How to teach data structure design: an incremental approach

Ernesto Cuadros-Vargas Roseli A. Francelin Romero* Markus Mock*[§]
[*]Instituto de Ciências Matemáticas e de Computação
Univ. de São Paulo at São Carlos-Brazil
[§]Dept. of Computer Science
University of Pittsburgh
ecuadros@spc.org.pe, rafrance@icmc.usp.br, mock@cs.pitt.edu

Abstract

While there are many ways to teach data structure design, it is important to distinguish alternative data structure designs in terms of maintainability, flexibility, scalability and the degree of data abstraction. This paper presents an incremental approach based on exposing students to different alternatives and showing their advantages and drawbacks. We believe that this novel incremental approach that is specifically based on the comparison of alternative designs is particularly instructive because of its attention to expressiveness, i.e., implementation methods and languages and implementation efficiency, i.e., good algorithm design.

1 Introduction

Programming techniques and methodologies have made significant advances since the appearance of the first high level programming languages Fortran and Cobol. For example, in the early days the use of the "GOTO" statement was common among programmers. First, because of the lack of higher level control structures and then also due to programmers' familiarity with it from assembly level programming. Despite many advances in programming methodology and programming languages, still a lot of code is written in ways that makes it hard to maintain and evolve. Moreover, many compute science textbooks that teach data structure and algorithm design present code examples that disregard these principles for the sake of simplicity, which may provide students with bad examples for their future software projects. It is not uncommon to find even computer science majors writing programs employing undesirable approaches, including overly restrictive algorithms and heavy use of global variables to name just two examples.

During the evolution of programming techniques and languages, many different design principles (e.g., structured programming and object oriented programming) have been developed. Some of them, have profoundly changed our notions of programming. Today, we teach students data structure design to provide them with fundamental building blocks for the construction of software. Moreover, we teach them efficient algorithms for the solution of common problems. In this paper, we present an incremental approach for the teaching of data structures that focuses on their implementation according to different program methodologies. For each alternative,

[*]This work was partially supported by the FAPESP-Brazil under Grant No. 99/11835-7

we examined the advantage and disadvantages to teach students important design principles for their own projects.

During the evolution of programming languages, some important new principles, like those presented by Knuth [5], have arisen and changed our concepts about programming profoundly. Data structures are used to organize data in order to retrieve the information as fast as possible. Besides the idea represented by a data structure, the used algorithms for their management (and implementation) are also important. This paper focuses on the algorithms used to implement data structures. The different techniques we explain here are the result of applying different programming paradigms and programming language's resources. The ultimate goal of this approach is to make students select both efficient and flexible solutions that make software easier to maintain and evolve.

It is important to highlight that we will not show new programming techniques or methodologies. However, by contrasting different approaches according to their maintainability, reusability, scalability and performance we strive to impart on students the importance of a careful balance of simplicity in design (e.g., use of global variables) and important software engineering principles. The second goal of our paper is to present, from a didactical point of view, the importance of teaching all these techniques to programmers and comparing their advantages and drawbacks when applied to different problems. If we teach all these techniques, the students will also understand the consequences of using each one in real systems.

To illustrate different alternatives, we use a running example, a vector data structure. However, the principles we examine are applicable to any data structure. We treat the vector data structure as a black box, providing some basic functionality, such as accessor functions (e.g., *Insert()*) or methods to change the size (e.g., *Resize()*). The implementation details, which are hidden for the final user of this component, will be the focus of this paper. We will also show some code involving other data structures such as linked lists or binary trees in order to show more implementation's problems. We use ANSI C++ [7] because it is a general-purpose programming language, with a bias toward systems programming that supports efficient low-level computation, data abstraction, object oriented programming and generic programming and is also commonly used in the class room.

The paper is organized as follows. In section 2, we classify design approaches according to several criteria such as dynamicity, data protection, data abstraction, and data encapsulation. In section 3, we present different techniques to implement data structures. In section 4, we present a short discussion to show how the quality of used algorithms can directly affect the final product. In section 5, we discuss the importance of knowing compiler features in order to improve the final product. Finally, the summary and conclusions are presented in section 6.

2 General Overview

Existing data structure design techniques can be classified according to various criteria, such as dynamicity, data protection, encapsulation, data abstraction among others. Using dynamicity as a criterion, we can come up with the classification shown in Figure 1. Using dynamicity as a classification criterion clearly distinguished designs of the vector data structure that use static memory allocation, an inflexible design, an a flexible design that uses dynamic memory allocation.

Data protection is another classification criterion, which became prominent with the advent of object oriented programming. Figure 2 distinguishes design alternatives wrt. data protection, for instance, using global variables (Sections 3.1, 3.2), variables into namespaces (Section 3.7) and those class members which were declared as public (Section 3.3). We can also distinguish variables with partial protection like protected members (in classes). The last possibility is fully

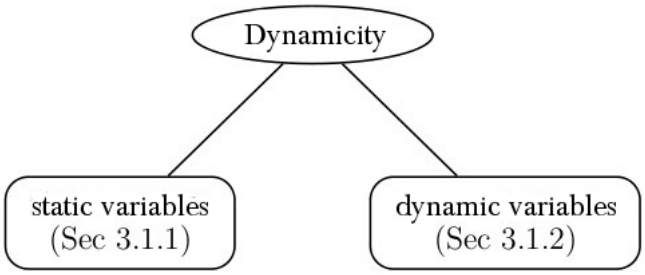

Figure 1: Classification according to Dynamicity

protected members which are better known as private members.

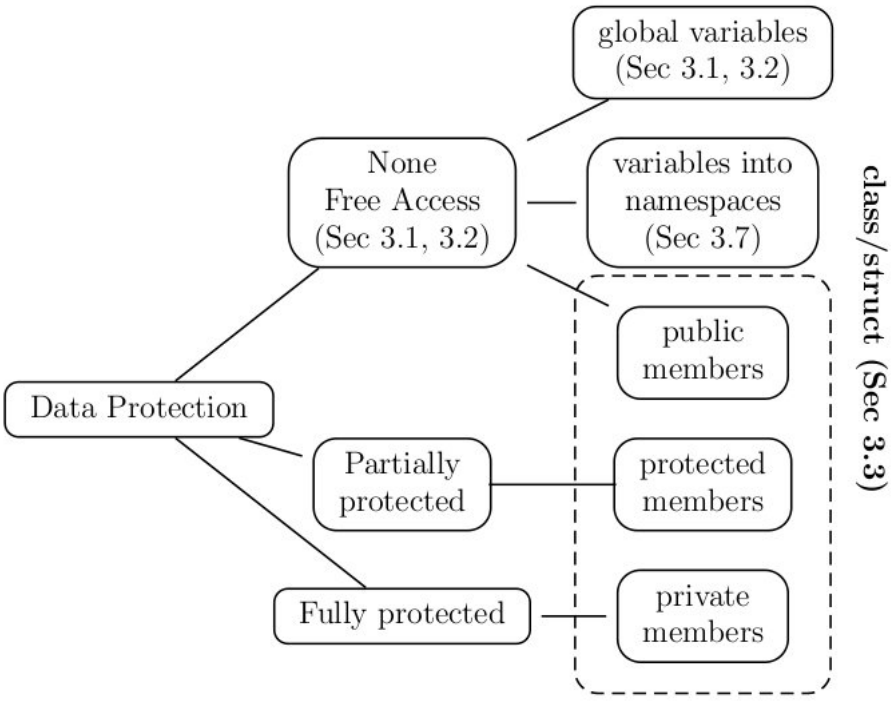

Figure 2: Classification according to data protection

Another important criterion for the classification of existing techniques is the **encapsulation**. In Figure 3 we present a possible classification.

Global variables and functions represent those cases where we data encapsulation is not used. The second case is represented by classes. A class can contain members, methods (Section 3.3) and even nested classes and structures (Section 3.6).

When a class becomes more complex, partial views of it are necessary (Section 3.9). Interfaces provide this technique to access a class partially.

From a higher point of view, we can see namespaces as the highest level of encapsulation (Section 3.7). A namespace can contain global variables (with public access), global functions, classes, structures and even nested namespaces.

We can also classify those techniques according to **data abstraction** (figure 4). This issue is specially important the teach abstract data types.

The first group is represented by inflexible techniques. In this groups data structures are

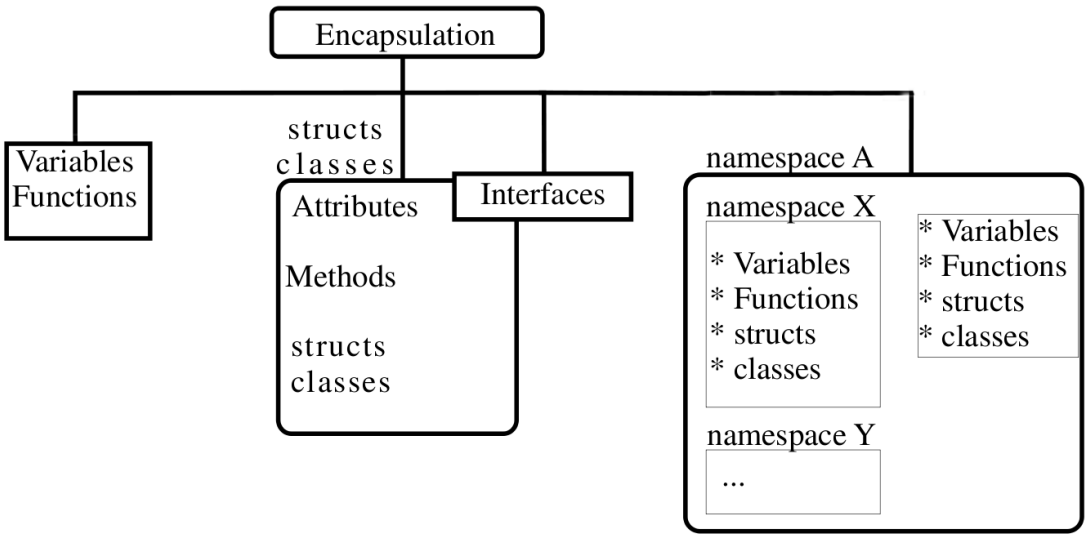

Figure 3: Classification according to encapsulation

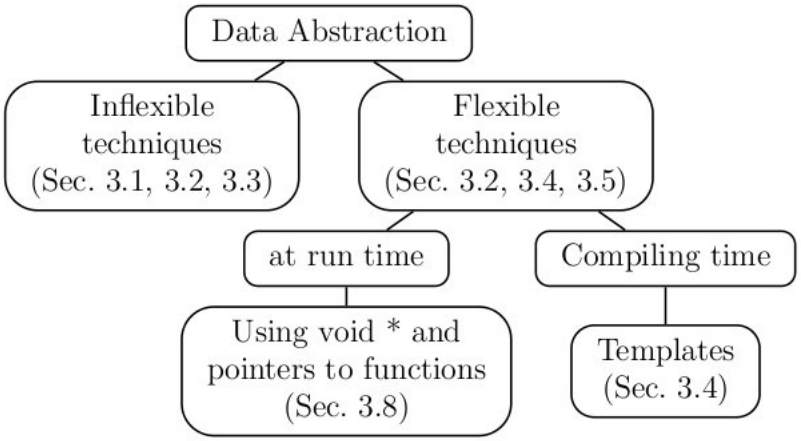

Figure 4: Classification according to data abstraction

designed for a specific data-type (Sections 3.1, 3.2, 3.3). We can also design data structures for abstract data types using templates (Sections 3.4, 3.5). Templates are useful to reduce code maintenance.

On the other hand, if the data type is only known at run time, we could not use templates, but we can use different techniques like the ones presented in section 3.8.

In the following sections we explain each one of these techniques in details.

3 Implementation alternatives

3.1 Fixed size vectors and variables

If the goal is to create a component which encapsulates a vector one of the most primitive solution could be to declare a global fixed size vector which is only shown here as the simplest solution, in terms of lines of code. The second alternative, which is probably still primitive, is

NOTE:
The sequence used to present the following sections does not necessarily represent a sequence in terms of performance. For example, section 3.1.2 discusses the use of dynamic vectors and is presented before templates (section 3.4) but it **does not imply** that all the templates are dynamics. In fact, we can use templates with static vectors. In this case the final performance correspond to the one presented in section 3.1.1 (Fixed size vectors) instead of the one claimed for templates.

to modify this first one and use dynamic vectors but still global variables. We present these two techniques in sections 3.1.1 and 3.1.2 respectively.

3.1.1 Fixed size vectors

In this section we will use a global fixed size vector (*gVect*), to store the inserted elements of type integer and, a global counter (*gnCount*), to control the number of used elements in the vector. These variables are declared according to following code.

```
int gVect[100]; // Buffer to save the elements
int gnCount;    // Counter to know the number of elements used
```

Figure 5: Global variables

As the vector's user needs some functions to interact with these internal variables, we will implement a first version for the *Insert*

```
void Insert(int elem)
{
   if( gnCount < 100 )          // we can only insert if there is space
       gVect[gnCount++] = elem; // Insert the element at the end
}
```

Figure 6: Insert function according to the first paradigm

The only probable advantage of this approach is that we do not need extra time to deal with memory management (alloc, malloc, free, new, delete).

Obviously, this technique has many drawbacks but, we highlight the following ones:

- The more serious drawback to solve in this first approach is, doubtlessly, its inflexibility. If the user/system needs less than 100 elements, there is no way to save the remaining space. On the other hand, if our user/system needs to insert more than 100 elements, this approach is not useful anymore.
- It is not possible to use this code for more than one vector in the same system. It is not flexible for more than one data type. However, to solve this last problem we could duplicate the same code and change the type. It is probably not a good choice for practical purposes.

3.1.2 Dynamic Vectors and global variables

As we saw in Subsection 3.1.1, more flexible structures are needed. We will try to solve this problem in this section but, it is necessary some previous knowledge about pointers and dynamically allocated memory (see [7] for a good reference).

In order to overcome its inflexibility, we will start with a pointer and we will increase the size of the vector gradually according to the requirements. In Figure 7 we can see the initial variables for this case.

```
int *gpVect = NULL;  // Dynamic buffer to save the elements int
gnCount = 0;         // Counter to know the number of used elements int
gnMax   = 0;         // Variable containing the size of the allocated
                     // memory
```

Figure 7: Global variables for dynamic vector

We use *gnCount* to control how many elements we are using considering that he have allocated memory for *gnMax* positions. The gap between *gnCount* and *gnMax* represents the space available for new possible elements. Another problem which is out of the scope of this paper is to determine the *delta* used to increase the vector when a overflow occurs. Just for practical purposes we will use $delta = 10$.

The code for *Insert* is now a little bit different (Figure 8).

```
void Insert(int elem)
{
  if( gnCount == gnMax )    // There is no space at this moment for elem
     Resize();              // Resize the vector before inserting elem
  gpVect[gnCount++] = elem; // Insert the element at the end of the sequence
}
```

Figure 8: Insert function using global variables but dynamic vector

The function resize could be defined according to Figure 9.

```
void Resize()
{
    const int delta = 10;             // Used to increase the vector size
    gpVect = realloc(gpVect, sizeof(int) * (gnMax + delta));
    // You can also use the following code:
    // int *pTemp, i;
    // pTemp = new int[gnMax + delta]; // Alloc a new vector
    // for(i = 0 ; i < gnMax ; i++)    // Transfer the elements
    //     pTemp[i] = gpVect[i];       // we can also use the function memcpy
    // delete [ ] gpVect;              // delete the old vector
    // gpVect = pTemp;                 // Update the pointer

    gnMax  += delta;                  // The Max has to be increased by delta
}
```

Figure 9: Resize function using global variables but dynamic vector

Among the advantages of this approach, we can note:

- This structure is more flexible than the one presented in the previous section.
- The allocated memory space will always be equal or a little bit greater than the user needs.

Among the drawbacks, we can highlight:

- This approach is still based on the use of global variables, so only one vector per program is possible.
- Compared with the previous solution, another drawback is the extra necessary time to deal with the memory management (even when the benefits clearly overcome this problem). This last drawback is outweighed by the flexibility introduced by pointers.

As we can see, this approach would not have been possible without using pointers (in *C++*).

From our point of view when the student learns to deal with pointers he has to understand how important they are specially for this kind of problems.

We will solve part of these drawbacks using Modular Programming in the following section.

3.2 Modular Programming

A possible solution to allow the coexistence of two or more vectors in the same system is to send the involved parameters (i.e. *gpVect*, *gnCount* and *gnMax*, in this case) for the function *Insert()*. The parameters must be sent by reference in order to allow some internal modification. According to this approach we can redefine the function *Insert* as follows. (Figure 10)

```
void Insert(int *& rpVect, int& rnCount, int& rnMax, int elem)
{
  if( rnCount == rnMax )     // Verify the overflow
      Resize(rpVect, rnMax); // Resize the vector before inserting elem
  rpVect[rnCount++] = elem;  // Insert the element at the end of the sequence
}
```

Figure 10: Preparing the Insert function for using multiple vectors in the same system

The function resize could be defined according to Figure 11.

```
void Resize(int *& rpVect, int& rnMax)
{
    const int delta = 10;           // Used to increase the vector size
    pgpVect = realloc(gpVect, sizeof(int) * (rnMax + delta));
    rnMax += delta;                  // The Max has to be increased by delta
}
```

Figure 11: Preparing the Resize function for using multiple vectors in the same system

Even considering that *Insert* and *Resize* work with the same data, their parameters are not necessarily the same. The function *Resize* only needs to deal with *rpVect* and *rnMax* to increase the vector size while *Insert* needs all of them. In general, different functions could need different parameters even when they belong to the same group and work with the same data. A useful solution to solve the general case where we need n parameters is to group all the possible parameters creating a C *struct* (See Figure 12).

```
struct Vector
{
    int*m_pVect,  // Pointer to the buffer
        m_nCount, // Control how many elements are actually used
        m_nMax,   // Control how many are allocated as maximum
        m_nDelta; // To control the growing
};
```

Figure 12: Struct vector encapsulating all the data for this data structure

Using a *struct* like *Vector* for each existing vector in our system, we only need to send the pointer of the struct to the function. This technique also allows us to access to *mp_Vect*, *m_Count* and *m_Max* through only one extra parameter (a single pointer to the *struct*). In Figure 13, we see the new code for functions *Insert* and *Resize*.

Among the advantages of this approach, we have:

- It is possible to have more than one vector per program, defining local variables that will be passed to the functions.
- Besides the flexibility, we do not need a variable number of extra parameters. It is important because it reduces the maintenance.

```
void Insert(Vector *pThis, int elem)
{
    if( pThis->m_nCount == pThis->m_nMax )    // Verify the overflow
        Resize(pThis);    // Resize the vector before inserting elem
    // Insert the element at the end of the sequence
    pThis->m_pVect[pThis->m_nCount++] = elem;
}

void Resize(Vector *pThis)
{
    pThis->m_pVect = realloc(gpVect, sizeof(int) *
                                (pThis->m_nMax + pThis->m_nDelta));
    // The Max has to be increased by delta
    pThis->m_nMax += pThis->m_nDelta;
}
```

Figure 13: Insert and Resize function with the data structure information encapsulated using a struct

Among the drawbacks, we can highlight:

- There is no data protection. All the members of a *C* struct are public by default.

3.3 Object Oriented Programming

If we pay attention to the code presented into the previous section, we will see that it is close enough to the Object Oriented Programming [7]. If we analyze carefully, we have *encapsulated* our data *m_pVect*, *m_Count*, *m_Max* and *m_Delta* into the *Vector* structure. When we send this *struct* as the first parameter for all the functions working around our data structure (vector in this case), we are in front of the same principle used by *C++* classes. In Figure 14 and 15, we can see the same code, but using Object Oriented Programming.

The function *Insert* from Figure 13 is now implemented as method of vector class (See Figure 15).

Among the advantages of this approach, it is necessary to note that:

- We now have data protection. Only the authorized functions can access our data.

Among the drawbacks, we can note:

- If we need, in the same system, two vectors using different types (i.e. int, double, etc.), we must code two different classes with the same code (except for the type declarations) in all functions.

```
// Class CVector definition
class CVector
{
    private:
    int *m_pVect,   // Pointer to the buffer
        m_nCount,   // Control how many elements are actually used
        m_nMax,     // Control how many are allocated as maximum
        m_nDelta;   // To control the growing
        void Init(int delta); // Init our private variables, etc
        void Resize();        // Resize the vector when occurs an overflow

    public:
        CVector(int delta = 10);  // Constructor
        void Insert(int elem);    // Insert a new element

        // More methods go here
};
```

Figure 14: Designing a vector using Object Oriented Programming

```
void CVector::Insert(int elem)
{
    if( m_nCount == m_nMax )    // Verify the overflow
        Resize();               // Resize the vector before inserting elem
    m_pVect[m_nCount++] = elem; // Insert the element at the end
}
```

Figure 15: Insert function using Object Oriented Programming

3.4 Abstract Data Types

As we have seen at the beginning of this article, the requirement was: "the user needs some black box encapsulating the vector data structure". But the type of elements to be inserted was not specified. It means, the black box should be useful independently of the data type. A first attempt to solve this problem is frequently found in many systems in which a global *typedef* is created as showed in Figure 16.

This technique only allows to reduce cost of maintenance of code if we need some other type like float, double, etc. Nevertheless, this code is still not useful for a user who needs two (or more) vectors with different types (i.e. char, float, short, etc) at the same time in the same system. Fortunately, some languages like *C++* [7] have templates, which allow the parameterization of the element type. In Figure 17 we can see a short implementation for the concept of vector considering an abstract data type.

Using templates is important from the didactic point of view because the student can really see, in terms of a programming language, the concept of Abstract Data Types. If students learn the philosophy behind the templates they will be mainly focused on the concept of the data structure and not on type-specific implementations. *C++* templates are probably the closest programming construct to implement Abstract Data Type.

```
typedef int Type;
class CVector
{
    private:
    Type*m_pVect;                   // Pointer to the buffer
        ...

    public:
    void Insert(Type elem);         // Insert a new element
        ...
};
```

Figure 16: First try to create a generic vector class

```
template <typename Type> class CVector
{
    private:
    Type*m_pVect;         // Pointer to the buffer
    int m_nCount,         // Control how many elements are actually used
        m_nMax,           // Control the number of allocated elements
        m_nDelta;         // To control the growing
    void Init(int delta); // Init our private variables, etc
    void Resize();        // Resize the vector when occurs an overflow

    public:
    CVector(int delta = 10);     // Constructor
    void Insert(Type elem);      // Insert a new element
    // ...
};
```

Figure 17: Generic vector class using templates

```
// Class CVector implementation
template <typename Type> CVector<Type>::CVector(int delta)
{
   Init(delta);
}

template <typename Type> void CVector<Type>::Insert(Type &elem)
{
    if( m_nCount == m_nMax )    // Verify the overflow
        Resize();               // Resize the vector before inserting elem
    m_pVect[m_nCount++] = elem; // Insert the element at the end
}
```

Figure 18: Implementing methods for template class CVector

Among the advantages of this approach, we have:

- This implementation for the concept of vector is useful for as many data types as necessary.
- The maintenance around the code corresponding to the concept of "vector" is well located. Any modification of this data structure implementation will be in only one class. We just need to modify the code once.
- The use of templates does not affect the performance because the type-specific code is generated when the program is being compiled according to the data types required by the user.

Among the drawbacks, we can highlight:

- The abstract parameter for a template must be defined before the program is compiled. It would not be useful if we need to define the type at run time.

3.5 Design Patterns

A pattern is a recurring solution to a standard problem. When related patterns are woven together they form a "language" that provides a process for the orderly resolution of software development problems. Pattern languages are not formal languages, but rather a collection of interrelated patterns, though they do provide a vocabulary for talking about a particular problem. Both patterns and pattern languages help developers communicate architectural knowledge, help people learn a new design paradigm or architectural style, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience[1].

Expert programmers do not think about programs in terms of low level programming language elements, but in higher-order abstractions. This can be seen in the different sections across this article and the reader can compare the advantages and drawbacks of these various levels of abstraction.

This section is focused on Designs Patterns and specifically on Iterators. Further reading about this topic can be found in [2]

[1]Guest editorial for the Communications of the ACM, Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10, October 1996

A robust data structure must always provide some mechanism to execute some operations over the data it contains. There are several techniques to achieve this goal. One of them was introduced by the Standard Template Library (STL) [6]. STL implements several containers for the most common data structures like linked list (class list), vector (class vector), queue (class queue) which are implemented using templates. All these classes also have an internal user-defined type called **iterator** which allows to iterate through the data structure. The internal instructions to walk through binary trees are different from those used for linked lists. However, a function to print and iterate through these two data structures will be exactly the same and it can be implemented as a template. In Figure 19 we can see both the list and vector data structure being printed by the same function (*Write*()).

```
#include <vector>   // without .h
#include <list>
#include <iostream>

using namespace std;

template <typename Container> void Write(Container &ds, ostream &os)
{
    Container::iterator iter = ds.begin();
    for( ; iter != ds.end() ; iter++ )
        os << *iter << "\n";
}

int main(int argc, char* argv[])
{
    list<float>      mylist;
    vector<float>    myvector;

    for( int i = 0 ; i < 10 ; i++ )
    {
        mylist .push_back( i );
        myvector.push_back(i+50);
    }

    Write(mylist,   cout);
    Write(myvector, cout);
    return 0;
}
```

Figure 19: Using a common function for both a list and a vector

In Figure 19 we should pay special attention to the *Write*() function. We will now explain that function in more details. First, the *Write*() function has to be a template in order to receive any possible class as parameter (list and vector in this case). Second, all the containers provided by STL have methods to retrieve the information at the beginning and at the end. That is the reason to use *begin*() and *end*() methods.

Probably the most important issue to highlight is the abstraction represented by the instruc-

tion **iter++** contained in the for statement. This operation represents the **operator++** which has been overloaded by both iterators. Obviously, the code for walking to the next element in the list is different than the one for a vector, but the **operator++** provides a higher level of abstraction. Finally, ***iter** gives us access to the element where the iterator is at that moment.

Another well known technique used for applying common operations to all elements in a container are the **Callback** functions. In this technique the user provides a pointer to a function which will be called by the container for each element (the Callback function). For applying the operation implemented by the callback function to the elements of the container, the data structure only needs to implement a method to walk through the elements calling this function for each one of them.

A more sophisticated technique to overcome this problem is known as **Function Objects**. In Figure 20 we can see an example where we increase by one all the elements into the container. This technique is specially useful for those who do not like to use pointers to functions. In terms of safety, function objects are probably safer than function pointers.

Among the advantages of this approach, we can specially highlight the higher level of abstraction.

The only probable drawback for a new STL programmer is to understand the extremely long error messages and warnings generated by nested templates. This problem can be quickly overcome by reading the bibliography presented in [6].

3.6 Preparing the code for future workgroups projects

An important point to be discussed here is to prepare the student/programmer (or user) for working in groups. If each student/programmer develops a part of a bigger system, it is necessary to decide first how the sub-systems are going to communicate to each other when the whole system is finished. As programmers, we know that designing software needs a lot of care. Nevertheless, it is not easy to avoid some conflicts when we join all the subsystems. In this section we will show a common problem and we will present a possible solution.

In order to explain the problem, we suppose that there are only two programmers who are in charge of programming a Linked List and a Binary Tree, respectively. We also suppose that the programmers have already read the section 3.4 about templates and they want to use that level of abstraction to solve the problem.

The first student should implement the Linked List in a class, probably named **CLinkedList**. This data structure also requires an internal structure to store each node which will be called **NODE**. This internal structure should be designed to store the data for the node and the pointer to the next node. As we saw above, the CLinkedList will be designed as a template class. It means, the structure for internal nodes (NODE) should also be designed as a template.

A possible design solution is showed in the Figure 21.

The code to define the Binary Tree will be represented by the class **CBinaryTree** could be similar except that the struct NODE should consider the *m_pLeft* and the *m_pRight* pointers instead of *m_pNext*.

If we try to use these two files, `LinkedList.h` and `BinaryTree.h`, in the same system, we will have a duplicated name because the name NODE exists in both of them but it has different internal structures. If we change one of them, we will probably change it by another identifier which could not be as intuitive as NODE. If we pay more attention to this problem, we will see that the problem here is not to change the name because it is appropriate for both of them.

Another problem in these codes is that, in both cases, the structure NODE was designed for internal used only. It means, the user does not need to know about its existence. In other words, they should have been defined inside the respective classes and not at the same level with

```
class CMyComplexDataStructure
{
    vector<float> m_container;
    public:
    void Insert(float fVal) { m_container.push_back(fVal); }
    template <typename objclass>
    void sumone(objclass funobj)
    {
      vector<float>::iterator iter = m_container.begin();
      for (; iter != m_container.end() ; iter++)
             funobj(*iter);
    }
};

template <typename objclass>
class funcobjclass
{
    public:
    void operator ()(objclass& objinstance)
    {
      objinstance++;
    }
};

int main(int argc, char* argv[])
{
     CMyComplexDataStructure ds;
     ds.Insert(3.5);
     ds.Insert(4.5);
     ds.Insert(6.5);
     ds.Insert(3.9);

     funcobjclass<float> x;
     ds.sumone(x);

     return 0;
}
```

Figure 20: Implementing a function object

```
// Code for the student implementing a Linked List
// File LinkedList.h

// Structure for nodes in a Linked List
template <typename T> struct NODE
{
    T m_data;                // The data goes here
    struct NODE<T> *m_pNext; // Pointer to the next node
    static long id;          // Node id

    NODE()                   // Constructor
        : m_data(0), m_pNext(NULL) {}

    // More method go here
};

long NODE::id = 0;  // Inicialization of the node's id

template <typename T> class CLinkedList
{
    private:
    NODE<T> *m_pRoot;        // Pointer to the root

    public:
    // More methods go here
};
```

Figure 21: First CLinkedList class definition

it. If the programmer designs the code using this tip we avoid future problems. The following code shows the same code with these modifications.

This better design technique solves the problem partially, however, realistic problems do not involve only two classes. For example, software that uses a worksheet and a text editor from two different developers, it is unlikely that there won't be any name conflicts. How to avoid these, we'll discuss in the next section. For this approach, we can highlight the following advantages:

- We can prevent local name conflicts when the software is developed by a group.
- The scope for the internal NODE structure is better defined now preventing future problems.

Among the drawbacks, we can highlight:

- It does not allow to avoid some conflicts when the software is made by independent developers or large groups.

```
// Code for the student implementing a Linked List
// File LinkedList.h
template <typename T> class CLinkedList
{
    private:    // Structure for nodes in a Linked List
    struct NODE
    {
        T              m_data;   // The data goes here
        struct NODE * m_pNext;   // Pointer to the next Node
        // Some methods go here
    };
    NODE*   m_pRoot;             // Pointer to the root
    public:    // More methods go here
};
```

Figure 22: Second CLinkedList class definition

```
// Code for the student implementing a Binary Tree
// File BinaryTree.h

template <typename T>
class CBinaryTree
{
    private:        // Structure for nodes in a Binary Tree
    struct NODE
    {
        T              m_data;          // The data goes here
        struct NODE * m_pLeft, m_pRight;// Pointer to the left and right node
        // Some methods go here
    };
    NODE*   m_pRoot;                     // Pointer to the root
    public:    // More methods go here
};
```

Figure 23: CBinaryTree class definition

3.7 Namespaces

Let's think about a software which has to use a worksheet and a text editor. Evidently, we will not attempt to build them from scratch. We will try, for example, to use Excel and Word to achieve our goal. The same idea could also be applied for any Operating System like UNIX, Linux, etc. In the specific case of MS Excel and MS Word they make available some methods and DLL's to interact with them directly by program. We can extract the contained code from the DLL's and convert it to *C++* classes and methods by using the **#import** directive according to the code presented in Figure 24.

```
#import <mso9.dll> no_namespace rename("DocumentProperties",
                                       "DocumentPropertiesXL")
#import <vbe6ext.olb> no_namespace
#import <excel9.olb>

rename("DialogBox", "DialogBoxXL") \
                    rename("RGB", "RBGXL") \
                    rename("DocumentProperties", "DocumentPropertiesXL") \
                    no_dual_interfaces

#import <msword9.olb> rename("DialogBox", "DialogBoxWord") \
                    rename("RGB", "RBGWord") \
                    rename("DocumentProperties", "DocumentPropertiesWord") \
                    no_dual_interfaces
```

Figure 24: Importing code from a DLL in order to be used

The #import directive generates two files with extensions TLH (Type Library Header) and TLI (Type Library Implementations) which are automatically added to the project by a hidden #include.

As the reader can imagine, the number of generated classes from a software like Excel is big. If we apply the #import directive to extract the classes and methods from Word and Excel in the same system, you can see many classes with the same name in both of them. To prevent this kind of problems, the ANSI *C++* was enriched with the **namespaces**.

The syntax to create a namespace is quite similar to classes and structs. The difference is that we can not create instances using namespaces. They were exclusively designed to provide a new and higher encapsulation level which can contain global variables, functions, methods, classes, structs. In Figure 25 we can see the code to for a possible namespace named **MyNS**.

When we use namespaces we are increasing in one the level of encapsulation. It means, to identify any member we have to add the name of the namespace NS1 and the scope resolution operator :: like accessing a method. The Figure 26 illustrates how to implement the members for the namespace *MyNS*. Among the advantages of this approach, we can highlight:

- We can prevent conflict when the software is developed by two different groups or even by different manufacturers.

We do not see considerable drawbacks of using templates.

```
// File MyNS.h
namespace MyNS
{
    int gnCount;   // Global counter

    // Some small function implemented inline
    double Addition(double a, double b)  // Global function
    {    return a+b;     }

    // Some prototypes
    long factorial(int n); // Global function

    class CTest
    {
     public:
                Test();
        void    Method1();
    };
    // More variables, functions, methods, classes, structs go here
};
```

Figure 25: Using namespaces

3.8 Run time data type definition

As we saw in section 3.4, the user is forced to define the abstract parameter before using a template. Nevertheless, there are several cases where the type is only known at run time.

Probably the most obvious case is a database engine where the user defines, at run time, if a field is integer, char, float, etc. but, it is not necessary to compile the program again. All this flexibility must be incorporated into the program trying to minimize the possible impact in maintenance and performance.

A possible solution, which is frequently found into the programs is the use of a switch to determine the data type selected by the user. In Figure 27 we can see this technique.

This code solves the problem partially, but it creates new ones. Among the new problems we can highlight the following ones:

- Each new possible type would require a new case into the switch, which basically would contain the same code, only changing the type.
- Using this approach, we are almost forced to implement the *Insert_Elements*() function as a template. This problem is propagated to the functions internally used by *Insert_Elements*().
- This technique has effects over the performance because each time we have to use the template we will probably need a similar switch.

```
//File NS1.cpp
#include ‘‘MyNS.h’’ long MyNS::factorial(int n)
{
    // Some complex code goes here
}

// Constructor for class CTest
MyNS::CTest::CTest()
{
    // Initialize something
}

void MyNS::CTest::Method1()
{
    // You have to write your code here
}
// More implementation for NS1.h prototypes goes here
```

Figure 26: Implementing methods contained into namespaces

In front of this problem, it is common to minimize the possible effect of this small and inappropriate *if*, but if repeated several times the difference could be considerable. If the reader wants to know that difference, test the code shown in Figure 28 and then repeat the same experiment without the *if* which means, with the loop empty.

If we execute the code showed in Figure 28 with the loop empty using a conventional computer (1.2 GHz Intel Pentium III Processor, running Windows XP, 512 Mb of RAM memory, using MS Visual C++ 6.0 compiler[2]), and compare their result with the one obtained with the empty loop. We will see a considerable difference (just try!). This is enough to have an idea of the effect when a *if* like this is added.

[2]We did not change the optimization switches used by this compiler

```
cin >> option;
switch(option)
{
    case INT_TYPE_ENUM:
    {    CArray<int> a;
        Insert_Elements(a);
        break;
    }
    case DOUBLE_TYPE_ENUM:
    {    CArray<double> a;
        Insert_Elements(a);
        break;
    } // More cases for other types go here
}
```

Figure 27: First try to use user-defined types at run time

```
const unsigned long max = 500 * 1000 * 1000;
const unsigned long step = 100 * 1000 * 1000;
unsigned long count;

    for(count = 0L ; count < max ; count++)
        if( count % step == 0L )
            printf("some text ...\n");
```

Figure 28: Comparing the performance after adding a single if into a loop

But the question now is: is it possible to design a generic class able to work even when the data type is defined at run time?. The answer is: yes. Lets imagine our old *CVector* class, but we will redesign it considering the restrictions discussed in this section. The new *CVector* class is shown in Figure 29.

```
class CVector
{
    private:
    void **m_pVect;    // Pointer to the buffer
    int m_nCount,      // Control how many elements are actually used
        m_nMax,        // Control the number of allocated elements
        m_nDelta,      // To control the growing
        m_nElemSize;   // Element size

    // Pointer to the function to compare
    int (*m_lpfnCompare)(void *, void*);
    void Init(int delta);   // Init our private variables, etc
    void Resize();          // Resize the vector when occurs an overflow

    public:

    CVector( int (lpfnCompare)(void *, void*),
                        int nElemSize, int delta = 10); // Constructor
    void Insert(void * elem);         // Insert a new element

    // More methods go here
};
```

Figure 29: Designing a vector without templates

The first restriction if we only know the data type at run time is that we can neither use templates nor define our dynamic vector using a specific type. That is the reason to have our *m_ppVect* as a *void* $**$. If we do not know the size of each elements we need a variable to control it (*m_nElemSize*). Another new problem is how to compare two elements in order to insert them in the correct position. As the user will exactly know the type at run time he could also provide a generic function to compare two elements (*m_lpfnCompare*). We will come back to explain this last member after analyze Figure 30.

```
// Class CVector implementation
CVector::CVector(int (*lpfnCompare)(void *, void*),
                     int nElemSize, int delta)

{    Init(delta);
     m_lpfnCompare = lpfnCompare;
     m_nElemSize = nElemSize;
}

void CVector::Insert(void *pElem)
{
    if( m_nCount == m_nMax )     // Verify the overflow
        Resize();                // Resize the vector before inserting elem
    m_pVect[m_nCount++] = DupBlock(pElem); // Insert the element at the end
}

void* CVector::DupBlock(void *pElem)
{
  void *p = new char[m_nElemSize];
  return memcpy(p, pElem, m_nElemSize);
}

// More implementations go here
```

Figure 30: Trying the define user-types at run time

A possible implementation for a function to compare integer can be seen in Figure 31.

```
int fnIntCompare( void *pVar1, void *pVar2 )
{
    // < 0 if *(int *)pVar1 < *(int *)pVar2,
    // > 0 if *(int *)pVar1 > *(int *)pVar2
    // else 0

    return *(int *)pVar1 - *(int *)pVar2;
}
```

Figure 31: Function used to compare two integers considering two generic addresses

3.8.1 Moving into the real world

A real case is presented by the Open Database Connectivity (ODBC) [3] which is frequently used to access heterogeneous databases. ODBC is basically a standard which provides the mechanisms to access different databases (i.e. Oracle, Microsoft SQL Server, Sybase, etc) homogeneously.

The main point to discuss here is how ODBC can deal with several databases from different manufacturers and formats minimizing the effect over performance. This lose of performance can be dismissed we get something in return.

One of the best techniques to achieve that goal is to use pointer to some functions. According to this technique, each manufacturer (for each new ODBC driver) must provide a set of predetermined common function to execute all the possible operations with a database (i.e. insert, remove and update registers, execute some SQL instruction, etc). This set of common functions are registered into the operating system and recognized as a new driver.

Just to simplify the process we suppose in this section that the required operations for all the databases are: *Open*(), *InsertNewRegistry*(), *DeleteRegistry*(), *MoveNext*(), *MovePrev*() and *Close*(). The *POOL* containing the pointer to functions could have been defined in Figure 32.

```
// odbc.h: shared file for new databases manufacturers

struct POOL
{
    long (*lpfnOpen)(char *pszDBName);
    void (*lpfnInsertNewRecord)(long DBID);
    void (*lpfnDeleteRecord)(long DBID);
    void (*lpfnMoveNext)(long DBID);
    void (*lpfnMovePrev)(long DBID);
    void (*lpfnClose)(long DBID);
    // More operations go here
};
```

Figure 32: Pool of pointers to functions

If a new DB manufacturer *ABC* wants to add his driver to ODBC, they will probably prepare their own pool according to Figure 33.

```
POOL ABCPool = {&ABC_Open,           &ABC_InserNewRegistry,
                &ABC_DeleteRegistry, &ABC_MoveNext,
                &ABC_MovePrev,         &ABC_Close};

long ABC_Open(char *pszDBName) {   // Some complex code go here }
```

Figure 33: Preparing the pool of pointers to functions

When the user opens a database source, ODBC creates a new connection with the *POOL* of functions corresponding to that database. After that point the client will be able to execute all the operations directly using that pointers. All this complexity is hidden by ODBC and it is transparent for the final user. The only requirement is that all the *Open* functions should have the same prototype. The same idea is applied for the rest of the functions.

Frequently, the pointers to functions are confused with "complex programming techniques", in fact that is the reason to have many languages which do not use pointers. We show their potentiality in the following section.

3.8.2 Reducing code size through pointers

Using pointers to functions could help us to reduce the code considerably, but if they are not used properly can create more problem than they solve. It is important to comprehend the advantages and disadvantages, in terms of performance and specially code maintenance, between the two codes shown in Figure 34 and 35 respectively.

```
// Code A
float a, b, c; int opt;

// enter the operands
cin >> a >> b;

// OPT-> 0-Addition, 1-Subtraction, 3-Multiplication, 4-Division
cin >> opt;

switch( opt )
{
    case 0: c = Addition(a, b);              break;
    case 1: c = Subtraction(a, b);           break;
    case 2: c = Multiplication(a, b);        break;
    case 3: c = Division(a, b);              break;
    // more cases operations go here
}
```

Figure 34: Calling similar functions by using switch

As the reader can see, the four involved functions have the same prototype. All of them take two float values as parameters and return a float. Using pointers to functions we can reduce the number of cases and, consequently eliminate the switch. In the other hand, code using pointers

```
// Code B
// User type to simplify the declaration

typedef float (*lpfnOperation)(float, float);
// CVector of pointer to functions
lpfnOperation vpf[4] = {&::Addition,       &::Subtraction,
                        &::Multiplication, &::Division};

float a, b, c; int opt;
// enter the operands
cin >> a >> b;

// enter the operation 0-Addition, 1-Subtraction, 3-Multiplication, 4-Division
cin >> opt;

// The next line replaces the switch and replaces the whole switch
c = (*vpf[opt])(a, b);
```

Figure 35: Calling similar functions by using pointer to functions

is not easy to optimize by compilers. The last instruction in Figure 35 $c = (*vpf[opt])(a, b)$; could potentially call several functions and the compiler will have a hard time to optimize it. In the other hand, code showed in Figure 34 is easier to optimize for a compiler.

Even more, if we create the vector vpf using dynamic allocated memory it could be even more flexible.

Among the advantages of this approach, we can note:

- We now have the possibility to select, at run time.
- Scalability. It allows to customize the pointers and, consequently, to execute user-specific tasks even when the program is already compiled.
- We can dramatically reduce our code and simplify the maintenance especially when a switch has many similar cases.

Among the drawbacks, we can highlight:

- More complexity for programmers. Pointers to functions are useful in terms of flexibility but they are dangerous when used improperly.
- More complexity for compilers. The use of function pointers makes compiler optimizations difficult and can result in degraded performance.
- This code does not prevent possible conflict when the code is programmed by many developers simultaneously. However, we will explain how to overcome this drawback in the following section.

3.9 Interfaces

The idea that inspired the concept of *class* is **encapsulation**. A class is created based on the following idea:

If we have same variables that are used only for groups of functions, lets encapsulate them and create a **class**. *The variables become* **attributes** *and the functions are now known as* **methods** [7].

This process is well know as **encapsulation** and is considered a the keystone of object oriented programming. We will consider the class *CVector* presented in section 3.3 as the start point. Besides the methods to modify the contained data (i.e. *Insert*() and *Remove*(), etc), the class should have some methods like *Write*() and *Read*() to make the objects from this class persistent. Up to now, all these four methods belong to our class but, if we note, the methods *Insert*() and *Remove*() do not have semantic relation with the *Write*() and *Read*(). The question is: if there is no semantic relation between this two groups, but they belong to the same class, how should we code them? the answer is: creating two **interfaces**, one containing *Write*() and *Read*() and the second one containing *Insert*() and *Remove*(). We will call these two interfaces **IPersist** and **IContainer** and in this case the class *CVector* must implement both of them.

Figure 36: A class with interfaces

For coding purposes, an interface can be implemented as an abstract class. An interface does not have code, its only goal is define the methods to communicate the users with the object. If a class is inherited from an interface, it must implement all the methods contained in it.

```
// Class CVector definition
class CArithmetic
{

    private:
    // Some attributes go here

    public:
    static double Addition       (double a, double b);
    static double Substraction   (double a, double b);
    static double Multiplication (double a, double b);
    static double Division       (double a, double b);
    // More methods go here
};
```

Figure 37: Class to execute arithmetic operations

This new approach clearly shows us a better semantic organization which is useful for maintenance and scalability purposes. Classes like *CVector*, *CList*, *CBinaryTree* will need to imple-

ment the same interfaces but, they will made that with different codes because, for example, *Insert()* an element is different in *CList* and *CVector*. The implementation is different but the method's name will be the same.

There are some libraries like ATL (ActiveX Template Library) [8] which has already implemented several default interfaces, we just need to use them. The same idea is being used by Java and many other modern languages.

Among the advantages of this approach, we have:

- The methods are clearly well organized semantically.
- It makes maintenance easier and enhances scalability preventing future problems.

Among the drawbacks, we can point out:

- It is necessary to be more familiar with templates, etc. This drawback can be minimized if compared with the advantages we have using ATL.

3.10 Creating fault tolerant software

An important point to be considered when a software is being designed is the fault tolerance. It means, if something unexpected happens, the software should be capable to return to a known point without forcing the user to close it. Just to visualize this approach, lets think about a class *CArithmetic* to execute arithmetic operations like *Addition*(), *Substraction*(), *Multiplication*() and *Division*(); all of them returning a double value according to the following code.

```
double x, y, z; ...
try // our critical section with possible exceptions begins here
{
    z = CArithmetic::Division(x, y);
    cout << z;
}

catch (int errorcode) // catching the exception
{
    // Display some message, etc.
}
```

Figure 38: Using an exception

```
double CArithmetic::Division(double a, double b)
{
    if ( b == 0.0 )
        throw (int)0; // an exceptions is thrown here
    return a/b;
}
```

Figure 39: Throwing an exception

Sometimes we can control if something wrong happened because the function return an especial code. For example, the factorial function can only return positive values an we could return -1 if the user calls it with a negative parameter but, unfortunately, it is not always possible. Lets think about the *Division(x,y)* method called with $y = 0$. In this case we could not return -1 or 0 to say that something wrong happened because both values (0 and -1) could also be the result for a division.

An elegant solution for this case is using exceptions. In Figure 38 and 39 we can see how to use them.

If something wrong happens the program counter will exit the function restoring the stack and looking for some catch with the appropriate type (int in this case).

Among the advantages of this approach, we can highlight:

- Fault tolerance and clarity.

Among the drawbacks, we can highlight:

- More complexity and knowledge is required.

3.11 Programming language considerations

The chosen programming language is, obviously, another important point to analyze. Each programming language has its own strengths and weaknesses. The goal is to use each programming language for those tasks where it is more effective.

In the specific case of data structures we recommend the use of ANSI $C++$ is a very popular programming language, which provides many features for the design of flexible data structures. With C it shares very efficient compilers but offers better support for data abstraction through its object-oriented features. It was designed as a general-purpose programming language, with a bias toward systems programming that supports efficient low-level computation, data abstraction, object oriented programming and generic programming [7].

Today there are many programming languages, but if the goal is speed and efficiency we have to consider $C++$. This is the main reason $C++$ was chosen as the default programming languages for $UNIX^{TM}$, Linux, Microsoft Windows, Solaris, etc. On the other hand we also have to consider the extra time necessary to debug programs in $C++$.

$Java^{TM}$(Sun Microsystems, Inc.) [4] is also another interesting alternative specially for web development that is increasingly gaining popularity. While it's performance is still somewhat inferior to $C++$ programs, its support of automatic memory management and strict type checking discipline prevents many common programming errors. Since programmer time is more valuable and more expensive than CPU time, nowadays, and compilers have already become considerably better for Java, future software development in Java is likely going to increase. Its strong type checking and garbage collection make it also attractive for student programming courses since they prevent common mistakes and allow students to concentrate on the main task at hand and not waste time on errors that can be prevented at the programming language level.

Therefore, when program performance is not critical-which is the case for most software projects-language features should be more important that low level efficiency. While Java is a great step in the right direction towards preventing simple programming mistakes, many, in particular $C++$ programmers, miss templates. However, future versions may incorporate that feature as well, which would then make Java the perfect vehicle for teaching data structures in the class room.

3.12 Beyond the programming language and operating system barriers

All the existing programming language present its own strengths and weaknesses. In order to improve our final product, we have to use the best of each language. According to this point of view, and, even more, considering the exponential growth of internet, our software should be designed to be used not only from clients programmed with the same language but also from programs from other languages.

Taking into account these considerations, it is very important to create cross-programming language objects. The same requirement holds for compatibility among different operating systems. There are many programming languages and models which allow to achieve this goal.

Among the most important we can cite *C++*, Java from SUN Microsystems (java.sun.com), and the newest ones like Microsoft .NET platform , etc.

It is also important to consider STL [6] as an alternative to create portable data structures. This library is part of ANSI *C++* and it must be supported by any *C++* compiler.

Using different programming languages, we have the opportunity to take advantage of the best features of each one of them.

4 The quality of used algorithms

All the techniques presented from section 3.1 to 5 would not be useful whether the algorithm being used is not efficient. The efficiency and computational cost of an algorithm will always be the most important point when it is being designed [5]. We also have to pay attention to clarity and maintenance of the code.

Just to illustrate this point, we analyze the well known Fibonacci number series, which is defined recursively as follows: $fibo(0) = 0$; $fibo(1) = 1$; $fibo(\text{n}) = fibo(\text{n-1}) + fibo(\text{n-2})$. Recursion is an important and useful construct provided by most programming languages, which enables the formulation of elegant and concise solutions for many problems. However, naive use of recursion can degrade performance and iterative solutions often provide faster alternatives.

Figure 40 shows a direct recursive implementation of the Fibonacci series in the *C* programming language:

```
unsigned long fibo(unsigned n)
{
    if( n < 2 )
        return 1;
    return fibo(n-1) + fibo(n-2);
}
```

Figure 40: Recursive implementation for fibonacci numbers

An alternative, iterative version is shown in Figure 41. It requires more code than the first solution and is somewhat harder to understand-although, given the smallness of the function, it is still easy to grasp overall. On the other hand, it requires less memory (stack space for parameters or locals). Even more important, it is considerably faster since the recursive version does considerably more work as shown in Figure 41. In contrast, the iterative implementation only executes a loop to produce the same output at linear cost. Moreover, as $\frac{F_{n+1}}{F_n} \sim \frac{(1+\sqrt{5})}{2} \sim 1.618$ then $F_n > 1.6^n$, and to compute F_n we need 1.6^n recursive calls, that is the run time of the recursive version is exponential in the size of the input! Obviously, the naive use of

recursion, even though attractive because of the direct correspondence to the original problem, can have serious drawbacks. Figure 42, which shows the recursive computations performed for *fibo*(4), demonstrates the wastefulness of the recursive implementation by making the repeated calculation of the same values obvious.

```
unsigned long fibo(unsigned n)
{
    if( n < 2 )
        return 1;
    unsigned long f1 = 0, f2 = 1;
    unsigned long output;
    do
    {
        output = f1 + f2;
        f1      = f2;
        f2      = output;
    }while(--n >= 2);
    return output;
}
```

Figure 41: Iterative implementations for fibonacci numbers

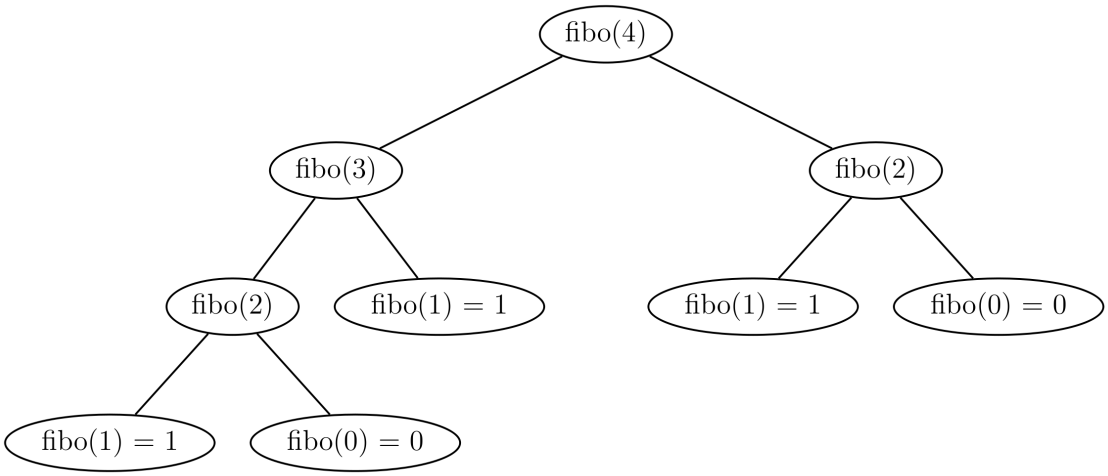

Figure 42: A class with interfaces

5 With a little help from the compiler: writing clear but efficient code

While teaching students the importance of efficient data structure and algorithm design and implementation, we do not want to loose sight of the major goal of writing clear and readable code. In the past, when optimizing compiler technology was still in its infancy, clarity and efficiency were sometimes at odds. Frequently this lead programmers, particulary C programmers, to write efficient yet hard to read and understand code. However, today's optimizing compilers can help us concentrate on the essential steps of choosing a good algorithm and adequate data structures and then leave much of the performance tuning to the compiler's optimizer. Knowing what compilers can and cannot do, is therefore important when we teach students how to implement their algorithms.

In section 4, we use the fibonacci function as an example to show students to be wary of a simple, or rather naive, recursive implementation because of their associated exponential time and space costs. However, often recursion is the clearest way to state an initial solution and we can then either transform it to iteration by hand, or in some cases, let the compiler do that task for us. To illustrate this point, we can use the binary search routine shown in 43, which finds a value `key` in a section of array `a`, from element `a[l]` to `a[h]`, inclusively. The array contains double precision numbers and we assume that it is sorted in descending order. If the `key` value is found, its position in the array is returned, otherwise the routine returns `-1` to indicate that the value is not present.

```
int bsearch(double a[], double key, int l, int u)
{
    int m;
    if (u<l)
         return -1;
    m = (l+u) / 2;
    if (a[m] == key)
         return m;
    if (a[m] < key)
         return bsearch(a, key, l, m-1);
    else
         return bsearch(a, key, m+1, u);
}
```

Figure 43: Recursive binary search routine. Since *bsearch*() is tail-recursive, a good optimizing compiler will automatically turn the tail recursion into iteration, as shown in 44.

Binary search has a very clear and concise recursive formulation. First, there is no value to be found in an empty array section; this is checked by the comparison of `u` and `h`. Then the index of the pivot element is computed into a helper variable `m`, and if `key` is found there, the index is returned. If `key` is larger than the pivot element, since the array is sorted in descending order, the key can only be found in the lower half and recursively the lower half is searched, otherwise, the upper half is searched in a recursive call. Since there are no computations after the recursive call to `bsearch`, the routine is tail-recursive and we can show students how such a routine can be mechanically converted into an iterative version: simply perform the assignments of the actuals to the formals in the recursive call and replace the call by a goto to the start

of the routine (of course, in practice one would wand to merge the goto and the if to a `while loop`) and not use `goto` in the program.

Since this is a simple, mechanical procedure, it should be easy to realize for students that in fact, a good compiler should be able to do this automatically, and for illustration, we can present the output of the widely available GNU C compiler for this routine, shown for the Alpha processor in 44.

Of course, not all recursive functions can be turned into iterative versions without explicitly simulating the stack. However, many functions that are not naturally tail-recursive, can easily be rewritten to be tail-recursive by using an *accumulator*, which is used to carry intermediate results from invocation to invocation.

```
bsearch:
     .frame $30,0,$26,0
$bsearch..ng:
         .prologue 0
$L5:
         cmplt $19,$18,$1        # !(u<l) goto L2
         beq $1,$L2              #
         lda $0,-1               # m = -1 and return (from L7)
         br $31,$L7
$L2:
         addl $19,$18,$1         # compute u+l
         srl $1,63,$2            # divide by 2 with shift
         addq $1,$2,$1           #
         sra $1,1,$0             # m = (u+l)/2
         s8addq $0,$16,$3        # compute address of a[m]
         ldt $f11,0($3)          # a[m] -> $f11
         cmpteq $f11,$f17,$f10   # a[m] == key goto L7
         fbne $f10,$L7
         cmptlt $f11,$f17,$f10   # a[m] < key goto L4
         fbeq $f10,$L4
         subl $0,1,$19           # u = m-1
         br $31,$L5              # goto L5
$L4:
         addl $0,1,$18           # l = m+1
         br $31,$L5              # goto L5
$L7:
         ret $31,($26),1         # return m (in $0)
         .end bsearch
```

Figure 44: The recursively written binary search routine from Figure 43 compiled by the GNU C compiler (*gcc*) with optimization option leve O2 for the Alpha processor. The compiler automatically turned the recursion into iteration and chose to perform the integer division by 2 with a shift operation. This code is as efficient as any iterative version written by hand but the recursive formulation is clearer.

Figure 45 shows a simple example for this technique for the factorial function. Factorial is rewritten using a helper function (*fact_helper*) that takes two arguments a and n, and which recursively computes $a \star n!$. By accumulating intermediate results in the first argument of the helper function, the recursive function becomes tail-recursive and the factorial function then is simply implemented as a specific call to the helper function that initializes the accumulator to 1. By declaring the helper function static[3], the programmer communicates to the compiler that the helper function is used only locally, so that the compiler can safely delete its code if it can inline the helper into (all) its callers. Moreover, when we compile this version of the factorial function with an optimizing compiler, it will automatically turn the recursion into iteration, inline the helper function into the *fact* function, which eliminates the additional call

[3] `static` is a keyword in the C programming language, which, when used to qualify a function, restricts the scope (visibility) of the function to the file in which the function is defined, so that the function cannot be called from functions in different files.

overhead introduced by the call to the helper function, and eliminate the helper function from the program text since it is not called from anywhere else. That is, without any additional overhead in program size or execution time, the programmer can write the function recursively, in a form very close to its mathematical definition.

```
typedef unsigned long ulong;

static ulong fact_helper(ulong a, ulong n)
{
  if (n == 0)
    return a;
  else
    return fact_helper(a*n, n-1);
}

ulong fact(ulong n)
{
  return fact_helper(1, n);
}
```

Figure 45: Tail-recursive version of the factorial function. Using a helper function, factorial can be implemented with tail recursion, which is automatically converted to iteration by an optimizing compiler.

Obviously, the example is trivial enough that it would be easy to write iteratively in practice. However, more complicated recursive functions can be implemented with this technique in a clear and concise way without sacrificing performance, thanks to modern compiler technology[4].

What students can take away from these examples is that it pays to know just a little bit about what is going on underneath the covers. Efficient code does not have to unreadable, and more often than not, code that is hard to optimize by a compiler, will also be hard to understand, read, and maintain by human programmers. Therefore, when we teach efficient data structure and algorithm design, it is important to not loose sight of what should be the overriding goal: producing good, easy to maintain designs without disregarding performance. And as the examples have illustrated, efficiency and simplicity can usually be achieved at the same time.

6 Summary and Conclusions

In this article we presented different approaches to implement data structures. Almost all these approaches can be used for any data structure.

In section 2, we presented a categorization for all those techniques according to some criteria like dynamicity, data protection, encapsulation and data abstraction.

In section 3, we presented twelve different approaches to implement data structures such as Modular Programming, Object Oriented Programming, Abstract Data Types, Design Patterns, code for workgroups, namespaces, fault tolerance, among other.

[4]Of course, in practice one should always check that the compiler at hand is in fact smart enough to perform the outlined transformations. If not, rewriting a function in the outlined way is still useful as a first step in converting the recursive implementation to an iterative implementation by hand.

In section 4, we presented the importance of designing a good algorithm. It is important to pay attention before using recursion. It is an excellent alternative when used properly.

In section 5, we explained the importance of taking advantage of an optimizing compiler to design code that is both readable and efficient.

There are many alternatives to speed up the whole performance of a program. The first and most important is to improve the involved code and the data structures, but it is also important to take advantage of CPU idle time produced by I/O operations. An interesting alternative to achieve this goal are threads [1]. When we design a Multi-Threading program, we are not only preparing the code for a multi-processor environment but also to take advantage of the idle time even with only one processor.

Another important point for data structure coding, and software development, is the documentation. It makes easier the work in groups and the software maintenance during the whole cycle of development.

Among the final conclusions we can highlight:

- An efficient data structure is the result of combining the best of all these techniques according to the circumstances.
- It is very important to recognize the potential differences among the approaches presented in order to apply them properly.

Acknowledgements

The authors would like to thank Alex Cuadros-Vargas and Eduardo Tejada-Gamero from University of Sao Paulo-Brazil and Miguel Penabad from Universidade da Coruña-Spain for their useful comments. They would also like to thank Julia Velásquez-Málaga from San Pablo Catholic University-Perú for her work reviewing and fixing typos.

References

[1] Aaron Cohen, Mike Woodring, and Ronald Petrusha (Editor). Win32 multithreaded programming. O'Reilly & Associates, 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns elements of reusable object-oriented software. Computing Series. Addison Wesley Professional, October 1994.

[3] K. Geiger. Inside *ODBC*. Microsoft Press, 1995.

[4] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. Java(tm) language specification. Addison-Wesley Pub Co, 2nd edition, June 2000.

[5] Donald Ervin Knuth. The art of computer programming. *Addison-Wesley, Reading, MA*, 1-3, 1973.

[6] Alexander Stepanov and Meng Lee. The standard template library. HPL 94-34, HP Labs, August 1994.

[7] B. Stroustrup. The *C++* programming language (special edition). Addison-Wesley, 2000.

[8] Andrew Troelsen. Developer workshop to com and atl 3.0. Wordware Publishing, 2000.

Contents