# Abstract Data Types

## Standard Abstract Types

An **abstract data type** (ADT) is a collection of data objects, together with a set of operations that act on the data objects. The operations should be described without reference to the way the data objects are represented in memory.

Most languages provide a variety of built-in abstract data types. For example, the C++ programming language provides char, int, float, double, etc. Code that is based on an abstract data type is more portable than code that is not. For example, if a programmer writes code that is based on the fact that a machine has 16-bit integers, the code will require modification before it can be used on a machine that has 32-bit integers.

## Strings

C++ has elegant and powerful facilities for creating new abstract data types. As an example, we'll consider the creation of a C++ "string" data type.

First, recall that a string is a sequence of characters which is thought of as a single object. String literals such as "hello there!", are usually enclosed in quotes and thought of as a single object. Some languages, such as BASIC, provide strings as a built-in ADT. Almost all languages provide some sort of support for string constants, since they are so useful in output statements. For example, C++ allows programmers to use string literals in output statements and in a few other places. However, C++ does not provide these strings as a built-in abstract data type. Rather, a string is viewed by the compiler as a "pointer" that holds the address of the first character of the string. Note also, that C++ programmers know the internal details about how strings are stored. In fact, the code for many standard C library routines is based on this knowledge.

## Implementing a New ADT

One of C++ most powerful features is its ability to implement new abstract data types. The rest of this handout will illustrate this by discussing the implementation of the abstract data type "*String*". If this is done well, then the *String* type will be almost as easy to use as the types that are built into the language, such as char or float.

In any language, to **implement an abstract data type**, one must

1) Decide on the internal representation of the data type, i.e. using data types that already exist, create an appropriate type definition for the new data type.

2) Write the necessary member functions and overload the operators to support the data type.

The decision in 1) is usually done after considering how it would affect execution speed and storage efficiency.

To aid our decision making process when thinking about 1), let's assume that we plan to use our strings in situations where we will frequently access the length of strings, and that these strings can be up to 255 characters in length. This implies that we do not want to have to search for the null terminator.

One possibility is to choose an internal representation for strings that uses the C++ struct facility. For example, we might choose the following as the internal representation of strings that can be up to 255 characters long.

```
struct String
{
  char           Chr[255];
  unsigned char  Len;       // holds small int between 0 and 255
};
```

The idea here is that we can think of a string as a two part entity: an array *Chr* that holds the actual characters, and an int *Len*, that holds the length or number of characters in the string. The tail end of the array would be regarded as undefined garbage. We won't need a null terminator to mark the end of the string, since with this representation, we have access to the string length at all times.

Choosing the struct internal representation above is a good first try, but it does not hide the details of our implementation. If we hide the details of the implementation, then we can have complete control over how programmers access Strings. In particular, we might not want programmers to do things like:

```
S.Chr[0] = 'A';  or S.Len = 0;
```

for if we change or implementation, such code will be broken, i.e. not compile.

Fortunately, C++ has mechanisms which easily allow us to hide certain details of our implementation. In fact, even the struct mechanism of C++ can do the job. However, the preferred way is to use the *class* facility of C++. We can simply replace "struct" with "class" in the type definition above and use the following:

```
class String
{
  char           Chr[255];
  unsigned char  Len;       // holds small int between 0 and 255
};
```

Recall that the fields of a class are **private** by default. This means that those who use our string class cannot directly access the fields "*Chr*" and "*Len*". (Clearly, however, we'll need to provide clients some way to change the contents of String variables.)

One additional touch involves the elimination of the magic number 255. It will appear often in the code that implements Strings, so it should have a name. If we do this we'll have the following internal representation of String.

```
class String
{
  enum           { MAX_LENGTH = 255 };
  char           Chr[MAX_LENGTH];
  unsigned char  Len;         // holds small int between 0 and 255
};
```

## Providing String Operations

That takes care of step 1), but there is a great deal of work to be done in dealing with step 2), i.e. providing operations, or "support functions" for the class. For example, we would certainly want to write a function that can be used to access the length of a string. We also want to write the functions and operators required to make Strings "first-class" citizens. In particular, we would like to be able to:

1) Allow strings to be assigned values, as in

```
S = "The quick brown fox"
```

2) Allow strings to be initialized and declared at the same time, as in

```
String S = "Hello there!";
```

3) Provide "natural" operator support for our string type, such as

```
"=="  for comparisons
"+" for string concatenation
"<" for less than
```

4) Have an output mechanism that works like that provided for built-in types, such as

```
cout << S;
```
or
```
cout << S1 << S2;
```

5) Provide automatic type conversions, such as

```
S = S + "AB";
S = S + 'C';
```

6) Have an input mechanism that works like that provide for built-in types, such as

```
cin >> S;
```