In computer science, an abstract data type (ADT) is a mathematical model for data types. An abstract data type is defined by its behavior (semantics) from the point of view of a user, of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. This mathematical model contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

Consider the ADT: Counter. A counter is ... well, a counter; it supports the following operations:

|  |  |  |
| --- | --- | --- |
| Operation: | Preconditions: | Postconditions |
| INIT() | None | A zeroed counter exists |
| INCREMENT() | INIT | The counter is incremented by one. |
| INT SHOW() | INIT | Returns the counter's current value |

Consider, for example, the ADT: stack. A stack is a collection of items (or things on the list) that supports the following operations:

|  |  |  |
| --- | --- | --- |
| Operation: | Preconditions: | Postconditions |
| CREATE() | None | An empty stack exists |
| ISROOM() | CREATE | TRUE if there is room for at least one more item, FALSE otherwise. |
| ISEMPTY | CREATE | TRUE if there are zero items on the stack, FALSE otherwise. |
| PUSH(ITEM i) | CREATE AND ISROOM | i is on the stack |
| itemType POP() | CREATE AND !ISEMPTY | Returns the item that has been on the stack the least time. (Last in, first out)  It is an error to POP an empty stack. |
| Pseudo-operation: print() | CREATE | Simply prints the stack for diagnostic purposes. |

Also, consider the ADT: list. A list is a collection of items (or things on the list) that supports the following operations:

|  |  |  |
| --- | --- | --- |
| Operation: | Preconditions: | Postconditions |
| CREATE() | None | An empty list exists |
| ISROOM() | CREATE | TRUE if there is room for at least one more item, FALSE otherwise. |
| PUT(ITEM i) | CREATE AND ISROOM | ISTHERE(i) will return TRUE |
| REMOVE(ITEM i) | CREATE | ISTHERE(i) will return FALSE |
| ISTHERE(ITEM i) | CREATE | If PUT(i) has executed and not REMOVE(i), then returns TRUE, FALSE otherwise |
| Pseudo-operation: print() | CREATE | Simply prints the list for diagnostic purposes. |

Some considerations: will our list maintain duplicate items? Notice that the preconditions and postconditions don't tell us how to handle that. It is not a precondition to PUT(i) for i not to be already on the list; the postcondition says that after PUT(i), i is on the list.

Similarly, in this world, REMOVE(i) does not require that i be on the list before the fact; afterward, it isn't.

Notice also that return types are not specified for PUT and REMOVE.

What is an "*item*"? We must know something of its properties.

An "*item*" is an object upon which there are required methods defined:

An *item* must have a constructor and a copy constructor defined. It must also have assignment, as well as the six relational operators ('>', '<', '==', and their opposites) defined. For our convenience, we will also require that the insertion operator ('<<') be defined; this last just makes our lives easier as we study ADTs and how they behave.

Here is the header file for an *item*:

// itemType.h

using namespace std;

struct itemType

{

<payLoadType> payLoad; // payLoad may be any known type

itemType(); // constructor

itemType(const itemType &); // copy constructor

void operator =(itemType &); // assignment

void operator =(payLoadType); // optional user-defined

bool operator ==(itemType &);

bool operator >(itemType &);

bool operator <(itemType &);

bool operator !=(itemType &);

bool operator <=(itemType &);

bool operator >=(itemType &);

friend ostream &operator << (ostream &, const itemType &);

};

It's a struct; I will use struct when the object will not have private members.

One of the fields is named "payLoad"; this can be \*anything\*; it may span multiple fields or be an object.

Optionally, I have chosen to define an assignment operator that assigns a string directly to the payload. The constructors and eight operators are required; others may be included at your discretion.

Here is the header file for listType:

//listType.h

using namespace std;

template <class elementType>

class listType

{

private:

static const int MAX=10;

int top, envVar; // discuss the uses of the environmental variable: envVar

elementType\* myList;

public:

listType();

~listType();

void create();

bool isRoom();

void put(elementType &);

bool isThere(elementType &);

void remove(elementType &);

void print(); // diagnostic

};

There are five required methods. put() and remove() may return a Boolean status at the programmer's discretion. It has a private pointer variable; therefore, it should have a destructor implemented.

Notice that this is a class template. This means that, at compile-time, it will receive the object *elementType*, at the moment, it's left undefined and promised later.

This has consequences: in our usual C++ programming paradigm, we would normally have a header file (.h) and an implementation file (.cpp); the .cpp file *#include*s the header file. The application using the code defined thus would *#include* only the header file.

With a class template, this approach won't work. The application must *#include* the .cpp file, which defeats the purpose of separating the headers from their implementation; therefore, in a template file, it is acceptable (at the discretion of the programmer) to have the headers and the implementation in the same .cpp file.

Thus, the application (which I will call *driver*) would look like:

// *driver.cpp*

#include "itemType.h"

#include "listType.cpp" // *notice that it's a .cpp file for templates.*

using namespace std;

int main()

{

listType<itemType> list;

*// yaba*

*// yaba*

*// just go wild!*

return 0;

}

Before examining the code in detail, we must understand and strictly observe the levels of abstraction in this schema: the code in *driver* may reference the members of *itemType* as friend. I didn't write the example this way; however, the two are intimately connected. The class *listType* is completely isolated from *itemType*; it is profoundly ignorant of its existence until compile time when a list is instantiated thus:

listType<itemType> myList;

At this point, *elementType* in the implementation of *listType* is replaced with *itemType*. The implementation of the list uses (or might use) assignment and the relational operators, these must be defined on *itemType*.

Examining the implementation of *listType*, we see that, operationally speaking, it's a *packed array*.

listType<itemType> list;

itemType i;

list.create();

top=0

|  |  |
| --- | --- |
|  | myList |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |

i="fee";

list.put(i);

top=1

|  |  |
| --- | --- |
|  | myList |
| 0 | fee |
| 1 |  |
| 2 |  |
| 3 |  |

i="fie";

list.put(i);

top=2

|  |  |
| --- | --- |
|  | myList |
| 0 | fee |
| 1 | fie |
| 2 |  |
| 3 |  |

i="foe";

list.put(i);

top=3

|  |  |
| --- | --- |
|  | myList |
| 0 | fee |
| 1 | fie |
| 2 | foe |
| 3 |  |

i="fum";

list.put(i);

top=4 (MAX)

|  |  |
| --- | --- |
|  | myList |
| 0 | fee |
| 1 | fie |
| 2 | foe |
| 3 | fum |

(The list is full; list.isRoom() would return FALSE.)

i="fie";

list.remove(i);

top=3

|  |  |
| --- | --- |
|  | myList |
| 0 | fee |
| 1 | fum |
| 2 | foe |
| 3 | fum |

In the last example, note that "fum" has been copied to myList[1] and that *top* has decremented meaning that a new item would be copied into myList[3]. (This implementation does not keep the list sorted! ... that will come later.)