.1994, Vol.6/No.8, pp.22-29,35.

11. Stroustrup B.: Making a vector fit for a standard, C++ Report, Oct.1994, Vol.6/No.8, pp.30-34.

12. Soukup J.: Quality Patterns, The C++ Report, Vol.8, No.9, Oct.1996, pp.34-45. Also: Managing Groups of Cooperating Classes, same issue, pp.46-47.

13. C/C++ Data Object Library, Code Farms Inc., http://www.CodeFarms.com

14. Stroustrup B.: The C++ Programming Language, Second Edition, Addison-Wesley 1991, ISBN 0-201-53992-6

15. Pattern Templates Library, Code Farms Inc., http://www.CodeFarms.com

16. Koenig A., Stroustrup B.: Foundations for Native C++ Styles Software Practice and Experience, Vol.25, Special Issue S4, Dec.1995