From Modula-2 ADT's to C++ classes

by SAM
sam@cs.vu.nl

version 1.0

With thanks to Gertjan van Wingerden

June 20, 1996

1 Introduction

The step from Modula-2 to C++ is a rather big one. One important part of this step is applying your knowledge about abstract data types learned in Modula-2 to C++. This document will offer some clues on how you can apply that knowledge to create classes in C++ that act as ADT's.

Since this document is not a tutorial of C/C++, some basic knowledge about C and C++ is assumed.

2 Separation of interface and implementation

In Modula-2 the interface is called a *definition module* and the implementation an *implementation module*. In C++ these are respectively called *header file* and *source file*. By convention, header files have the suffix `.h` and C++ source files the suffix `.cc` or `.C`[1]

3 Defining the interface

The header file should, just like in Modula-2, reveal as little as possible of the implementation. But in contrary to Modula-2, in C++ it's very hard to hide everything of the implementation, as we'll see later on.

An example of the interface of a stack as it could be defined in Modula-2:

[1] Since some filesystems (like that of MS-DOS) are case insensitive and therefore can't tell the difference between a **.c** and **.C** file, I prefer to use **.cc**.

```
DEFINITION MODULE IntStacks;

(* Elements: Although the elements could be of a variety of types, for
    simplicity we assume that they are of type INTEGER *)

(* Structure: Linear, Last in first out *)

(* Domain: The number of elements in a stack is bounded to a certain
    maximum. *)

TYPE
    IntStack;                  (* Opaque type to represent a Stack. *)

PROCEDURE Push(VAR S: IntStack; e: INTEGER);
(* pre  - S is not full.
   post - S includes e as its most recently arrived element. *)

PROCEDURE Pop(VAR S: IntStack; VAR e: INTEGER);
(* pre  - S is not empty.
   post - e is the most recently arrived element of S-pre;
          S no longer contains e. *)

PROCEDURE Empty(S: IntStack): BOOLEAN;
(* post - TRUE : S has no elements
          FALSE: S has at least one element *)

PROCEDURE Full(S: IntStack): BOOLEAN;
(* post - TRUE : S has reached its bound
          FALSE: S can at least contain one more element *)

PROCEDURE Create(VAR S: IntStack): BOOLEAN;
(* pre  - S does not exist
   post - TRUE : S has been created, S is empty
          FALSE: there was not enough memory left to create S *)

PROCEDURE Terminate(VAR S: IntStack);
(* post - S-pre does not exist. *)

END IntStacks.
```

A typical header file of the same ADT might look like

```
// intstack.h

#ifndef _INTSTACK_H_
#define _INTSTACK_H_

class intstack
{
public:
    intstack();
        // post - The stack has been created, the stack is empty
    ~intstack();
        // post - The stack-pre does not exist
```

```
    void push(int e);
        // pre  - The stack is not full.
        // post - The stack includes e as its most recently arrived element.
    void pop(int& e);
        // pre  - The stack is not empty.
        // post - e is the most recently arrived element of the stack-pre;
        //        the stack no longer contains e.
    bool empty();
        // post - true : The stack has no elements.
        //        false: The stack has at least one element.
    bool full();
        // post - true : The stack has reached its bound.
        //        false: The stack can at least contain one more element.

private:
    int* stack_;
    int  top_;
};

#endif
```

The `#ifndef`/`#define` construction is used to prohibit multiple inclusions of the header file.[2]

The interfaces in Modula-2 and C++ are very similar, but they differ on some important points. The most noticeable difference is the absence of a parameter of the stack type in all member functions. This is because functions do not work *on* an instance, but are requests *to* an instance to perform some action. Compare calling the `Full` function on a variable in Modula-2:

```
:
    IF Full(MyStack) THEN
:
```

to asking a stack-instance if it's full:

```
:
    if (mystack.full())
:
```

Another difference is the overall structure of the interface. In Modula-2, all functions are in fact seperate functions with one common parameter. In a C++ class, the functions are all member functions of the ADT and are therefore declared within the `class intstack { ... };` statement.

The third difference is the exposure of the internal variables of a class. The only thing known about the type `IntStack` in the Modula-2 interface is the fact that it exists; it's completely opaque. But in C++, all variables used in the implementation have to be declared in the interface, thereby making it virtually impossible to create an ADT that's really abstract. Making these member variables `private` will shield them from being used directly by a user.

The last difference is the names of the functions to create and destroy instances of the ADT. C++ offers constructors and destructors, so you should not give them other names like `Create` or `Terminate`.

[2] Multiple inclusions can lead to compile-errors, for example when a constant value is declared in a header-file.

4 Defining the implementation

In Modula-2, the implementation of the stack class could look like:

```
IMPLEMENTATION MODULE IntStacks;
(* IntStacks; array implementation *)

FROM  Storage IMPORT
    Available,
    Allocate, Deallocate;

CONST
    MaxSize      = 10;

TYPE
    IntStack         = POINTER TO IntStackInstance;
    IntStackInstance = RECORD
                           top  : CARDINAL [0..MaxSize];
                           stack: ARRAY [1..MaxSize] OF INTEGER;
                       END;

PROCEDURE Push(VAR S: IntStack; e: INTEGER);
BEGIN
    WITH S^ DO
        INC(top);
        stack[top] := e
    END
END Push;

PROCEDURE Pop(VAR S: IntStack; VAR e: INTEGER);
BEGIN
    WITH S^ DO
        e := stack[top];
        DEC(top)
    END
END Pop;

PROCEDURE Empty(S: IntStack): BOOLEAN;
BEGIN
    RETURN (S^.top = 0)
END Empty;

PROCEDURE Full(S: IntStack): BOOLEAN;
BEGIN
    RETURN (S^.top = MaxSize)
END Full;

PROCEDURE Create(VAR S: IntStack): BOOLEAN;
BEGIN
    IF Available(SIZE(IntStackInstance)) THEN
        Allocate(S, SIZE(IntStackInstance));
        S^.top := 0;
        RETURN TRUE
    ELSE
```

```
        RETURN FALSE
    END
END Create;

PROCEDURE Terminate(VAR S: IntStack);
BEGIN
    Deallocate(S, SIZE(IntStackInstance));
END Terminate;

END IntStacks.
```

In C++, the implementation could look like:

```
// intstack.cc

#include "intstack.h"

const int maxsize = 10;

intstack::intstack()
{
    top_ = 0;
    stack_ = new int[maxsize];
}

intstack::~intstack()
{
    top_ = 0;
    delete[] stack_;
}

void intstack::push(int element)
{
    stack_[top_] = element;
    top_++;
}

void intstack::pop(int& element)
{
    top_--;
    element = stack_[top_];
}

bool intstack::empty()
{
    return (top_ == 0);
}

bool intstack::full()
{
    return (top_ == maxsize);
}
```

The `#include "intstack.h"` is needed because a class has to be defined before the implementation can be understood by the compiler. The rest of the implementation is just C++ code.

Any program that wants to use the stack class, has to include the header file in the source and has to link in the object file of the stack implementation.

4.1 Some notes

In C++, it's possible to include source in the header file by means of inline functions. Sometimes, it's very tempting to do so; for example, it had been easier to implement the functions `empty` and `full` as inline functions. It goes without saying that this reveals no small part of the implementation and is in general bad practice.

Not all C++ compilers support boolean types, but g++ does. If you use some other compiler and are making a hush application, including the file `hush/bool.h` will provide the same boolean types as g++.

5 Examples of using ADT's

The biggest difference between Modula-2 ADT's and C++ classes is that in Modula-2, the instance of the ADT that's being processed is passed as parameter to some function, while in C++ the functions work on the member variables of the object of which the function is being called. Modula-2:

```
MODULE StackTest;

FROM SimpleIO IMPORT
    WriteInteger, WriteString, WriteLn;

FROM IntStacks IMPORT
    IntStack,
    Create, Terminate,
    Push, Pop, Empty;

VAR
    MyStack: IntStack;
    Elt: INTEGER;

BEGIN
    IF Create(MyStack) THEN
        Push(MyStack, 27);
        Push(MyStack, 3);
        Pop(MyStack, Elt);
        WriteInteger(Elt, 4);
        WriteLn;
        Pop(MyStack, Elt);
        WriteInteger(Elt, 4);
        WriteLn;
        IF Empty(MyStack) THEN
            WriteString("Stack is now empty.");
            WriteLn;
```

```
        END;
        Terminate(MyStack)
    END
END StackTest.
```

C++:

```
#include "intstack.h"
#include <iostream.h>

int main()
{
    intstack mystack;           // automatic creation
    int elt;

    mystack.push(27);
    mystack.push(3);
    mystack.pop(elt);
    cout << elt << endl;
    mystack.pop(elt);
    cout << elt << endl;
    if (mystack.empty())
        cout << "Stack is now empty." << endl;

    return 0;
                                // mystack will be automatically destroyed
}
```

6 From Modula-2 ADT to C++ class, summary

6.1 Interface

- The name of the ADT will become the name of the class.
- In all functions the parameter that represents the instance of the ADT is removed.
- All functions will become public member functions of the class.
- Functions like `Create` will become the constructor of the class.
- Functions like `Terminate` will become the destructor of the class.
- All variables internal to the ADT will become protected member variables.
- It is regarded good practice to prepend member variables with an underscore ('_') so the difference between member variables and other variables is always clear.

6.2 Implementation

- All code from functions like `Create` is moved to the constructor of the class.
- All code from functions like `Terminate` is moved to the destructor of the class.
- Code like

```
WITH S^ DO
    :
END
```

is not needed; member variables are always directly accessible.