The OMAR Programming Language

Reference Manual & Programming Guide

Revision: September 1999

Hypercosm, Inc.
(608) 821-0500
www.hypercosm.com

For technical support, please contact Hypercosm at the phone number above or send email to support@hypercosm.com.

To see support options, frequently asked questions, and information on the email discussion group, go to http://www.hypercosm.com/support/index.html.

Contents

CHAPTER 1 Introduction

This manual was written as a guide for users who have at least some experience with other programming languages. It describes the general language features of OMAR, but does not describe Hypercosm's graphics extensions to the language. Those are described in *The Hypercosm 3D Graphics System Guide.*

Readers interested in using OMAR for 3D graphics development should read both manuals in conjunction. *The Hypercosm 3D Graphics System Guide* can get you started in creating Hypercosm graphics, but you cannot make full use of Hypercosm's capabilities without first gaining the basic understanding of OMAR that this manual provides.

Reasons for OMAR

There are perhaps hundreds of different programming languages, so it might seem like the last thing that the world needs is yet another one. If you are already a programmer, a new programming language just means another syntax to get used to and even worse, another set of semantic rules to confuse with the rules you already know.

There are, however, a number of very good reasons for OMAR. It was developed because there simply are no other languages which have the features of safety, architecture independence, generality, and expressiveness that are required for today's applications.

You may say that people are getting along just fine so far with C and C++, the standard programming languages these days. Yes, it is theoretically possible to write bug-free, understandable code in C, but in practice, code written in C is almost always bug-ridden, difficult to read and understand, and almost impossible to share and reuse.

The more recently developed Java programming language, unlike C, generally does provide the safety and architecture independence that OMAR aims to provide. In fact, there are many similarities between OMAR and Java, but not because one is derived from the other. It is more a case of parallel evolution.

OMAR was derived from a programming language called SMPL, which was created for the purpose of controlling a 3D graphics modeling, rendering, and animation system (hence, the current name, 'OMAR': **O**bject-oriented **M**odeling **A**nd **R**endering). Java, originally Oak, was created to control consumer appliances such as set top boxes and so forth.

What the designers of Java were in need of was a safe, platform independent language in order to program the higher level functionality of a particular system. When they found that no existing language was really appropriate for the task, it became necessary to create a new language. The designers of OMAR, faced with a similar challenge, came to the same conclusion.

OMAR, now developed far beyond the scope of its special-purpose beginnings, aims to take programming to a level beyond Java. This evolutionary process is nowhere near complete. You will find that many of the things that can be described in Java are difficult to describe in OMAR, but there are many things you *can* describe in OMAR that are still difficult or even impossible to describe in Java.

The OMAR Design Philosophy

The design philosophy that guided the development of OMAR can be summarized by just a few words: OMAR aims to be safe, architecture-independent, general-purpose, and expressive.

Safety

Safety was probably the number one priority in the development of OMAR. As software systems become more and more complex, and as computer usage becomes more and more pervasive in society, writing safe code becomes more and more important. A lower-level language such as C allows a programmer great flexibility, but such flexibility can easily result in poor memory management and obnoxiously cryptic bugs, even for experienced programmers.

We wanted a language that was powerful, yet we wanted to avoid at all costs features that were unsafe and might cause the software to crash or behave erratically. Although no programming language could ever make it impossible to write unsafe code, OMAR at least makes it harder.

Architecture Independence

OMAR was originally intended to describe 3D graphics and animation, something that exists outside of the realm of any particular type of computer or operating system. Obviously, there is no such thing as a Windows sphere or a Unix cube. So naturally, we sought to have a programming system that would allow a program to run more or less identically on any system so long as that system supported our programming and graphics environment.

There are many added benefits to an architecture-independent language. Many software projects are so complex that they may span the life cycles of several hardware platforms and operating systems. Porting a complex application from one platform to another can be a nightmare in complexity if too many assumptions about the underlying system are made.

During the recent explosion of internet usage, architecture independence has taken on even more importance. Computer users across the world are viewing web pages on every imaginable type of hardware and operating system. Writing a single web applet that can run on any of the major computer systems requires a programming language that can disregard the particulars of any one system. This is a major reason why Java has recently become so widely used in such a short period of time.

We generally believe that software should be written from the highest possible level of abstraction with as few assumptions as possible about the system that it is to run upon. The primitives that are made available to the programmer should be general enough to be supported by many different types of hardware and operating systems so that the application programmer can concern himself with the application itself instead of the nuts and bolts of a particular operating system.

Generality

Designing a programming language, like many engineering projects, requires balancing two opposing priorities. In our case, we must balance the need for safety with the quest for speed. OMAR, like Java, occupies the middle ground between high-performance, but error-prone languages such as C and fault-tolerant, high-level languages such as Visual Basic or Perl. Wherever possible, we chose to have error checking take place at compile-time, before the program is loaded and run, instead of at run-time for the speediest possible execution. To be completely safe, however, some extra run-time error checking must be done, so a program written in OMAR will never be quite as fast as one written in C. Work is being done, however, to create safe, high-level language constructs that can help offset the extra cost, so eventually the speed issue may be less of a factor.

Because OMAR was originally intended to be used for a 3D graphics system, it could have been designed as a special-purpose language that restricted itself so much that there was no need to worry about the user writing dangerous code. This is what has been done in the past (for example, Pixar's Renderman Shading Language) and continues to be done today (for example, VRML) for many applications where the existing general-purpose languages lack the flexibility to succinctly describe the situation or allow the programmer too much flexibility to go astray. What is needed is not more and more special-purpose languages but safer, more descriptive general-purpose languages.

Expressiveness

Whenever you translate your ideas into a computer program, you are forced to think in terms of the computer language. The language itself determines what is easy and what is difficult to think about. The same thing is often said about human languages. If no words exist to describe an idea that you have, you're stuck with a vague, unsettled feeling that wants to be an idea but can't.

Numerous efforts have been made to make OMAR as expressive as possible. Features such as OMAR's smart arrays and optional parameters don't let you do things that you couldn't otherwise do in C, but they do let you do things with more clarity and elegance. This level of expressiveness translates into less code which translates into a more productive programming environment.

Unfortunately, in the field of computers, the issue of aesthetics is usually treated as something of secondary importance. This is foolish because it is easy to build programs which are far more difficult and costly to fix and maintain than they were initially to write. The designers of Ada, the programming language created for the military knew this when they said that it is more important for a programming language to be *readable* than it is for the programming language to be *writable*. Since the programming language is the base from which all software is expressed, it makes sense to begin with a solid foundation. OMAR attempts to give you as much power and flexibility as possible to translate your ideas safely into code.

CHAPTER 2

Comparing OMAR to Java

If you are already familiar with one or more programming languages, then learning OMAR should be relatively easy for you. If you already know Java, then it ought to be especially easy.

This chapter outlines the many similarities between OMAR and Java, and notes the few major differences. If you are not already familiar with Java, then you may want to skip ahead to the next chapter.

Similarities to Java

OMAR is probably more similar to Java than to any other language. Although they may look different because of cosmetic, syntactical differences, beneath the surface you will find that they share many features and capabilities in common.

Object-Oriented Features

Both Java and OMAR have similar mechanisms for defining and using classes. OMAR borrows the concept of interfaces from Java instead of employing multiple inheritance like C++.

Memory Management

Like Java, OMAR provides automatic garbage collection, so you never have to worry about memory leaks or bad pointers, which are the most common cause of program crashes. In addition, since all user-defined objects are heap-allocated, the pointer dereferencing is implicit, as in Java, instead of being explicit as in C or Pascal.

Smart Arrays

In OMAR, arrays know their bounds as they do in Java. OMAR arrays are even more powerful, however, because you can create true square multidimensional arrays and because the array indices don't always need to begin at 0 as in Java.

Primitive Data Types

OMAR employs about the same set of primitive data types as Java with two main exceptions: OMAR uses the keyword `scalar` in place of `float`, and OMAR does not implement a pseudo-primitive string type like Java's. OMAR also introduces two new primitive data types, `complex` and `vector`, which are described in the chapter titled **Variables & Data Types**.

Differences from Java

Besides the obvious syntactical differences, there are just a few major differences between OMAR and Java that might cause some confusion.

OMAR Forward Declarations

The OMAR compiler is designed to process the code in one pass instead of the two passes used by the Java compiler. This makes it necessary for declarations to always precede their use, instead of occurring anywhere in the body of code as in Java.

In addition, circular declarations must be broken by preceding the later declaration by a forward declaration, as in C. The reason for this design decision is that it forces the programmer to declare things before they are used, so when you are searching through the code for a declaration, you always know that you must only look backwards towards the beginning, instead of having to search through the entire body of code.

OMAR Implicit Object & Array Allocation

In Java, whenever an object is declared, it is assumed to be null until it gets explicitly allocated with a new statement. In OMAR, the object is implicitly allocated, as with the primitive types, unless explicitly initialized to null.

OMAR Class Interfaces

Class declarations in OMAR are broken up into two parts, an interface part, which lists exported methods and members, and an implementation part, which lists private methods, method implementations, and private members. In Java, these are all grouped together into one, so you must search through the class declaration to find the exported parts.

Reference or Var Parameters

OMAR allows what C++ calls reference parameters and Pascal calls var parameters, and the implementation is less cumbersome and obscure than in Java.

Global Variables

In Java, everything that you declare must belong in a class declaration. Not so in OMAR, where, if you like, you can have free-standing variable, method, or type declarations. This was done because the extra structure imposed by Java, while often helpful in larger programs, is a hindrance to getting the job done in smaller programs.

No Threads or Exception Handling

OMAR implements neither exception handling nor threads. Rather than being a design decision, these features simply haven't yet been implemented and may be incorporated into a future version.

CHAPTER 3

OMAR Elements & Structure

This chapter outlines the essentials of the OMAR programming language. It introduces the code elements that make up an OMAR file, and describes how OMAR files are generally structured.

OMAR Vocabulary

The text of a computer program is made up of a variety of different components that have different functions. The text of an OMAR program can be broken down into reserved words, identifiers, and special symbols.

Reserved Words

Programming languages rely on a number of words having predefined, unchangeable meanings. These words, called *reserved words*, cannot be used as identifiers when you write OMAR programs. Because OMAR is meant to be both more expressive and more human-readable than other popular programming languages, it also makes use of many more reserved words. The following table lists OMAR's reserved words.

Table 3-1: OMAR Reserved Words

abstract	**double**	**integer**	**objective**	**shape**
and	**each**	**interface**	**or**	**short**
anim	**else**	**include**	**parallel**	**some**
answer	**elseif**	**is**	**perpendicular**	**static**
boolean	**end**	**isn't**	**picture**	**struct**
break	**enum**	**its**	**private**	**subject**
byte	**exit**	**itself**	**protected**	**then**
char	**extends**	**long**	**public**	**true**
complex	**false**	**loop**	**question**	**type**
const	**final**	**max**	**read**	**vector**
continue	**for**	**min**	**redim**	**verb**
cross	**free**	**mod**	**reference**	**when**
dim	**global**	**native**	**refers to**	**while**
div	**has**	**new**	**renew**	**write**
do	**if**	**none**	**return**	**with**
does	**implements**	**not**	**scalar**	
dot	**in**	**num**	**shader**	

Identifiers

When you create new variables or define new data types, they must be given names to identify them in a unique way. Such user-defined names are called *identifiers*.

Here are the rules for creating a new identifier:

- It must not be a reserved word.
- It must begin with a letter and may be followed by letters, number, or the underscore character: _.

The maximum allowable length of identifiers may vary from system to system, but should be around 256 characters, plenty for most names.

Table 3-2: The OMAR Alphabet

'a'..'z'	the lowercase letters
'A'..'Z'	the uppercase letters
'0'..'9'	the digits
'_'	the underscore character

Example: Valid Identifiers

name	**number_of_widgets**	**George_Washington**
counter1	**a**	**not_yet_found**

Example: Invalid Identifiers

Identifier	Reason Why Identifier Is Invalid
24_bit_mode	identifiers may not begin with a number
integer	identifier is a reserved word
why?	identifiers may contain only letters, numbers, and underscore characters
bit-mode	same reason as above

Special Symbols

In various places in the language, special symbols are required. These symbols may not be used in any other parts of the program where they are not specifically required and may not be used as parts of identifiers.

Table 3-3: The OMAR Symbols

(	)	[	]	;	:	<	>	+	"
*	/	=	^	{	}	..	.	-	

The semi-colon is perhaps the most important special symbol. In OMAR, as in C and Java, every instruction must end with a semi-colon.

Case Sensitivity

Identifiers and reserved words are *not* case sensitive, which means that capital letters and small letters are not considered to be different. This means, for example, that the identifier, goo, is the same as the identifier GOO, Goo or GoO.

Free Format

OMAR uses free-format layout. This means that the amount of white space between the keywords and identifiers makes no difference in the meaning of the program. For example, the following two code snippets use different formatting, but have the exact same meaning and will produce the same results:

Example: Free Format

```
while (counter < 10) do
      counter = itself + 1;
end;
```

```
while (counter < 10) do counter = itself + 1; end;
```

OMAR File Types

There are two different OMAR file types:

- **OMAR source files** are runnable files that contain essential instructions for producing OMAR programs. Source file names should carry the extension **.omar**.
- **OMAR resource files** are not runnable, but contain ‘pieces’ of OMAR code that may be imported for use in source files. Resource file names carry the extension **.ores**.

OMAR Source File Structure

There are five major components to an OMAR source file:

- The program header
- Declarations
- Statements
- Include directives
- Comments

The Program Header

At the beginning of OMAR source files, a program header must appear that indicates where the computer should start running the program. The header is composed of the keyword `do` followed by the name of the first procedure[1] the computer should run. If you want a program to begin with a procedure named `big_task`, you must begin a file with the line, `do big_task;`

C and Java do not use a header but instead rely on the convention that the first procedure run is named `main`. OMAR, on the other hand, allows you to name your procedures whatever you’d like, but requires that you include a program header to indicate which procedure to run first.

Only one procedure may be named in the header. Note that there may be many procedures declared in the code that are not listed in the header. These procedures can still be executed by being called from another procedure. All procedures that are executed are called either directly or indirectly from the procedure listed in the header.

Note: The major difference between source files and resource files is that resource files need not contain a header, and therefore cannot be compiled and run by themselves. Otherwise, resource files have the same structure as source files.

1.Procedures are independent sections of code. In other languages and contexts, procedures are also known as functions, routines, subprograms, and methods. Procedures will be discussed more in later chapters.

Declarations

Following the program header is a list of declarations. The declarations compose the body of the program. Declarations fall into three general categories:

- Data declarations
- Type declarations
- Procedure declarations.

Example: A Program Header & Declarations

```
do task2;                          // Program header: task2 will be run first.

integer a;                         // A data declaration
enum answer is yes, no, maybe;     // A type declaration

verb task1 is                      // A procedure declaration
      {declarations}
      {statements}
end;

verb task2 is
      {declarations}
      {statements}
      task3;                       // task3 is called here in task2.
end;

verb task3 is
      {declarations}
      {statements}
      task1;                       // task1 is called here in task3.
end;
```

Data Declarations

Data that is used by the program is placed in storage locations called variables. During the execution of the program, the actual values that are contained in the variables may change, but the type of the data will remain the same.

For example, an integer variable might at some time have a value of 5. Later on, it might be equal to 10, but it will never contain values like 0.001, "A" or false because these values are not considered to be integers. This is because the storage boxes needed to hold some kinds of data are bigger than the ones that are required for other kinds of data.

All data that is used in OMAR programs must be explicitly declared before it can be used. If you find that you need a variable to store a value, you must first tell the computer what type the variable is and what name the variable has. This is what is known as its declaration. In addition, you may be able to specify attributes about how the variable is stored, such as whether or not it can be changed.

When you declare a variable, you do not have to specify its value. If you do not, then its value is undefined until you explicitly set the value of the variable. This is usually done with the assignment statement but you can also initialize the variable when you declare it.

The different data types that appear below will be described more fully in a later chapter.

Example: Data Declarations

```
integer counter;
scalar temperature = 32.0;
const scalar pi = 3.1415936;
boolean done is false;
scalar x, y, z;                                          // Here, three variables are declared at once.
vector red =<1 0 0>, green = <0 1 0>, blue =<0 0 1>;   // Here, three variables are declared
                                                         // and initialized.
complex i = <0 1>;
char quit is "q";
string type name is "Bob";                               // Here, string is a user-defined type.
```

Type Declarations

Type declarations are used to define a new type of data when the built-in data types are not sufficient. Type declarations begin with the keywords enum, struct, or subject. These will be covered in more detail later.

Example: Type Declarations

```
enum material is straw, wood, brick;     // material is a new data type that can have values of
                                         // straw, wood, or brick.

struct person has                        // person is a new data type composed of an integer and an
                                         // array of type char.
      integer age;
      char name[];
end;
```

Procedure Declarations

Procedure declarations make up the body of the programming language because all programming instructions (statements) must reside inside of procedures.

In OMAR, there are two basic types of procedures:

- *Verbs* specify some sequence of actions that the computer should do.
- *Questions* are much like verbs except that they also return a value to the procedure that called them.

When using OMAR for 3D graphics, four more types of procedures are used: *shapes*, *pictures*, *anims*, and *shaders*.

Statements

Statements are the part of a program that actually tells the computer what to *do* with data. There are many different kinds of statements—assignment statements, if statements, looping statements, read and write statements, and procedure calls, for example. All of these will be described in more detail later. However, there are some important general rules that apply to all statements:

- *All statements must reside inside of procedure declarations.* Declarations, on the other hand, can appear in any part of a file after the header and after any include directives.
- *In any given block of code, all statements must come* after *all declarations.* The tricky part of this rule is in understanding exactly what a 'block of code' is. This concept will be made clearer later.

The Include Directive

In order to enable greater modularity, OMAR files have the ability to import code from other OMAR files. Each source file has its own header line at the top, which tells which of the procedures to run if that file is to be executed. When a file is included, its header line (if it has one) is ignored. Also, the language keeps track of which files have been included so far, so if a file gets included more than once, then the second include is ignored.

Example: The Include Directive

```
do test;

include "hashtables.ores";           // Includes a resource file that contains hashtable definitions.
include "thing.omar";                // Includes a source file that contains the definition of a thing type.

verb test is
        hashtable type table;        // A data declaration: creates a hashtable called table.
        thing type thing1;           // A data declaration: creates a thing called thing1.

        table add thing1 as "fred";  // A statement.
end;
```

Comments

Comments are sections of text in a file that are ignored by the computer. It is a very good idea to use comments to describe exactly what your code is doing, and to label different sections of code so that you can recognize those sections more easily. Comments can also be used to temporarily disable a piece of code without permanently deleting it from the file. This is called *commenting out* your code.

OMAR supports two types of comments: line comments and block comments.

Line comments

The first type of comment is called a *line comment* because it spans only one line in the program. Line comments are indicated by two forward slashes, //. When the compiler encounters this symbol, it ignores all text until the end of the line, so you can include any kind of text or symbols on the line as a comment.

Example: Line Comments

```
// This is a comment
integer a;  // This is a comment following a code declaration
```

Block comments

The other, more powerful form of comment is the *block comment*. Block comments may span multiple lines and may even include other line or block comments. Block comments are formed by enclosing the commented text by a pair of curly braces. Generally, block comments should be used for commenting out blocks of code and not in cases where line comments may be used instead.

Example: Block Comments

```
{ This is a block comment. }

{
This is also a block comment.
}

{
This shows how block comments can enclose other {block comments}
and also // line comments.
}
```

CHAPTER 4

Variables & Data Types

Variables

Variables are used to store data that is used by a computer program. They can represent such things as color, temperature, positions of objects—almost anything that can be measured quantitatively or represented symbolically. Once data is stored in variables, it can be retrieved, examined, and changed.

Keep in mind these rules about variables:

- Each variable that you use must be *declared* before you can use it. A variable declaration is where a variable is 'born' and given its name and type.
- The names given to variables must be unique so that there is no confusion as to which variable you are referring to.
- Variables may be given initial values, but if no initial value is given, variables remain undefined until they are given a value at some point in the future.

To declare a new variable, use the following format:

Figure 4-1: Variable Declaration Syntax

```
<data type name>  <variable name>  <optional initializer> ;
```

Example: Variable Declarations

```
integer counter;
double pi = 3.14159265358979323846;
scalar speed_of_light = 3  * (10^8);
vector location = <5 0 100>;
```

Data Types

Each variable is said to be of a certain *data type*. The type determines what kind of data the variable can store. Once the variable is created, it can never change its type.

The OMAR language has a predefined set of basic, or *primitive*, data types. (More complex data types can be defined by the user as discussed later.) A more detailed description of the properties of each of these data types is given in the following sections.

Table 4-1: The OMAR Primitive Data Types

Name	Contents	Size	Min Value	Max Value
boolean	True Or False	1 Bit	N.A.	N.A.
char	Unicode Character	16 Bits/2 Bytes	Chr(0)	Chr(32767)
byte	Signed Integer	8 Bits/1 Byte	-128	127
short	Signed Integer	16 Bits/2 Bytes	-32768	32767
integer	Signed Integer	32 Bits/4 Bytes	-2.147 Billion	2.147 Billion
long	Signed Integer	64 Bits/8 Bytes	-9.223 Quintillion	9.223 Quintillion
scalar	Single Precision FP	32 Bits/4 Bytes	+/- 1.402 E -45	+/- 3.40 E 38
double	Double Precision FP	64 Bits/8 Bytes	+/- 4.94 E -324	+/- 1.79 E 308
complex	A Pair Of Scalars	64 Bits/8 Bytes	Same As Scalar	Same As Scalar
vector	A Triplet Of Scalars	96 Bits/ 12 Bytes	Same As Scalar	Same As Scalar

Note that there is no predefined string type as there is in Java. In OMAR, strings are implemented as arrays of characters, as they are in C. Otherwise, OMAR's set of primitives is very similar to Java's.

Boolean

A boolean value is either true or false. These values are useful in describing such things as whether something is on or off, done or not done, or in any other situation where there are only two possible states. The constants, true and false, are predefined and represent the two possible boolean values.

Two boolean values may be combined using boolean operators to yield a boolean result. For example, one boolean variable, named bool1, is set to true. Another boolean variable, named bool2, is set to false. The expression (bool1 **and** bool2) evaluates to false because bool1 and bool2 are not both true. The expression (bool1 **or** bool2) evaluates to true because at least one of the two boolean values is true. The expression (**not** bool1) evaluates to false because bool1 is true, and not bool1 is therefore false. If an expression involving both and and or is evaluated, then the and operator takes precedence over the or operator. For example, the expression, (a **or** b **and** c) is evaluated as (a **or** (b **and** c)).

Table 4-2: Boolean Operators

Operator	Purpose
and	Logical And
or	Logical Or
not	Logical Not

When a boolean variable is assigned a value, the keyword **is** is used instead of the standard assignment operator, **=**. This feature of OMAR is one of many that are meant to make OMAR a more readable language.

Variables of all primitive types may be compared with other variables of the same type for equality or inequality. Integer and scalar operators may also be compared using the greater than or less than operators to yield boolean values.

Table 4-3: Relational Operators

Operator	Purpose
=	Equality
<>	Inequality
>	Greater Than
<	Less Than
>=	Greater Than Or Equal
<=	Less Than Or Equal
is	Equality (Between Non-Numerical Types)
isn't	Inequality (Between Non-Numerical Types)

Example: Using the Boolean Data Type

```
boolean done is false;
boolean condition3 is condition1 and condition2;
boolean overflow is (number > limit)
boolean error is (number = 0) or not done;
boolean condition is (char1 is "a") and (char2 isn't "b");
```

Char

A char is a variable that can represent any element of the character set present on a computer keyboard. Examples of chars are the letters of the alphabet *a* through *z*, the capital letters of the alphabet *A* through *Z*, the characters representing the digits *0* through *9*, and special symbols such as the period, '.', or the semicolon, ';'. In addition, a char may also be a special non-printable character which has some special meaning to the computer, such as a tab, space, or carriage return. To denote a particular character, place the character within double quotes.

Every character is represented as a number in the computer, which means that each character has a unique matching number code. The functions below can be used to convert between a char and its integer number code. To use them, you must include the resource file **math.ores** in your OMAR file.

Table 4-4: Char-Integer Conversion Functions

Function	Purpose
chr X	**Returns the character with the integer character code of X**
ord X	**Returns the integer character code for a particular character, X**

The char type, like the boolean type, uses the **is** operator for assignment instead of using **=**.

Example: Using the Char Data Type

```
include "math.ores";                  // include this file in order to use the chr function

char ch is "A";
char space is chr 32;
char name[] = "Fred Freugelbugger";   // an array of chars
```

Strings

Often you need a way of creating a variable to hold a list, or *string*, of characters. In OMAR, strings are implemented as char arrays (arrays will be covered in greater detail in later sections.)

It is possible define a type string to stand for an array of char so you don't have to use array-indicating brackets each time you want to want to use the type. This is actually already done for you in the file **strings.ores**, which you may include in your files when you want to use strings. To learn more about how strings work, you should first learn how arrays work, and learn a little about OMAR's object-oriented features. Then you can look at **strings.ores** itself to see how strings are implemented there.

Even if you don't completely understand string implementation, you should still find strings easy and helpful to use. To assign a string as a unit, use the is operator. See the chapter on arrays for a more detailed discussion of array operators and issues. The example file below, which you can run yourself, shows how strings can be used.

*Listing 4-1: Using the String Type in **strings.ores***

```
do write_strings;

include "strings.ores";            // You must include this to use OMAR's standard string type.

verb write_strings is
      string type name is "Larry"; // Because the string type is a user-defined type (defined in the file
                                   // "strings.ores") and not a primitive type, the keyword type must
                                   // be used when you use the string type.

      write name, ;                // This statement writes out "Larry".
      name[1] is "H";              // This changes the first character in name to "H". This works because
                                   // strings work just like arrays.
      write name, ;                // Writes out "Harry".
end;        // compare_strings
```

Byte, Short, Integer, & Long

Integers are the familiar counting numbers, 1, 2, 3, etc., and are useful for representing things that can have only whole number values, such as the number of characters in a file, the number of objects in a list, etc.

To store integer data in an OMAR program, you use one of the types byte, short, integer, or long. The difference between these types is not in what kind of data they store, but in their sizes. A byte, as its name implies, is only one byte in size, and can therefore only hold numbers in the range of -128 to 127. Consult the table "The OMAR Primitive Data Types" on page 18 to see what the number ranges are for the other integer data types.

The following operators can use integer operands to yield integer results.

Table 4-5: Integer Operators

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
div	Integer Division
mod	Modulo (Remainder)
- (Unary)	Negation

The operators mod and div are special integer division operators, required because integers must always be whole numbers. The div operator is the same as / except that the result of a div operation is rounded down to the nearest integer, and the mod operator gives the remainder of a division. Thus (11 div 3) results in 3, and (11 mod 3) results in 2.

There are a number of functions in the **math.ores** resource file that return integer results. To use these, you must include **math.ores** in your source file.

*Table 4-6: Integer Functions in **math.ores***

Function	Purpose
iabs X	Absolute Value of the Integer X
isqr X	Result Is the Integer X Squared
trunc X	X Is A Scalar Value, The Result Is The Whole (Integer) Part Of X
round X	X Is A Scalar Value, The Result Is The Nearest Integer. If X Is Midway Between Two Integers, Then We Return The Larger One.

Example: Using the Integer Data Types

```
integer i = 0;
short goo = -1000;
integer value = trunc 3.1415;
byte a = -value;
long b = 400000 * (isqr 144);
```

Scalar & Double

Scalars are numbers that represent a continuous range of values, like height, temperature, or radius, where there can be a theoretically infinite number of intermediate values. In the mathematical world, these are known as real numbers. Computer scientists refer to the method of representing real numbers in computers as *floating-point* arithmetic.

To store floating-point numbers in OMAR, you use either a `scalar` or `double` type. Here again, the difference between the two types is not in what kind of data they store, but in their sizes. A `scalar` is four bytes in size, and a `double` is eight bytes in size.

There are a variety of operators that can take integer and scalar operands and produce a scalar result. Since integers are a subset of scalars, integers are automatically converted to scalars where the situation warrants it.

Table 4-7: Scalar Operators

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
- (Unary)	Negation

There are also a number of functions in **math.ores** that take a scalar operand and produce a scalar result. Integers may also be used as the operands to these functions and they will automatically be converted to scalars to yield the proper result.

*Table 4-8: Scalar Functions in **math.ores***

Function	Purpose	Function	Purpose
abs X	Absolute Value of X	**sin X**	Trigonometric Sine of X
sqr X	Result Is X Squared	**cos X**	Trigonometric Cosine of X
ln X	Natural Logarithm of X	**tan X**	Trigonometric Tangent of X
log X	Log in Base 10 of X	**asin X**	Trigonometric Arcsine of X
exp X	Exponential (e^x)	**acos X**	Trigonometric Arccosine of X
sqrt X	Square Root of X	**atan X**	Trigonometric Arctangent of X
Note: Trigonometric functions use degrees, not radians			

Example: Using the Scalar & Double Data Types

```
scalar temperature = 32.0;
double infinity = 10^50;
scalar pi = 3.1415926;
scalar x = 0;
double length = sqrt 0.48;
scalar a = (-pi / 4) + length * x;
```

Complex

Complex numbers are numbers of the form: *a* + *(b* * *i)* where *i* represents the square root of -1. These numbers are used in a number of scientific applications and in computing fractal images such as the famous Mandlebrot set. The variable *a* is called the real part because it is not multiplied by *i*. The variable *b* is called the imaginary part because it is multiplied by *i*, which is not a member of the set of real numbers.

Complex numbers are a superset of scalars because if the imaginary part is equal to 0, then the complex number is the same as a scalar. If the imaginary part is not equal to 0, then special rules must be used to perform operations on these numbers. The most common operations are covered by the built-in operators listed below.

Table 4-9: Complex Operators

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
- (Unary)	Negation

Extracting the Real & Imaginary Components

In order to extract the complex and imaginary parts of an imaginary number, you must use the two functions, real and imag, which are found in the resource file **native_math.ores**. They return scalar values when passed a complex operand.

Table 4-10: Extracting The Scalar Fields of a Complex Number

Function	Purpose
real X	Returns The Real Part Of X
imag X	Returns The Imaginary Part Of X

Example: Using the Complex Data Type

```
complex i = <0 1>;
complex i_times_2 = <0 1> * 2;
```

Vector

In OMAR, a vector is simply a set of three scalars. You can represent many concepts in the real world by these triplets of three numbers. Vectors are used in mathematics, physics, and computer graphics to describe locations and directions in three-dimensional space. In addition, colors can be specified by vectors because all colors can be described as a combination of three primary colors.

Table 4-11: Vector Operators

Operator	Purpose
+	Vector Addition
-	Vector Subtraction
*	Vector Or Scalar Multiplication
/	Vector Or Scalar Division
dot	Dot Product (Scalar Product)
cross	Cross Product (Vector Product)
parallel	Parallel Component
perpendicular	Perpendicular Component
- (Unary)	Negation (Reverse Direction)
Note: Vector operations produce vectors, except for dot products, which produce a scalar. Multiplication and division can be performed on either two vectors, or on a vector and a scalar.	

Accessing the Components of a Vector

The three components of the vector are called the x, y, and z components of the vector. They can be accessed much like fields of a C or Java object, by using a period. For example, to access the x component of a vector named location, you use the syntax: location.x.

Example: Using the Vector Data Type

```
vector white = <1 1 1>;
vector location = <1 -3 2> cross -<1 2 5> dot <1 1 0>;
vector u = location * 5;
vector w = u * <.5 .3 .5>;
vector v = u parallel w;      // v is the component vector of u parallel to w,
                              //      or also, the projection of u onto w
w.y = 0.8;                    // Sets the y component of w to 0.8
location.z = location.x;      // Sets the z component of location equal to its x component
```

Constants

Constants are just like variables except that their values cannot be changed. All constants must be given an initial value, which is permanent. Constants may be declared any place that variables are declared. The format for declaring constants is similar to variables:

Figure 4-2: Constant Declaration Syntax

```
const <type name> <variable name> <mandatory initializer> ;
```

Example: Constant Declarations

```
const scalar freezing_point = 32;
const scalar pi = 3.14159265;
const scalar e = 2.718281828;
const boolean on is true, off is false;

pi = 3.14159265;            // Error - a constant may only be assigned by its initializer
```

Reference Variables

Reference variables are special variables that look and behave just like regular variables. They are different, however, because the actual memory for their data must be shared with a normal variable. A reference variable declaration is just like a normal variable declaration except that the keyword `reference` follows the data type.

References are a lot like *pointers* in C++ or Pascal, except that they have extra checking built in to guard against the kind of mistakes that pointers are notorious for causing.

In order to assign a reference to a variable, use the syntax: `a refers to b`. If you are a C programmer, you can think of this as `a = &b`. If a reference does not refer to anything at all, then it is said to refer to `none`. This is like `null` in C or Java. All reference variables are initialized to refer to `none` when they are created, and if an attempt is made to use them before they are assigned to refer to something, then a run-time error results.

Figure 4-3: Reference Variable Declaration Syntax

```
<data type name> reference <variable name> <optional initializer> ;
```

Example: Using Reference Variables

```
integer a = 0, b = 0;
integer reference c refers to none;

c refers to a;
c = 10;                 // Assigns the value 10 to c and a
c refers to b;
b = 47;                 // Assigns the value 47 to b and c

write "a = ", a, ;      // Writes out "a = 10"
write "c = " , c, ;     // Writes out "c = 47"
```

CHAPTER 5

OMAR Statements

This chapter describes OMAR statements and how to use them.

The Assignment Statement

The most basic statement is the assignment statement, which simply assigns a value to a variable. The value of the variable is specified by an *expression*. The expression may be as simple as the name of another variable of the same type, or as complex as a mathematical formula involving many different terms. In OMAR, it is considered an error to assign variables to themselves, since this effectively does nothing. The basic form of the assignment statement is as follows:

Figure 5-1: The Assignment Statement

```
<variable identifier> <assignment operator> <expression> ;
```

The Assignment Operators

Four different assignment operators are used to assign different types in OMAR:

- **=** is used to assign numerical types and the contents of structures and arrays.
- **is** is used to assign symbolic types that can only assume a limited set of values. These types include boolean, char, enumerated types, and references to structures and arrays. The difference between assigning the contents of structures and arrays and assigning references to structures and arrays will be made clear in later sections.
- **refers to** is used to assign reference variables.

- **does** is used to assign procedure interface variables (as described in the chapter on procedures.).

Table 5-1: The Assignment Operators

Data Type	Assignment Operator
boolean, char	is
short, byte, integer, long	=
scalar, double, complex, vector	=
enumerated type	is
structure, subject, array contents	=
structure, subject, array reference	is
variable reference	refers to
procedure interface variable	does

Note that the assignment operator used in the assignment statement is the same symbol that is used as the relational operator to test for equality. This is not a problem because the context determines which operation to perform. If the = sign appears in the context of a statement, then it is compiled as an assignment operator. If the = sign appears in the context of an expression, then it is compiled as an equality operator.

Assignments may be made to any type of variable provided that the type of the variable and the type of the expression are the same and that the variable is not a constant or final variable. Note that structured types such as arrays may be assigned as a unit or tested for equality as a unit using the = operator. This is similar to how it is done in C or Pascal, but different from Java, where the `clone` and `equals` methods must be invoked.

Expressions

Expressions are formed by evaluating a sequence of operators or functions and their operands to yield some value as a result. The types of the operands are checked to make sure that they match the operators and functions that are used. The functions (question procedures) may be built-in functions or user-defined functions.

The order in which the operators are applied depends upon the rules of precedence. Operators with the highest precedence will be applied first, followed by operators of lower precedence. Operators of the same precedence will be applied from left to right in the order that they appear in the expression. The precedence rules can always be overridden by adding parentheses around the expressions that are to be evaluated first.

Example: Operator Precedence in Expressions

```
2<3<4 * 5                        = (2 < 3) and (3 < (4*5))         = true
4 * 5+3 * 3                      = (4*5) + (3*3)                   = 29
sqrt 4 + 12                      = (sqrt 4) + 12                   = 14
4*2 ^ 2                          = 4 * (2^2)                       = 16
true or false and true           = true or (false and true)        = true
<0 0 1> dot <1 1 1> cross <1 0 0> =<0 0 1> dot (<1 1 1> cross <0 1 0>) = 1
```

Table 5-2: Operator Precedence

Operators Are Listed In Order From Highest Precedence (Top) To Lowest Precedence (Bottom)
- (Unary), not
^
cross, parallel, perpendicular
dot
***, /, div, mod, and**
+, -, or
=, <>, <, >, <=, >=

Compact Expressions

Occasionally, in expressions, you find that you need to test a particular value against a number of different expressions. The syntax used in this type of situation can be made more compact and readable if you remember what the left-hand side of each expression is and then when you encounter successive clauses in the expression, if you don't find a new left-hand-side expression, you assume that the previous one was intended.

For example, in English, instead of saying "are the clothes clean and are the clothes dry and are the clothes pressed or are the clothes new or are the clothes unworn", it would be easier to say "are the clothes clean and dry and pressed or new or unworn".

Example: Compact Expressions

```
integer a, b, c;
thing type thing, last;         // Note: the "thing" type is a just a dummy type used in examples.

if a > b and a < c then
end;

while thing isn't none and thing isn't last do
end;

// The above expressions can be replaced by the following:
//
if a > b and < c then
end;

while thing isn't none and isn't last do
end;
```

Pronouns: its and itself

To help make code more readable, OMAR provides two pronouns, itself and its.

- Use itself to refer to whatever the previous expression referred to. You can often use itself in place of the C++ and Java operators, +=, -=, *= and /=.
- Use its with structures (which will be introduced in a later chapter) to refer to a field of whatever structure was in the previous expression (assuming that the previous expression referred to a structure).

Example: Using OMAR Pronouns, it *and* itself

```
struct chain has
        chain type next is none;
end; // chain

integer i = 1, iterations = 0;
boolean parity is false;
chain type chain;

while some chain do
        i = itself * 2;
        iterations = itself + 1;
        parity is not itself;
        chain is its next;
end;
```

Short-Circuit Expressions

Normally, all of the terms in an expression must be evaluated in order to determine the result. If the expression involves the operators and and or, however, the value of the expression is sometimes determined by the first operand alone, so the second operand need not be evaluated.

For example, in any expression (a and b), if a is false, then the expression as a whole must be false no matter what the value of b is, so you don't need to evaluate b. In the case of an or expression, (a or b), if a is true, then the expression as a whole must be true, so you needn't evaluate b. This is called *short-circuit* evaluation.

There are two main reasons why it is useful for short-circuit evaluation to occur. The first reason is that it can make code faster. If the expressions are complex, then it will take more time to evaluate both operands than it will to test after the first operand is evaluated and potentially drop out. The second and more important reason for short-circuit evaluation is that you can write expressions where the later terms can only be evaluated without causing an error if the earlier terms check out first. You could restructure this kind of an expression using if-then statements to avoid evaluating the later terms but this makes the code less readable.

For example, the following expression could cause an error if short-circuit evaluation is not used, because if the first test fails, then the second test cannot be evaluated without causing an error.

Example: Code Requiring Short-Circuit Evaluation

```
integer a = 0, b = 1;

// If short-circuit evaluation is not used, this could cause an error because
// sqrt(a) can not be calculated when a = 0
//
if a > 0 and sqrt(a) > b then
      b = sqrt(a);
end;
```

The code in the example could be restructured by separating the two and operands of the expression into two if statements, so the second expression won't be evaluated if the earlier test fails. For example:

Example: Short-Circuit Evaluation Using Nested Ifs

```
integer a = 0, b;

if a > 0 then
      if sqrt(a) > b then
            b = sqrt(a);
      end;
end;
```

In OMAR, as in most other programming languages, short-circuit evaluation occurs by default. However, you can also choose to disable short-circuit evaluation and force all elements of an expression to be evaluated. You may need to do so when you have an expression that makes a series of procedure calls, and it is important that every procedure call is made. To disable short-circuit evaluation, use the operators `and if` and `or if` in place of `and` and `or`.

Example: Disabling Short-Circuit Evaluation

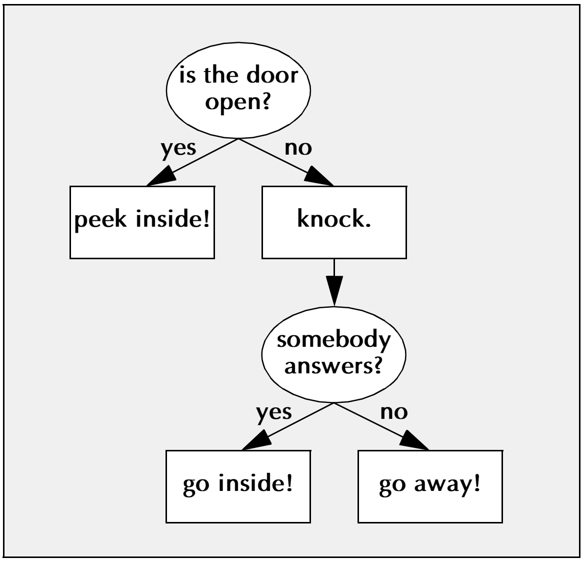
```
integer a = 0, b = 1;

if a > 0 and if a > b then
	b = sqrt(a);
end;
```

Conditional Statements

One of the most basic things that a computer can do is test for some condition and take a different course of action depending on the outcome. We human beings do this all the time. For example, if a door is not open, then you open it and go in, otherwise, you just go in.

Frequently, conditionals are nested in a complex sequence. If the door is not open, then knock; if someone answers, then go in, else go away. Otherwise, if the door is open, peek in and see if there's anyone around, etc. This set of conditionals can be represented as a decision tree (as in the figure below), with the conditions placed in ovals, and the statements placed in rectangles.

Figure 5-2: A Decision Tree



If Statements

The most basic form of conditional is the if-then statement. The if-then statement relies upon solving a boolean expression that determines whether or not a certain action is taken. If the boolean expression evaluates to true, then the statements are executed. If the boolean expression evaluates to false, then nothing happens and the computer goes on to the next statement. The basic form of the if-then statement is as follows:

Figure 5-3: The If-Then Statement

```
if <boolean expression> then
        <declarations>
        <statements>
end;
```

Example: If-Then Statement

```
scalar d = sqr b - 4 * a * c;

if d < 0 then
        write "no roots found", ;
end;
```

A more complex form of the if statement is the if-then-else statement. This statement works by deciding upon one of two possible courses of action based on the value of the boolean expression. If the boolean expression evaluates to true, then the first block of statements is executed, else, the second block of statements is executed. The form of the if-then-else statement is as follows:

Figure 5-4: The If-Then-Else Statement

```
if <boolean expression> then
        <declarations1>
        <statements1>
else
        <declarations2>
        <statements2>
end;
```

Example: The If-Then-Else Statement

```
scalar d = sqr b - 4 * a * c;

if d < 0 then
        write "no roots found", ;
else
        scalar root1 = (-b + d) / (2 * a), root2 = (-b - d) / (2 * a);
        write "roots = ", root1, ", ", root2, ;
end;
```

Another useful form of the if statement is used when you want to perform a series of tests and execute some statements as soon as you find a condition that evaluates to true, otherwise, keep testing. This statement is called the if-then-elseif statement. The form of this statement is as follows:

Figure 5-5: The If-Then-Elseif Statement

```
if <boolean expression1> then
        <declarations1>
        <statements1>
elseif <boolean expression2> then
        <declarations2>
        <statements2>
        .
        .
elseif <boolean expressionN> then
        <declarationsN>
        <statementsN>
end;
```

Note that since this statement continues testing until either a true condition is found or it reaches the end, it is a good idea to place conditions that are likely to be true at the top of the statement so they are reached first to avoid a lot of unnecessary testing.

Example: The If-Then-Elseif Statement

```
scalar d = sqr b - 4 * a * c;

if d < 0 then
        write "no roots found", ;
elseif d = 0 then
        scalar root = -b / (2 * a);
        write "root = ", root, ;
end;
```

The last possible form of the if statement is created by fitting an else clause onto the end of an if-then-elseif statement. This is useful when you want to perform a series of tests, and if none of the conditions are true, then some default set of statements should be executed. This is the if-then-elseif-else statement, which is illustrated below:

Figure 5-6: The If-Then-Elseif-Else Statement

```
if <boolean expression1> then
        <declarations1>
        <statements1>
elseif <boolean expression2> then
        <declarations2>
        <statements2>
        .
        .
elseif <boolean expressionN> then
        <declarationsN>
        <statementsN>
else
        <declarations>
        <statements>
end;
```

Example: The If-Then-Elseif-Else Statement

```
scalar d = sqr b - 4 * a * c;

if d < 0 then
        write "no roots found", ;
elseif d = 0 then
        scalar root = -b / (2 * a);
        write "root = ", root, ;
else
        scalar root1 = (-b + d) / (2 * a), root2 = (-b - d) / (2 * a);
        write "roots = ", root1, ", ", root2, ;
end;
```

The When Statement

The other form of conditional statement is the when statement. The when statement is like the if statement, but instead of using a boolean value to determine what action to take, it uses an enumerated type or char type to determine a course of action.

OMAR's when statement is very similar to C and Java's case statements, except that the syntax is a little different, and the when statement can only use enum and char types, and no integer types. Note that it doesn't make as much sense to use an integer or scalar to switch on because they (conceptually) have an infinite number of possible states. The only types which have a fixed number of states are the boolean, the enumerated type, and the char.

When statements evaluate the expression and, based on the value of the result, go to the proper case, and execute the statements. The order of the cases inside the when statement doesn't matter.

Figure 5-7: The When statement

```
when <char or enum expression> is
      <value1>:
            <declarations1>
            <statements1>
      end;
      <value2>:
            <declarations2>
            <statements2>
      end;
      .
      .
      <valueN>:
            <declarationsN>
            <statementsN>
      end;
end;
```

Example: The When Statement

```
enum situation is fire, flood, tornado;
situation type situation is tornado;

when situation is
      fire:
            write "drop!", ;
      end;
      flood:
            write "swim!", ;
      end;
      tornado:
            write "run!", ;
      end;
end; // case
```

If, during the execution of the program, the expression takes on a value that is not listed under any of the cases, then this causes an error. To avoid this problem, the when statement can be fitted with an else clause so all possible values of the expression can be handled.

Figure 5-8: The When-Else Statement

```
when <expression> is
        <value1>:
                <declarations1>
                <statements1>
        end;
        <value2>:
                <declarations1>
                <statements2>
        end;
        .
        .
        <valueN>:
                <declarationsN>
                <statementsN>
        end;
else
        <declarations>
        <statements>
end;
```

Example: The When-Else Statement

```
enum siuation is fire, flood, tornado, nuke_strike, alien_attack, meteor_impact;
sitauation type situation is tornado;

in case situation of
        fire:
                write "drop!", ;
        end;
        flood:
                write "swim!", ;
        end;
        tornado:
                write "run!", ;
        end;
else
        write "pray!", ;
end; // case
```

Looping Statements

Probably the most powerful construct in computer programming is the loop. A loop is a means of repeatedly executing some action. Without the loop, most computers would run through all of the instructions in their memory in just a few seconds. Looping structures enable the program to do something a fixed number of times or until a certain condition is satisfied.

The While Statement

The while statement is used to execute a sequence of statements repeatedly, as long as a particular condition holds true. It is used whenever you don't know how many times the loop will have to be executed. This kind of loop is used for things like: while not finished, do task; while no character has been found, read keyboard, etc. The condition is tested for at the beginning of the loop, before any statements are executed, so it is possible for the statements not to be executed at all.

Figure 5-9: The While Statement

```
while <boolean expression> do
        <declarations>
        <statements>
end;
```

Example: The While Statement

```
integer i = 2, counter = 1;

write "powers of 2 less than 1000:", ;
while (i < 1000) do
        write "2 ^ ", counter, " = ", i, ;
        i = itself * 2;
        counter = itself + 1;
end;
```

Note that it is possible for the condition of the while loop to never evaluate to false. In this case, the loop will (theoretically) never terminate. This condition is known as an *infinite loop*. In general, you should avoid infinite loops, but there are certain situations in which they are commonly used, such as in anims, the graphics animation procedures.

Example: An Infinite Loop

```
while true do
        write "help!", ;
end;
```

The For Statement

The for statement is used whenever you want to execute the statements of a loop a fixed number of times. The for statement is useful when you want to do things like perform some action for each element in an array. The variable that is used to count the number of iterations in the loop must be of an integral type and may be used by expressions inside of the loop, but its value may not be changed. The loop counter variable is implicitly a constant. The start expression and end expression must also evaluate to an integer.

Figure 5-10: The For Statement

```
for <type> <counter> <assignment_operator>
        <start expression> .. <end expression>
do
        <declarations>
        <statements>
end;
```

Example: The For Statement

```
long factorial = 1, limit = 5;

for integer counter = 1 .. limit do
        factorial = itself * counter;
end;
write limit, " factorial = ", factorial, ;
```

The loop always counts in an increasing direction and always increments the counter variable by 1 each time through the loop. If you want a step size other than 1, or a loop in which the counter decreases, then use a while loop instead. If the start expression evaluates to the end expression, then the loop is executed only once. If the start expression evaluates to a value that is greater than the end expression, then the statements in the loop are not executed at all.

Example: For Loop Using an Enumerated Counter

```
enum animal is cat, dog, cow, pig, sheep;

for animal type animal is cat .. sheep do             // Enumerated for loop
        when animal is
                cat: write "meow", ; end;
                dog: write "woof", ; end;
                cow: write "moo", ; end;
                pig: write "oink", ; end;
                sheep: write "baa", ; end;
        end;
end;
```

For-Loop Counter Protection

For loops in OMAR are different than for loops in C, Pascal, or Java for two reasons:

- The first, and most important difference is that the loop counter variable in OMAR is protected. Inside of the body of the for loop, the loop counter is considered a constant and therefore you can use its value in expressions, but you may not change it. The value of a loop counter is controlled automatically by the loop and you may not interfere with it by trying to change its value yourself.

- The second difference between for loops in OMAR and in other languages is that the counter variable used in the loop only exists inside of the loop statement. As soon as the loop statement terminates, this variable disappears and may not be used for something else.

Example: For-Loop Counter Protection

```
for integer counter = 1 .. 10 do
        write "counter = ", counter, ;      // Ok - you may use the value of the for counter
        counter = itself + 1;               // Compile error - you may not change the loop counter
end;
write "counter = ", counter, ;              // Compile error - you may not use the value of the for counter
                                            // outside of the loop
```

The For-Each Statement

The for-each statement is a special for statement that is used for operating on arrays. You can use this statement to automatically iterate through the elements of an array.

Automating this process has two benefits. First, it is more convenient for the programmer and produces more readable code. Second, it allows a smart compiler to produce faster code by using pointer arithmetic to efficiently step through the elements of the array without having to do a full array dereferencing and bounds checking operations on each array access of each iteration.

In order to use a for-each loop, you need an index variable that is used to reference the current element of the array that you are stepping through. The index variable, like the counter variable in a for loop, is considered a constant and may not be changed by the statements inside of the loop.

Figure 5-11: The For-Each Statement

```
for each <type> <index> in <array>do
        <declarations>
        <statements>
end;
```

<u>*Example: The For-Each Statement*</u>

```
char name[] = "Freida Froglegs";

for each char ch in name do        // Implicitly finds min and max of array
       write ch;                    // Implicit array dereference
end;

// The example code above is equivalent to the following:
//
char name[] = "Freida Froglegs";

for integer counter = min name .. max name do    // Explicit references to the array min and max
       write name[counter];                       // Array dereference and bounds check
end;
```

The for-each statement can also be used with multidimensional arrays. When iterating through a multidimensional array, the loop accesses the elements in row major order, meaning that it begins with the first element of the first row, and continues until it reaches the end of that row, and then goes on to the next row until it has reached the last element of the last row.

<u>*Example: The For-Each Statement & Multidimensional Arrays*</u>

```
integer table[,] = [[1 7 4 2] [3 6 4 6] [2 5 4 9]];

for each integer i in table do
       write "i = ", i, ;
end;

// The multidimensional example above is equivalent to the following:
//
integer table[,] = [[1 7 4 2] [3 6 4 6] [2 5 4 9]];

for integer counter1 = min table .. max table do
       for integer counter2 = min table[] .. max table[] do
              write "i = ", table[counter1, counter2], ; // multidimensional array dereference
       end;                                              // and bounds check
end;
```

The Break and Continue Statements

The break and continue statements make it easier to control the flow of execution in a looping statement. These statements can only be used inside of a for or while loop. Whenever a break statement is encountered, the flow of control is transferred out of the innermost loop that encloses the break statement.

Example: The Break Statement

```
char name[] = "Fred Frogburger";

// Write out the first name only:
for each char ch in name do
	if ch is " " then
		break;
	end;
	write ch;
end;
```

The continue statement is used whenever you want to skip the remaining portion of the current iteration of the loop and jump directly to the top of the loop for the next iteration.

Example: The Continue Statement

```
char name[] = "Fred Frogburger";

// Write out the first and last name with no spaces in between:
for each char ch in name do
	if ch is " " then
		continue;
	end;
	write ch;
end;
```

The break and continue statements may also work with nested loops. When you want to break or continue a loop that is not the innermost loop containing the break or continue statement, you need to specify the designated loop with a label. Then you specify the intended loop to break or continue by stating the name of the label immediately following the break or continue statement. A label is formed by stating the keyword `loop`, followed by an identifier and a colon. For example:

Example: The Labelled Break Statement

```
integer i[,] = [[1 7 4 2] [3 6 4 6] [2 5 4 9]];

loop outer:
for integer counter1 = min i .. max i do
	for integer counter2 = min i[] .. max i[] do
		if i[counter1, counter2] = 5 then
			break outer;
		end;
		write "i[", counter1, ", " counter2, "] = ", i[counter1,counter2], ;
	end;
end;
```

The Return Statement

The return statement is a lot like the break statement, except that it is used to transfer control out of an entire verb procedure. To exit from a question procedure, you must use an answer statement as described in the chapter on procedures.

Example: The Return Statement

```
verb write_up_to_space
	char name[];
is
	// Write out characters until a space is found
	//
	for each char ch in name do
		if ch is " " then
			return;
		end;
		write ch;
	end;
end;		// write_up_to_space
```

The Exit Statement

The exit statement is used to exit from a program entirely. Exit statements are very often used to end a program when errors or invalid data are encountered.

Example: The Exit Statement

```
scalar number = 0.49, root;

// Much later in the program, "number" gets used, but its value is uncertain, so it is checked to be valid.
//
if number < 0 then exit;
else root = sqrt number; end;
```

Some Example Programs

The following example programs demonstrate how several different kinds of statements can be used.

Listing 5-1: Program To Write Out Roman Numerals For Powers of 2 < 5000

```
do Arabic_to_Roman;

verb Arabic_to_Roman is
        integer x, y;

        y = 1;
        while (y < 5000) do
                write y, "    ";
                x = y;
                while (x >= 1000) do write "M"; x = itself - 1000; end;
                if (x >= 500) then write "D"; x = itself - 500; end;
                while (x >= 100) do write "C"; x = itself - 100; end;
                if (x >= 50) then write "L"; x = itself - 50; end;
                while (x >= 10) do write "X"; x = itself - 10; end;
                if (x >= 5) then write "V"; x = itself - 5; end;
                while (x >= 1) do write "I"; x = itself - 1; end;
                write;
                y = itself * 2;
        end;
end;        // Arabic_to_Roman
```

Listing 5-2: Program to Convert Roman Numerals to Arabic Numerals

```
do Roman_to_Arabic;

verb Roman_to_Arabic is
        char roman[];
        integer arabic = 0;

        roman = "MDCCCCLXXXXVIIII";
        for each char c in roman do
                when c is
                        "M": arabic = itself + 1000; end;
                        "D": arabic = itself + 500; end;
                        "C": arabic = itself + 100; end;
                        "L": arabic = itself + 50; end;
                        "X": arabic = itself + 10; end;
                        "V": arabic = itself + 5; end;
                        "I": arabic = itself + 1; end;
                else
                        write "error", ;
                        exit;
                end;
        end;

        write "Roman = ", roman, ;
        write "Arabic = ", arabic, ;
end;        // Roman_to_Arabic
```

CHAPTER 6
Procedures

In order to solve any complex problem, you must find ways of breaking down the task into manageable units. This philosophy is exploited tirelessly in science and computers. The idea is often illustrated by visualizing each component of the solution as a black box, where you know what goes into the box and what comes out but have no knowledge of what goes on inside. Ideally, you should have no need to know what goes on inside the black box.

In terms of computer programming, this means you should be able to use a piece of code without knowing how it works internally, just as you can drive a car without knowing exactly how an engine works. This is why procedures are such a powerful tool—they are the black boxes of a program.

The Concept of Scoping

Just as an engine has a number of internal parts that are necessary for it to work but are not used by any other parts of the car, a procedure can also have its own data, data types, and even its own procedures that are not accessible to the rest of the program.

If something is accessible from a particular place in the program, then we say that it is *visible*. All the things that are visible to a procedure comprise the *scope* of the procedure. Visibility is determined by the textual arrangement of the program. Each time you begin a new procedure, you can declare new things that are not visible to the rest of the program. These things are referred to as *local* because they can only be accessed within that procedure. Anything that is declared outside of any procedure is referred to as *global* because it can be accessed from anywhere in the program after its declaration.

In addition to the things in its own local scope, a procedure can also have access to things that are declared in the scopes that enclose itself. If two variables with the same name but from different scopes are visible, then the one with the closer scope takes precedence, and the others are invisible. Usually, procedures are not nested, so they have access only to their own scope and to the global scope.

Figure 6-1: Procedures Can Access Things in Their Own Scope or an Enclosing Scope

```
do a, b;

// Global scope:
integer i = 1;

verb a is                // Beginning scope of a — scopes visible: global scope, scope of a
    // Declarations in a:
    integer j;

    // Statements in a:
    j = i;
end;                     // Ending scope of a

verb b is                // Beginning scope of b — scopes visible: global scope, scope of b (but not a)
    // Declarations in b:
    integer k;

    verb c is            // Beginning scope of c—scopes visible: global scope, scopes of b and c (but not a)
        // Declarations in c:
        integer m;

        // Statements in c:
        m = i;
        k = m;
    end;                 // Ending scope of c

    // Statements in b:
    k = i;
end;                     // Ending scope of b
```

Verbs

Verbs are procedures that are called upon to do a particular subtask. There are two parts to using verbs.

First, the procedure must be declared. The declaration tells what the procedure does, how it is supposed to do it, and how to call the procedure. In the black box analogy, the body of the procedure declaration specifies what is inside the black box.

Once the procedure is declared, it may be called anywhere where statements are allowed and the procedure's name is in a visible scope. To call a simple verb, all you have to do is state the name of the verb. Calling a procedure is like taking the procedure's code and inserting it wherever the procedure call is made. This makes code with procedures more compact than it would be without procedures, because you can call the same procedure many different places. Instead of having multiple copies of the code, you can reuse the same snippet of code multiple times.

Figure 6-2: Verb Declaration

```
verb <verb name> is
        <declarations>
        <statements>
end;
```

Figure 6-3: Basic Verb Procedure Call

```
<verb name> ;
```

Listing 6-1: Using Verbs to Calculate Averages

```
do averages;

integer n1, n2;

verb write_average is
        integer sum;      // Local intermediate variables
        scalar average;

        sum = n1 + n2;
        average = sum / 2;
        write "average = ", average, ;
end; // write_average

verb averages is
        n1 = 10;
        n2 = 30;
        write_average;    // Procedure call

        n1 = 15;
        n2 = 7;
        write_average;    // Procedure call
end; // averages
```

Questions

Often, procedures are used to do a small part of a larger calculation. In these instances, it is desirable to have the procedure return a value that can be used as part of an expression. In OMAR, this type of procedure is known as a *question*. Because they resemble mathematical functions, question procedures are also sometimes called *functions*.

Question declarations are similar to verb declarations except that you must precede the declaration with the name of the type of data to return. Also, inside of the body of a question declaration you must include an *answer statement*, which tells the procedure to end and return a value to the caller. Every question must terminate with an answer.

Figure 6-4: Question Declaration

```
<return type> question <function name>
is
	<declarations>
	<statements (including an answer statement)>
end;
```

In order to call a question, you must state the name of the question in a place where an expression is called for. Since verb calls go in places where statements go, think of verbs as statements. Questions go where expressions go, so they should be thought of as expressions.

Listing 6-2: Using Questions to Calculate Averages

```
do averages;

integer n1, n2;

scalar question average is
	answer (n1 + n2) / 2;			// An answer statement
end; // average

verb averages is
	n1 = 10;
	n2 = 30;
	write "average = ", average, ;		// Procedure call

	n1 = 15;
	n2 = 7;
	write "average = ", average, ;		// Procedure call
end; // averages
```

The answer statement in a question must include an expression of the same data type as the type that is returned by the question. The last statement in a question must be either an answer statement, or a conditional statement where all conditions end in an answer statement. In this way, the compiler guarantees that a value is returned before the question terminates.

Example: A Question Ending with a Conditional Statement

```
scalar question factorial
	of integer n;
is
	if n = 1 then
		answer 1;
	else
		answer n * factorial of (n - 1);
	end;
end; // factorial
```

Parameters

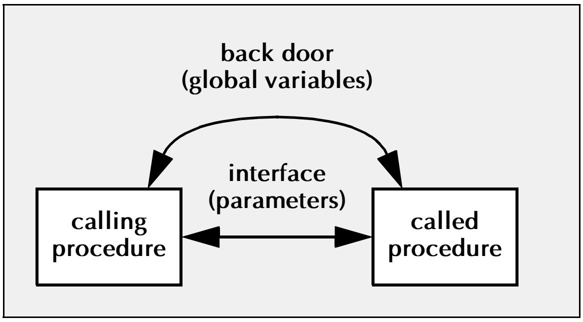
You have seen how to break the program down into manageable parts with procedures. What is needed is a way for all of these individual pieces to be neatly connected so that the pieces are independent but work together. Parameters form the programming glue that is needed to piece together all of the procedures into a cohesive whole. They are the means by which the various parts of the program are interconnected.

The Problem with Global Variables

The black box analogy of program structure assumes that there is some way to get the data into the black box and out of the black box. In the previous examples, we used global variables to communicate with the procedure. Although this works fine, it is not considered good programming style because you can't tell how to interface with the procedure simply by looking at the header of the procedure. Instead you must actually examine the code to see how it works.

In addition, the procedure relies on the existence of the global data which means that the procedure can't stand alone as its own unit. This violates the primary objective of black boxes, which is the encapsulation of functionality. What is needed is a clean way to specify the interface with the procedures.

Figure 6-5: The Procedure Interface



The Procedure Interface: Parameters

The solution to the problem is to specify a way to send values to the procedure and receive values back to the main program. This is done with parameters. Parameters are simply variables that belong to the procedure which can be changed each time the procedure is called. Parameters are declared immediately after the procedure name and before the local declarations.

Figure 6-6: Procedure Declaration with Parameters

```
verb <procedure name>
        <mandatory parameter declarations>
        <keyword parameter declarations (mandatory or optional)>
with
        <optional parameter declarations>
return
        <formatted optional return parameter declarations>
with
        <unformatted optional return parameter declarations>
is
        <declarations>
        <statements>
end;
```

The Different Kinds of Parameters

In OMAR, there are six different kinds of parameters that procedures can use:

Table 6-1: The Six Kinds of Parameters

mandatory	**mandatory-keyword**	**reference**
optional	**optional-keyword**	**optional return**

Since parameters of differing kinds may be mixed together in the same procedure declaration, the calling syntax can be made as natural and expressive as possible. At first it may seem that there are a bewildering number of choices in deciding how to create the procedure interface. Actually, though, in order to decide on what kind of parameter to use, you must ask yourself only three questions:

Question #1: Must the data be returned from the procedure?

The parameter is like an access road to the procedure. Sometimes, you only need a one way street, with data going into the procedure, but not returning. In other circumstances, however, you need a two way street where data can be passed into and returned from the procedure. In still other cases, the caller may want the option of retrieving extra data from the procedure—data that it did not initially pass in.

Question #2: Are there appropriate default values for the parameter?

In many cases, there are logical default values for parameters to have. For example, it's natural for a sphere to have a default radius of 1. Another example might be to define a car object with doors that open and close depending upon the value of a parameter. Since it's logical to have the doors default to the closed position, this is a good choice for an optional parameter.

Question #3: Is the parameter usually used in a context preceded by keywords?

In some cases, the procedure call can be stated very naturally in an English-like way by relying on the fact that the values of certain parameters are often preceded by certain keywords. For example, if you want to create an arrow, it's natural to say `arrow from <here> to <there>` where the values of the endpoints of the arrow are determined by the keywords `from` and `to`. When you call the procedure, you expect to find the values of the endpoints in the places that are indicated by the keywords `from` and `to`.

Table 6-2: Selecting Parameter Usage

must values be returned from the procedure?					
yes		no			
must values come from the caller?		can variable be specified by keywords?			
		yes		no	
		does variable have appropriate defaults?		does variable have appropriate defaults?	
yes	no	yes	no	yes	no
use reference parame- ters	use optional return parame- ters	use optional keyword parame- ters	use mandator y keyword parame- ters	use optional parame- ters	use mandator y parame- ters

Mandatory Parameters

As their name implies, mandatory parameters are parameters that must be supplied to the procedure. An example of mandatory parameters is the parameter that is used by the math function, `sin`. To call the `sin` function, you must supply a scalar value immediately following the name of the function. If no value is supplied, an error message is issued. Mandatory parameters should be used whenever there is no logical choice for default values for the parameters.

Declaration of Mandatory Parameters

Mandatory parameters are specified in the procedure declaration by listing their declarations immediately following the name of the procedure. There is no limit to the number of parameters that may be declared.

Assignment of Mandatory Parameters

When a procedure that uses mandatory parameters is called, a list of expressions is expected following the name of the procedure. The expressions must evaluate to the proper type to be assigned to the parameters that they match.

Example: Calling a Question with Mandatory Parameters

```
scalar question average
	integer a, b;
is
	answer (a + b) / 2;
end; // average

scalar a;

a = average 30 40;
a = average (round 3.5) 40;
a = average -10 40;

a = average 30 .5;           // Compile Error! — .5 is not an integer
a = average (sqrt 10) 40;    // Compile Error! — (sqrt 10) is not an integer
a = average 30 40 15;        // Compile Error! — too many parameters
```

Listing 6-3: Using Questions with Parameters to Calculate Averages

```
do averages;

scalar question average
	integer n1, n2;
is
	answer (n1 + n2) / 2;
end; // average

verb averages is
	write "average = ", average 10 30, ;
	write "average = ", average 15 7, ;
end; // averages
```

Optional Parameters

Optional parameters are useful when it is not always necessary to specify all of the parameters. In these cases, logical default values can be specified for the parameters that are not specified in the actual procedure call.

When you use optional parameters, you need a method of specifying which of the parameters to assign. For the mandatory parameters, you simply match the parameters by order since the number of parameters always equals the number of expressions. Since the number of optional parameters may not be equal to the number of assignments, we can't match them by order. Instead we use the names of the parameters to match them with expressions and assign the parameters values just like a simple assignment statement.

Declaration of Optional Parameters

To signify the beginning of the optional parameters section, add the keyword with at the end of the mandatory parameters section followed by the variable declarations of the optional parameters.

Figure 6-7: Procedure Declaration with Optional Parameters

```
verb <procedure name> with
        <optional param declarations>
is
        <declarations>
        <statements>
end;
```

Assignment of Optional Parameters

Optional parameters are assigned slightly differently for verbs and questions. This is because a verb call acts as a statement, while a question call acts as an expression.

When calling a verb, optional parameters are assigned by following the name of the procedure by the keyword with and then listing any number of assignment statements until the keyword end is given.

Figure 6-8: A Verb Call with Optional Parameters

```
<verb name> with
        <assignment statements>
end;
```

When calling a question, optional parameters are assigned just as they are for verb calls, except that the keyword end is not used. This syntax may at first appear a little confusing because it requires that assignment statements (complete with ending semi-colons) appear in the middle of an expression. For that reason, it is a good idea to surround question calls that use optional parameters with parentheses.

Figure 6-9: A Question Call with Optional Parameters

```
<variable> = (<question name>  with <assignment statements>);
```

The assignment statements that occur in the assignment block are executed after the optional parameters have been created and initialized. It is as if the assignments were occurring at the point of the is in the procedure declaration, right before the body of the procedure begins. Because the assignments occur in the context of the procedure declaration, types that are defined in the optional parameters section of the procedure declaration are also recognized inside the with section of the procedure call.

Listing 6-4: Using Optional Parameters with a Verb

```
do test;

verb init_table
	integer table[];		// Mandatory parameter for table (arrays will be discussed in a later chapter)
with
	integer value = 0;	// Optional parameter for table contents
is
	for integer i = min table .. max table do
		table[i] = value;
	end;
end; // init_table

verb test is
	integer table[1..10];

	init_table table;			// Fill table with 0s (default value)
	init_table table with		// Fill table with -1s
		value = -1;
	end;
end; // test
```

Listing 6-5: Using Optional Parameters with a Question

```
do test;

include "math.ores";

scalar question arch_area
	scalar arch_angle;
with
	scalar radius = 1;
	boolean in_radians is false;
is
	if in_radians then
		answer (pi * sqr radius) * (arch_angle / (2 * pi));
	else
		answer (pi * sqr radius) * (arch_angle / 360);
	end;
end;

verb test is
	scalar area = (arch_area 1.12 with in_radians is true; radius = 2.5;);
	write area, ;
end;
```

Mandatory Keyword Parameters

Sometimes you want to require that extra words should be inserted into the procedure call to make the procedure call more readable. These extra words are known as keyword parameters.

Declaration of Mandatory Keyword Parameters

Mandatory keyword parameters are declared just like the mandatory parameters except that before the variable declaration comes one or more special identifiers, which are the keywords. Any identifier can be used as a keyword so long as it's not a reserved word. The keywords are used to signify that whenever they are encountered in the procedure call, the value of the parameter will follow. Note that the mandatory parameter declarations do not have initializers because initial values are guaranteed to be furnished when the procedure is called.

Assignment of Mandatory Keyword Parameters

To assign values to keyword parameters, state the keyword followed by an expression that can be evaluated to provide a value for that keyword. If the parameters are mandatory, then the parameter values and their corresponding keywords must be given in the same order as in the declaration. No mandatory parameter values may be omitted from the procedure call.

Listing 6-6: Mandatory Keyword Parameters

```
do test;

verb init_table
        integer table[];                // Mandatory parameter for table
        to integer value;               // Mandatory keyword parameter for table contents (no default)
is
        for integer i = min table .. max table do
                table[i] = value;
        end;
end; // init_table

verb test is
        integer table[1..10];

        init_table;                     // Compile error - procedure call must include the mandatory
                                        // keyword followed by the parameter's value (no default value)

        init_table table to 0;          // Fill table with 0s
        init_table table to -1;         // Fill table with -1s
end; // test
```

Optional Keyword Parameters

In some cases, it is desirable to use keywords in the procedure call but not to require all the parameters to be given. In this case, you can use the presence of the keyword to signify that you wish to assign the parameter and provide its value in the expression that follows. Any optional keywords that are not present in the procedure call do not have their parameter values assigned. Since it is possible that optional keyword parameters are not assigned in the procedure call, they must always have default values.

Declaration of optional keyword parameters

Optional keyword parameters are declared just like the mandatory keyword parameters except that they are given initializers. Optional keyword parameters are declared by giving the keyword followed by the variable declaration followed by the initializer.

Assignment of optional keyword parameters

The optional keyword parameters are assigned like the mandatory keyword parameters except that the order that the keywords and parameter values come in is flexible and any or all of the parameter assignments may be omitted.

Listing 6-7: Assigning Optional Keyword Parameters

```
do test;

verb init_table
        integer table[];            // Mandatory parameter for table
        to integer value = 0;       // Optional keyword parameter for table contents (with default)
is
        for integer i = min table .. max table do
                table[i] = value;
        end;
end; // init_table

verb test is
        integer table[1..10];

        init_table table;           // Fill table with 0s (the default value)
        init_table table to 0;      // Fill table with 0s
        init_table table to -1;     // Fill table with -1s
end; // test
```

Reference Parameters

It is sometimes necessary to return data from a procedure. This is most easily done using reference parameters. Reference parameters in OMAR are similar to reference parameters in C++ or var parameters in Pascal. Note that Java has no analogue to reference parameters, since Java has no generalized reference or pointer data type.

Declaration of Reference Parameters

Reference parameters are declared just like mandatory parameters except that the reserved word `reference` precedes the parameter name. When you do this, the parameter becomes a two way link to the variable that is passed in, so any changes that are made to the parameter will be reflected in the variable passed in when the procedure is finished executing.

Assignment of Reference Parameters

Reference parameters are also assigned similarly to mandatory parameters, with the parameter values immediately following the procedure name. One slight difference between reference parameters and mandatory parameters is that a variable must be passed in to the reference parameter instead of an expression.

For example, let's say we have a procedure named `increment` that takes an integer reference parameter and adds 1 to its value. In this case, the procedure call `increment a;` would be valid assuming that `a` is an integer variable. The procedure call `increment (a + 1);`, however, would not be valid because `increment` needs the name of a variable to place the returned value and `a + 1` is not a variable.

Listing 6-8: Use of Reference Parameters

```
do test;

verb swap
	integer reference i;
	integer reference j;
is
	integer k = i;

	i = j;
	j = k;
end; // swap

verb test is
	integer a = 1, b = 2;

	write "a, b = ", a, ", ", b, ;
	swap a b;			// Note that the values of a and b will be changed
	write "a, b = ", a, ", ", b, ;
end; // test
```

Optional Return Parameters

Sometimes, procedures may be called upon to perform some action and then optionally return some parameters that report the status of the operation. This is done using optional return parameters. When you use optional return parameters, it is left up to the individual procedure call to specify which parameters get returned, if any. A common thing to do is to have the procedure report whether it was successful in the operation, or if there was some sort of error. In this case, it is up to the caller to check whether everything is okay or not. In some cases, the caller may not care and so it would not call upon the procedure to return any values.

There are two kinds of optional return parameters, formatted and un-formatted. The relationship between these two kinds is similar to the relationship between mandatory and optional input parameters. Mandatory parameters are 'formatted' parameters because they always must be listed in the same order when called. Optional parameters (which appear in with sections) are 'un-formatted' parameters because they may used in any order, and furthermore, they needn't all be used.

Formatted return parameters, like mandatory parameters, must always be used in the same order, and if one of them is used, they must all be used. However, formatted return parameters are still optional—which means that a call to a procedure that has formatted return parameters need not use them. Un-formatted return parameters, like other optional parameters, may be used in any order, and you may select which ones to use.

Declaration of Optional Return Parameters

To signify the beginning of the optional return parameters section, add the keyword return after any other parameter declarations. Declare any formatted return parameters immediately after the return keyword. If you declare any un-formatted return parameters, they should be declared in another with section after the return keyword.

Neither formatted nor un-formatted return parameters may be declared with an initializer. This is because values should be assigned to the parameters somewhere within the procedure, so having default values for these parameters does not make sense.

Assignment of Optional Return Parameters

Formatted return parameters are returned by following the name of the procedure by the keyword return and then listing a number of variables to receive each of the formatted return parameters. Un-formatted return parameters are attained by following the return section of a procedure call with the keyword with, followed by a number of assignment statements. The assignment statements that occur in this block are executed after the body of the procedure has been executed. Note that if only un-formatted return parameters are used, the return keyword must still be used, so the parameters will appear after the keyword combination of return with.

Un-formatted return parameters are much like ordinary optional parameters but in reverse. With optional parameters, you assign values from the scope of the procedure call to the variables inside the procedure. With un-formatted return parameters, you assign values inside the procedure back out to variables in the scope of the procedure call.

Un-formatted optional return parameters also follow the same syntax as optional parameters in that the assignment syntax is different for verb calls and question calls. In a verb call, the with assignment block is followed by an end keyword. In question calls, the end keyword is never used.

Figure 6-10: Procedure Declaration with Optional Return Parameters

```
verb <procedure name>
        <mandatory parameters and keyword parameters (if any)>
with
        <optional parameters (if any)>
return
        <formatted optional return parameter declarations>
with
        <unformatted optional return parameter declarations >
is
        <declarations>
        <statements>
end;
```

Listing 6-9: Use of Optional Return Parameters

```
do averages;

verb average
        integer n1, n2;
return
        scalar average;
with
        integer sum = 0;
is
        sum = n1 + n2;
        average = sum / 2;
end; // average
```

Listing 6-9: Use of Optional Return Parameters (Continued)

```
verb averages is
      integer a;
      scalar b;

      // Return only the average
      //
      average 20 60 return b;
      write "average = ", b, ;

      // Return both the average and the sum
      //
      average 10 30 return b with a = sum; end;
      write "sum, average = ", a, ", ", b, ;
end; // averages
```

Scoping Modifiers

Whenever you begin a new scope while programming, there is a possibility that you may declare a new variable in the local scope that has the same name as a variable in an enclosing scope and therefore hides this other variable. This phenomenon is sometimes known as *variable shadowing*.

This problem has two possible solutions: First, you can choose to rename the local variable to avoid the ambiguity altogether. Second, you could precede the variable name by a scoping modifier in order to bypass the most local scope and refer to the hidden scope. You can use two scoping modifiers for this purpose.

Use the first scoping modifier, global, whenever you want to bypass the local scope and refer to a variable in the global, or outermost scope.

Listing 6-10: The Global Scoping Modifier

```
do test;

integer i = 5;

verb test is
      integer i = 10;              // Local i 'shadows' global i

      write "i = ", i, ;           // Will write the value 10
      write "i = ", global i, ;    // Will write the value 5
end; // test
```

You can use the second scoping modifier only in procedure calls when you are assigning optional or optional return parameters. In this case, the need is to bypass the scope of the procedure being called (the dynamic scope) and refer instead to the context of the place in the program where the procedure call is made (the static scope). For this, you use the scoping modifier, static.

Listing 6-11: The Static Scoping Modifier Used with Optional Parameters

```
do test;

verb write_integer with
        integer i = 0;
is
        write "i = ", i, ;
end; // write_integer

verb test is
        integer i = 5;                  // Local i

        write_integer with              // Entering scope of write_integer
                i = static i;           // Bypass scope of write_integer to get at local i
        end;                            // Leaving scope of write_integer
end; // test
```

Listing 6-12: The Static Scoping Modifier Used with Optional Return Parameters

```
do test;

integer question function
return with
        boolean error;
is
        error is false;
        answer 1066;
end; // do_stuff

verb test is
        boolean error;                          // Local variable

        integer i = (function return with       // Entering scope of do_stuff
                static error is error;          // Bypass scope of do_stuff to get at local error
        );                                      // Leaving scope of do_stuff
end; // test
```

User-Defined Types As Parameters

Sometimes you may find that it is convenient to have type declarations that are local to a certain procedure's declaration, except for being able to access them in the procedure call. In this sense, these types behave like parameters. To export a type for use in the procedure call, list the type declaration in the interface section of the procedure declaration along with the declarations of the parameters.

Listing 6-13: Types As Parameters

```
do greetings;

verb greeting
	enum language is English, French, German;
	in language type language is English;
is
	when language is
		English: write "Hello", ; end;
		French: write "Bonjour", ; end;
		German: write "Guten Tag", ; end;
	end;
end; // greeting

verb greetings is
	greeting;			// The default language is English
	greeting in French;		// Specify language to be French
end; // greetings
```

Constants As Parameters

Just as it may be handy to export certain types to be used in a procedure or procedure call, it may also be convenient to export constants. Note that since the values of the constants never change, it is not actually necessary to copy the values of the constants into the local variables as with normal parameters, so there is no extra cost associated with having the constants be parameters instead of global constants.

Listing 6-14: Constants As Parameters

```
do logarithms;

include "math.ores";

scalar question logarithm
	const scalar e = 2.71828;
	base scalar b = e;
	of scalar s;
is
	answer ln s / ln b;
end; // logarithm

verb logarithms is
	write "log in base e of 100: ", logarithm base e of 100, ;
	write "log in base 2 of 100: ", logarithm base 2 of 100, ;
	write "log in base 10 of 100: ", logarithm base 10 of 100, ;

	write "e = ", e, ;		// Error - the values of constants listed above are only
					// available inside of a call to the above question
end; // logarithms
```

Final Parameters & Variables

Most types of data are most appropriate as either constants or as variables, because they are either fixed for all time, or may change at any time during the program. There are certain cases, however, where you may want a value to be fixed within the execution of a procedure, but to be able to be changed from one invocation of the procedure to the next. This is what final parameters and variables are for.

A final variable is like a constant because it may not be changed within its scope, but unlike a constant, final variables are created anew each time a procedure or function is called. This allows final variables to have different values between calls.

Example: Final Parameters & Variables

```
scalar question average
        final integer table[];              // These values are protected for the
is                                          // lifetime of this procedure call
        integer sum = 0;
        final integer entries = num table;  // This value is protected for the
                                            // lifetime of this procedure call

        for each integer i in table do
                sum = itself + i;
        end;

        answer sum / entries;
end; // average
```

Static Variables

Normally, local variables are created each time you enter a procedure and they disappear when you leave. There is a special type of local variable, however, that does not disappear when the procedure terminates. This is called a *static variable*. Static variables are similar to global variables, because they always exist, but are like local variables, because they are only visible within the procedure's scope.

Procedure Interface Variables

Variables may be references to other variables, but also may be references to sections of code. A procedure interface variable lets you specify a procedure entirely in terms of its interface, and lets you specify the actual implementation at a later time. Or, you can choose to change the procedure implementation dynamically as the program runs. This allows a great amount of flexibility in certain situations.

Listing 6-15: Use of Static Variables

```
do test;

verb count is
	static integer a = 0;

	a = itself + 1;
	write "a = ", a, ;
end; // count

verb test is					// This will write out the integers 1 through 10
	for integer counter = 1 .. 10 do
		count;
	end;
end; // test
```

Procedure interfaces are similar to what are called *function pointers* in languages such as C or Pascal. The problem with simple function pointers is that normally in the C language, the function pointers are distinguished only by their return type and not by the parameters to the function. This lets you attach an implementation of a function to a function pointer with differing parameters. This is an error and it causes the software to crash or behave erratically.

For this reason, Java, which is based loosely on C, left this feature out entirely. If proper type-checking is enforced, however, this feature may be implemented safely. In OMAR, whenever you attach a procedure implementation to a procedure interface variable, the compiler checks the parameters and return type to ensure that this operation is safe. To assign a procedure implementation to an interface, use the does assignment operator. If no implementation is specified, then the interface variable is said to refer to none. By default, all procedure interface variables are initialized to refer to none. If an attempt is made to call a procedure with no implementation assigned, then a run-time error results.

Example: Using a Verb Procedure Interface Variable

```
verb action1 is
	write "performing action 1", ;
end; // action1

verb action2 is
	write "performing action 2", ;
end; // action2

verb action does none;		// Verb procedure variable declaration and initializer

if action does none then		// Assign action1 or action2 to procedure variable action
	action does action1;		// depending upon whether it has been previously assigned.
else
	action does action2;
end;

action;				// Call whatever procedure action refers to (in this case, action1.)
```

Example: Using a Question Procedure Interface Variable

```
integer question add
      integer a, b;
is
      return a + b;
end; // add

integer question mult
      integer a, b;
is
      return a * b;
end; // mult

integer question f does none     // Question procedure variable declaration and initializer
      integer a, b;              // Procedure parameter declarations
end;

if f does none then              // Assign add or mult to procedure variable f
      f does add;                // depending upon whether it has been previously assigned.
else
      f does mult;
end;

write "f of 4 and 5 = ", f 4 5;  // Call whatever procedure action refers to (in this case, add)
```

Recursion

In some cases, a problem can be more easily solved by a circular, or recursive, way of thinking. The way recursive thinking works is to imagine solving a large problem by breaking into it into pieces and using the same algorithm used to solve the whole problem on solving the subproblems. This may sound confusing, but some problems are actually more easily and efficiently solved in this way.

For example, look at the mathematical function called ‘factorial’. The factorial function, which is denoted by an exclamation point, ‘!’, takes an integer argument and is defined as the product of all the positive integers from 1 up to the argument. The factorial of 6 would be 6! = 1 * 2 * 3 * 4 * 5 * 6 = 720. The recursive way to think of this problem is to say that 6! = 5! * 6. For any number, N, N! = (N - 1)! * N except for 1, where 1! = 1. To solve the problem, instead of writing a loop to multiply all the numbers, you can simply call the function with an argument of 1 less than the argument that was passed in and multiply the result times the argument.
This particular case is presented because it is relatively easy to understand, but the factorial operation could actually be more efficiently coded in the conventional way. With other problems, this is not the case. One important thing to remember is that the recursion must eventually stop. In the example above, the factorial function is no longer called when you get to the number 1. If you didn’t specify that the recursion should stop at 1, then you would have what is known as infinite recursion. In theory, the program would continue forever but in practice, this causes an error because each time the procedure is called, some

Listing 6-16: Recursively Defined Factorial Function

```
do write_factorials;

scalar question factorial
        of integer n;
is
        if n = 1 then
                answer 1;
        else
                answer n * factorial of (n - 1);
        end;
end; // factorial

verb write_factorials is
        for integer counter = 1 .. 10 do
                write "factorial of", counter, " = ", factorial of counter, ;
        end;
end; // write_factorials
```

memory is used up and therefore, the infinite recursion causes the program to run out of memory at some point.

Any type of procedure may be recursive. The factorial function above is an example of a recursive question. Recursive verbs are often used to solve sorting problems. They work by breaking a list into two small lists, recursively calling the sorting procedure of the smaller lists, and then merging the two small sorted lists into one big sorted list.

Recursion is also used to define recursive objects, called fractals. An example of a fractal object is a tree that can be recursively described, by saying that a tree consists of a trunk with a number of branches protruding from it and the branches are actually smaller trees. At some point, the recursion must stop and the branches spawn leaves instead of more branches.

Multiple Procedure Calls

You can concatenate multiple successive calls to the same procedure into a single statement. To do this, separate the parameter lists by commas. This syntax is sometimes more readable and more compact, but will function exactly the same as if the procedure calls were listed separately.

Example: Multiple Procedure Calls

```
// These three individual calls to the same procedure ...
//
draw_line from <0 0 0> to <1 0 0>;
draw_line from <1 0 0> to <0 1 0>;
draw_line from <0 1 0> to <0 0 1>;

// ... can be replaced by
//
draw_line from <0 0 0> to <1 0 0>, from <1 0 0> to <0 1 0>, from <0 1 0> to <0 0 1>;
```

```
// These three object-oriented calls to the same method ...
//
stack push thing1;
stack push thing2;
stack push thing3;

// ... can be replaced by
//
stack push thing1, thing2, thing3;

// These two object-oriented calls to the same method ...
//
hashtable enter thing1 as "fred";
hashtable enter thing2 as "barney";

// ... can be replaced by
//
hashtable enter thing1 as "fred", thing2 as "barney";
```

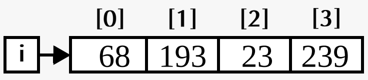
CHAPTER 7
Arrays

An array is a collection of variables of the same type that are grouped together as a single unit. They are useful when you want to create such things as lists or tables of data.

You can think of an array as being like a table of data, where each of the elements in the table contains the same kind of data. Individual elements are referred to by number. The array elements are numbered by consecutive integers. The number of the array element is called the *index*. Array indices in OMAR usually start at 1 and go up to the number of elements in the array, but they may begin at any number, such as 0, which is sometimes more convenient.

Table 7-1: A Simple Array

```
integer i[0..3] = [68 193 23 239];
```



Arrays as Reference Types

Arrays are similar to objects and structs in many ways because they are all reference types. This means that they may be allocated and deallocated at any time during the execution of the program. Also, you may have multiple array variables that refer to the same actual data, or you may have arrays that don't refer to any actual data at all and are therefore said to be `none`. Arrays that are none may not be accessed until they are allocated.

Arrays, like other reference types, may be allocated at the time that they are declared or at any later time during the execution of the program. If an array is to be allocated at some later time, it is still necessary to declare the array beforehand so that there is a reference that may be used to handle the array.

Creating Arrays

Arrays are declared in a similar way to normal variables. To declare an array, you begin by naming the type of the array elements, followed by the name of the variable, just like a regular declaration, except that after the variable name you add an expression of array dimensions that tells how big the array is and what the minimum and maximum array indices are. The array dimensions are of the form:

```
[ <min expression> .. <max expression> ]
```

where min expression evaluates to the integer that is the index of the first element in the array and max expression evaluates to the integer that is the index of the last element in the array. If the minimum index is greater than the maximum index, the array will have no elements.

Example: Some Array Declarations

```
boolean flags[0..4];
scalar radius[0.. (facets * 2)];
vector points[1..100];
char name[1..40];
```

Since array declarations are derived from regular variable declarations, you can freely intermix regular variables with array variables of the same type in the same declaration. For example:

Example: Mixing Array & Non-Array Declarations

```
boolean done is false, flags[0..4], found is false;
scalar temperature, radius[0 .. (facets * 2)], time = 0;
vector center = <0 0 0>, points[1..100], direction = <0 0 1>;
char ch, name[1..40];
```

Accessing Array Elements

Once you have created your array, you must know how to put individual elements into the array and retrieve them from the array. The individual elements of the array can be individually changed without having to change all of the elements of the array. The array elements, themselves, are treated just like individual non-array variables. This means that array elements can be used in assignment statements and expressions, just like ordinary variables. In order to specify an array element instead of the entire array, you need to follow the name of the array variable with an array dereference. The array dereference tells where in the array the element is found. The array dereference is of the form:

```
<variable_name> [ <integer expression> ]
```

where the expression evaluates to an integer that is between the minimum and maximum index of the array. If the expression is outside of the range of the array, then a run-time error occurs.

Listing 7-1: Use of Arrays to Calculate Averages

```
do averages;

integer n = 3, numbers[1..n];

scalar question average is
        integer sum = 0;

        for each integer i in numbers do
                sum = itself + i;
        end;

        answer sum / n;
end; // average

verb averages is
        numbers[1] = 10;
        numbers[2] = 30;
        numbers[3] = 40;

        write "average = ", average, ;
end; // averages
```

Dynamic Arrays

For more complicated programming tasks, you might often find that you don't know at the time an array is declared how many elements it should have. For these cases, it is possible to defer actually creating the space for the array elements to a later time. These types of arrays are known as dynamic arrays. To specify that you don't know how many elements to have in the array at the time of declaration, leave the square brackets empty.

Example: Dynamic Array Declarations

```
integer i[];
char name[];
thing type things[];
```

You cannot assign values to any of the array elements individually until the size of the array is specified. The dimensions of the array are specified using a *dim statement*. The dim statement is formed as follows:

```
dim <variable name> <array dimensions> ;
```

The array dimensions expression is of the same form as used in the array declarations described above. The dim statement works identically for arrays of primitive types as for arrays of reference types. When you use dim to create an array of reference types, it automatically 'news' each of the elements in the array. If you have an array of objects that are of a class with a constructor defined, the constructor is automatically called for each of the elements in the array. If you have an empty dynamic array of structures or objects and you simply want to create an array of none references without actually allocating all of the elements, you must place the keyword none after the dim keyword in order to suppress the allocation of the elements.

Example: Dimensioning Dynamic Arrays

```
struct part has
        char name[];                    // Dynamic array as a field of a struct
        integer inventory_number;
        scalar price;
end; // part

integer i[];                            // Dynamic array declarations
part type parts[], inventory[];

dim i[1..10];
dim parts[1..100];                      // Creates array[1..100] and then creates 100 parts
dim none inventory[1..1000];            // Creates array[1..1000] of null references to parts
for integer counter = 1..500 do         // Creates first 500 out of 1000 parts leaving the last
        new inventory[counter];         //      500 references in the array to be none
end;
```

A common use for dynamic arrays is as parameters to procedures where the array size is determined at the start of the procedure. Each time the procedure is executed, the array parameters may have different sizes and values.

Smart Arrays: Min, Max and Num

It is frequently necessary to know the bounds of an array, since accessing any array element outside of the valid range results in an error. Since the size of a dynamic array is not determined at the time that the array is declared, you need a way later on to find out what the size of the array is. In languages such as Pascal and C, which do not have 'smart arrays,' you must pass around auxiliary integer variables along with the arrays to tell what the array sizes are. This is very inconvenient, not to mention error-prone. In OMAR, as in Java and Modula-2, the arrays are 'smart' and you can ask the array about its size and bounds. This is provided by the built-in functions named min, max and num.

At any time, you can ask what the index of the first and last elements of the array are by calling min and max with the array in question as the argument. To find out what the size of the array (max - min + 1) is, use the function, num. This way, arrays always 'know' their size, so instead of having to keep around separate variables to keep track of the array size, you can just ask the array for its dimensions.

Listing 7-2: Use of Smart Dynamic Arrays to Calculate Averages

```
do averages;

scalar question average
        integer numbers[];
is
        integer sum = 0;

        for integer counter = min numbers .. max numbers do
                sum = itself + numbers[counter];
        end;

        answer sum / num numbers;
end; // average

verb averages is
        write "average = ", average [10 30 40 15 8 60], ;
        write "average = ", average [10 30 50 15 8 60], ;
        write "average = ", average [10 30 60], ;
end; //averages
```

Assigning Arrays

Although you can manipulate arrays by assigning each of their elements separately, it is sometimes more convenient to initialize and assign arrays as a whole unit. If two arrays have the same dimensions, then the values of one array can be assigned to the other array using a simple = assignment statement. If the array that you are assigning to is empty, then when you perform the assignment, it will automatically be allocated to the size of the array that you are assigning. In the case that both arrays are allocated but are of differing sizes, a run-time error occurs when the assignment is attempted.

Array Expressions

As an alternative to assigning one array to another, you can also assign an array a list of values given by an array expression. An array expression specifies each of the values of the elements of the array inside a pair of square brackets. The array that you are assigning to must either be none or have the same number of elements as the array expression, or else an error occurs. Arrays of type char can also be assigned by providing a string of characters inside of double quotes.

When you assign an undimensioned array with an array expression, its default minimum index is 1, as it is for the c array in the example below.

Example: Assigning Arrays

```
integer a[0..3], b[1..4];
integer c[] = [10 20 30];     // Indices for c become 1..3
char message[] = "game over - you lose";
char name[];

write c[1], ;                  // Writes "10"

a = [45 84 22 91];

name = "boo";
b = a;                         // OK — arrays both have 4 elements
c = a;                         // Run-time error — arrays are of differing sizes
```

Assigning Arrays by Reference

When you use the = operator to assign arrays, as in the example above, the program is really copying one array and assigning the duplication to another. Arrays, however, can also be assigned by reference, much like reference variables. Instead of using the refers to operator that reference variables use, arrays use the is operator to assign by reference.

When arrays are assigned by reference, it doesn't matter whether they already have the same dimensions or not; they need only be of the same type. Once an array is assigned by reference, it then refers to the array that was assigned to it, so that a single array may be referred to by multiple array names.

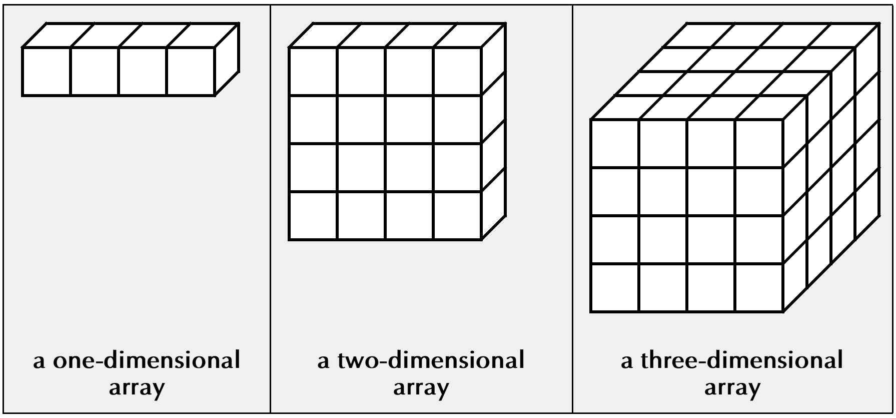
Example: Assigning Arrays by Reference

```
integer a[] = [50 51 52 53 54 55];           // Indices for a become 1..6
integer c[] = [10 20 30];                    // Indices for c become 1..3

char message[] = "game over - you lose";
char name[] is message;

write name, ;          // Writes "game over - you lose"

c is a;                // Now a and c refer to the same array.
write c[1], ;          // Writes "50"
c[6] = 60;             // Changes the 6th element of both c and a.
write a[6];            // Writes "60"
```

Multidimensional Arrays

An ordinary array is one-dimensional because it has length only. A two-dimensional array has length and width; therefore you can imagine a two-dimensional array as a table of entries. If you stack these tables one on top of another like floors in a skyscraper, then you have a three-dimensional array, with each element uniquely defined by three coordinates, the indices of the two dimensions in the table plus the index of the table in the stack. In theory, you can have arrays with any number of dimensions that you wish, although it gets difficult to visualize them past three dimensions. All multidimensional arrays are 'square' because the bounds of each dimension are the same for all subarrays.

Figure 7-1: Multidimensional Arrays



To declare a multidimensional array, use the same syntax as a single-dimensional array but with multiple min..max pairs separated by commas inside the brackets of the array dimensions expression. To access an array element, you must provide multiple indices separated by commas inside the square brackets of the array index expression. If you need to create a dynamic array, you must leave out the min..max bounds on each dimension in the array since it doesn't make sense to specify one dimension and not another of a multidimensional array.

Example: Multidimensional Arrays

```
integer a[1..10];          // One-dimensional array of integers
integer b[1..10, 1..20];   // Two-dimensional array of integers
integer c[,,];             // Three-dimensional dynamic array of integers

dim c[1..5, 1..8, 1..2];   // Dimensioning a multidimensional array
a[1] = 0;                  // Dereferencing a one-dimensional array
b[1, 1] = 0;               // Dereferencing a two-dimensional array
c[1, 1, 1] = 0;            // Dereferencing a three-dimensional array
```

Min, Max, and Num with Multidimensional Arrays

Using the functions min and max with multi-dimensional arrays is a bit tricky because you must keep in mind which dimension of the array you are examining. You can specify which dimension to refer to by adding extra empty square bracket pairs after the name of the array which is passed to the min, max, or num function. For example, if you have the following three dimensional array:

```
integer i[1..10, 2..20, 3..30];
```

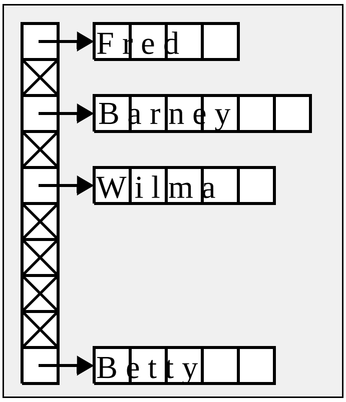
Then you can find the bounds on each dimension of the array as follows:

```
min i    =  1,   max i    =  10
min i[]  =  2,   max i[]  =  20
min i[,] =  3,   max i[,] =  30
```

Arrays of Arrays

For most applications, it is appropriate for multidimensional arrays to be square. There are a few cases, however, where you can visualize your data as belonging in tables that are not necessarily square. For example, if you have a table of names, then you conceptually have a two dimensional table of characters with each row having a different length.

Figure 7-2: An Array of Arrays

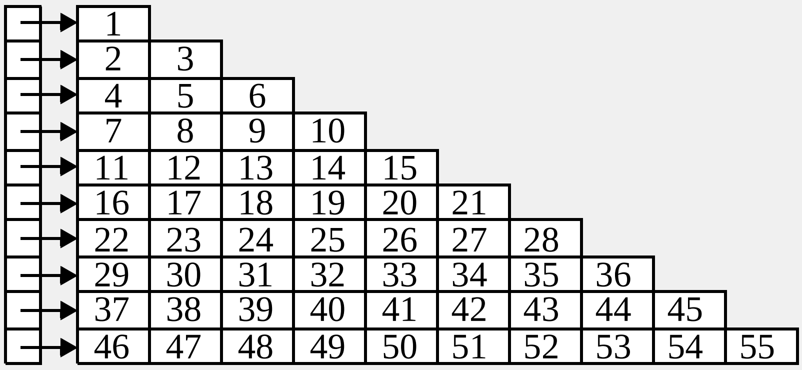


The proper way to implement non-square arrays is by creating arrays of arrays. This way, each subarray has its own min and max information and may be any size that is needed. To create an array of arrays, simply list the array dimensions expressions after the name of the variable in the declaration.

Example: An Array of Strings (An Array of Char Arrays)

```
char names[1..10][];

names[1] = "Fred";
names[3] = "Barney";
names[5] = "Wilma";
names[10] = "Betty";
```

Figure 7-3: A Triangular Array of Arrays



Example: Creating a Triangular Array of Arrays

```
integer i[1..10][];
integer index = 1;

for integer row = min i .. max i do
        dim i[row][1..row];
        for integer column = 1 .. row do
                i[row][column] = index;
                index = itself + 1;
        end;
end;
```

Since creating arrays of arrays is obviously more flexible than square multidimensional arrays, you may be wondering why not just make all arrays 'arrays of arrays'. Indeed, this is what is done in Java. For many applications, particularly scientific applications where large square arrays may be needed, this is a very bad idea. This is because each subarray allocation in a non square array has a cost associated with it in terms of allocating the array and in terms of the extra memory needed for the control information, including the min and max indices that are stored in each array. While this is not prohibitive for small arrays

like those you may find in a particular business application, it can become prohibitive for more demanding applications.
For example: the declaration:

```
integer i[1..10][2..20][3..30]
```

will result in the following:

```
1)  allocate an array 1..10 of references to type integer[][]
2)  repeat the following 10 times (once for each element):
    1)  allocate an array 2..20 of references to type integer[]
    2)  repeat the following 19 times (once for each element):
        1)  allocate an array 3..30 of integer
```

This one declaration will result in a total of 1 + (10) * (1 + 19) = 201 separate arrays being allocated, so you can see that allocating large arrays of arrays of arrays can be potentially very slow and memory consumptive. This is why it's best to use square multidimensional arrays whenever possible.

Listing 7-3: Min and Max with Arrays, Arrays of Arrays, and Multidimensional Arrays

```
do test;

verb test is
        // A single-dimensional array
        integer a[1..10];

        // Arrays of arrays
        integer b[1..10][1..20];
        integer c[1..10][1..20][1..30];

        // Multidimensional arrays
        integer d[1..10, 1..20];
        integer e[1..10, 1..20, 1..30];

        // Find min, max of a single dimensional array
        write "single dimensional array:", ;
        write "min, max a = ", min a, " .. ", max a, ;
        write;

        // Find min, max of arrays of arrays
        write "arrays of arrays:", ;
        write "min, max b = ", min b, " .. ", max b, ;
        write "min, max b[1] = ", min b[1], " .. ", max b[1], ;
        write "min, max c = ", min c, " .. ", max c, ;
        write "min, max c[1] = ", min c[1], " .. ", max c[1], ;
        write "min, max c[1][1] = ", min c[1][1], " .. ", max c[1][1], ;
        write;

        // Find min, max of multidimensional arrays
        write "multidimensional arrays:", ;
        write "min, max d = ", min d, " .. ", max d, ;
        write "min, max d[] = ", min d[], " .. ", max d[], ;
        write "min, max e = ", min e, " .. ", max e, ;
        write "min, max e[] = ", min e[], " .. ", max e[], ;
        write "min, max e[,] = ", min e[,], " .. ", max e[,], ;
end; // test
```

Assigning Sub-Arrays

Sometimes you find that you would like to be able to assign just a piece of one array to another array. In other computer languages, you would do this by writing a snippet of code to loop through the elements copying the desired elements one by one. In OMAR, however, you can automatically do this by specifying a subrange of the array to copy.

Example: Assigning Sub-Arrays

```
integer i[1..5], j[1..10];
integer a[0..2][0..2];
integer b[1..3, 1..3];
integer c[1..10, 1..10, 1..10];

// 1-dimensional arrays
//
i = j[1..5];                 // Copy a subarray to an array
i[1..3] = j[8..10];          // Copy a subarray to another subarray
i = j[6..];                  // Copy an implicit subrange to an array
i[..4] = j[7..];             // Copy an implicit subrange to another implicit subrange
i[..] = j[6..];              // More implicit subranges

// 2-dimensional arrays
//
a = b;                       // Copy a multidimensional array to an array of arrays
a = b[1..3, 1..3];           // Copy a subportion of a multidimensional array to a
                             // multidimensional array
a[1..2][1..2] = b[1..2, 1..2]; // Copy a subportion of a multidimensional array to a
                             // subportion of an array of arrays

// 3-dimensional arrays
//
i = c[1, 1, 1..5];           // Copy a row of a multidimensional array
j = c[1..10, 1, 1];          // Copy a column of a multidimensional array
a = c[1, 1..3, 1..3];        // Copy a multidimensional subportion of a multidimensional
                             // array to an array of arrays
b = c[1, 1..3, 1..3];        // Copy a multidimensional subportion of a multidimensional
                             // array to a multidimensional array
```

Resizing Arrays

Since arrays must be declared to be of some fixed size and programs often do not know ahead of time how much space may be required for certain kinds of data, we run into a problem. In many languages where array allocation is cumbersome and inflexible, the general approach is to use lists for everything. Linked lists can have a large overhead associated with them both in terms of extra storage and also in traversal time so this is not the ideal solution.

The most obvious thing to do when you run out of space in an array is to allocate a new, bigger array and to copy the information from the first array into the new array. In most applications, this will work just fine. There are cases, however, where this can present a problem. Suppose there are other data structures that refer to this first array. When you replace the old array with the new, larger array, the other data structures will still refer to the first array. In many cases, you may not be able to find out where these references are without checking every piece of data that is used by your program. This is usually a completely infeasible option.

Example: The Array Reallocation Problem

```
integer a[1..10], b[] is a;

write "before:", ;
write "min, max a = ", min a, ", ", max a, ;// a and b refer to thesame array
write "min, max b = ", min b, ", ", max b, ;//

a is none;
dim a[1..20];

write "after:", ;
write "min, max a = ", min a, ", ", max a, ;// a and b refer to different arrays
write "min, max b = ", min b, ", ", max b, ;//
```

The Redim Statement

The solution to this problem is to use the redim statement. The redim statement works just like the dim statement except that it is used to enlarge the size of a preexisting array instead of to create a new array. If the redim statement is called using a null array, then an error will result. Since the redim statement resizes an existing array without creating a new one, any preexisting references to that array will refer to the new array after the array is resized.

Example: The Redim Statement

```
integer a[1..10], b[] is a;

write "before:", ;
write "min, max a = ", min a, ", ", max a, ;// a and b refer to thesame array
write "min, max b = ", min b, ", ", max b, ;//

redim a[1..20];

write "after:", ;
write "min, max a = ", min a, ", ", max a, ;// a and b still refer to the same array
write "min, max b = ", min b, ", ", max b, ;//
```

CHAPTER 8
Input/Output

Although it is possible to write programs that are self-contained, many programs need to input data or output data or messages to the user. The methods used to do this are usually inconsistent and system-dependent because of the differences in operating systems and hardware involved. For instance, if you ask the computer to print a message and there are no windows open, where does it go? Or if you try to read a character, where does it come from? Although normally these procedures output data to the screen and take data from the keyboard, operating systems such as UNIX allow these sources to be redirected, so this may not be the case.

OMAR provides a small but useful set of procedures that can serve the majority of I/O related tasks. All input/output procedures are defined for the primitive types: `char`, `byte`, `short`, `integer`, `long`, `scalar`, `double`, `complex`, and `vector`.

Input: The Read Statement

The input procedure reads data from the standard source for input, which is usually the keyboard. If you are reading from the keyboard, then this procedure waits until the values are typed in before returning to the program. If the incoming data cannot be interpreted as the desired type then an error will occur. The form of a read statement is as follows:

Figure 8-1: The Read Statement

```
read v1, v2, v3, ... vN ;       // Where v1 ... vN are variables
                                // of a primitive type or array
                                // of chars (string)
```

Example: The Read Statement

```
integer number;

write "input a number to square:";
read number;
write "the square of ", number, "is ", (number*number), ;
```

Output: The Write Statement

The output procedure, write, is similar to the input procedure, read. Unless the output is redirected to some other destination by the operating system, the messages go to the standard output, meaning the console, or screen.

Write can take arguments of all of the primitive types, boolean, char, byte, short, integer, long, scalar, double, complex, and vector, plus arrays of characters (strings). If write is called with no arguments, then it prints a blank line. Usually, when output messages are printed, you want to go to the next line at the end of the message, so usually the last argument in a write statement is an empty argument.

The form of a write statement is as follows:

Figure 8-2: The Write Statement

```
write expr1, expr2, expr3, ... exprN ;// Where expr1 ... exprN are
                                      // expressions which evaluate
                                      // to a primitive type or an
                                      // array of chars (string)
```

Example: The Write Statement

```
integer N = 10, a = 64;
scalar b = 64;
complex c = <4 1>;

write "hello world!", ;
write "the value of the variable, N, is ", N, ;
write "the values of a, b, and c are ", a, ", ", b, ", and ", c, ;
```

Program Arguments

In operating systems such as UNIX, which operate from a command prompt, it makes sense to allow a program to receive messages from the command line. Also, when making OMAR applets to be placed in web pages, it is often useful to be able to pass arguments into the applet from the HTML code. In C, this is done using the argv and argc parameters to main. In OMAR, this is done using a similar mechanism.

Normally, any procedure listed in the program header must have no parameters. The one exception to this rule is to allow the main procedure to have one parameter, an array of strings (arrays of chars), to hold the program arguments. This parameter may have any name that you wish but it must be of the type, string[], or char[][].

If you are familiar with C, note that this is similar to the argv, argc convention but more elegant because arrays in OMAR implicitly know their own bounds and size, so you don't need a separate parameter for the size of the array.

Listing 8-1: A Program Receiving Program Arguments

```
do write_args;

type string is char[];

verb write_args
      string type args[];
is
      write "The program arguments are:", ;
      for integer counter = min args .. max args do
            write "argument[", counter, "] = ", args[counter], ;
      end;
end; // write_args
```

If, in HTML code, you type, (*name of OMAR applet*) shish boom bah, then the parameter, args, will end up referring to an array, size [1..3], that refers to three strings, "shish", "boom", and "bah". Each string of characters separated by spaces that comes after the name of the input file becomes an element in the program argument array.

CHAPTER 9

User-Defined Types

The primitive types that have been described so far—char, integer, scalar, and so forth—are the basis for representing every form of data that is used in a computer program. Theoretically, you could write any program using just these primitive types and no others. Most applications, however, require the computer program to describe more complex situations than may be conveniently expressed using just the primitive types.

User-defined types are a general way of structuring data to represent more complex situations. There are four basic kinds of user-defined types in OMAR: type aliases, enumerated types, structures, and classes. Classes, also known as subjects, are like sophisticated structures, and are described in much more detail in the later chapters on "Object-Oriented Programming".

Basic Syntax of User-Defined Types

User-defined types use a slightly different syntax than primitive types. To distinguish user-defined type names from other identifiers, OMAR requires that user-defined type names are followed by the keyword type when they are used. Thus, a variable declaration using a user-defined type has the following syntax:

Figure 9-1: Syntax of a Variable Declaration Using a User-Defined Type

```
<type name> type <variable name> <optional initializer> ;
```

*Example: Using the **type** Keyword*

```
enum material is plastic, glass, porcelain;       // An enumerated type declaration
                                                  // (as described in a later section)

material type jug_material;
material type vase_material is glass;
material type plate_material is porcelain;
```

Because user-defined type names are always distinguished from identifiers by the keyword `type`, a user-defined type and a variable of that type can have the same name. You can easily tell the difference between the variable name and the type name because the type name must be followed by the keyword `type`.

Example: Giving a Type & a Variable the Same Name

```
material type material;
material is plastic;           // Assigns a value to the variable material
```

Type Aliases

The *type alias*, the simplest user-defined type, is really just a way of giving a new name to an existent type. Type aliases can be useful when you are frequently using a primitive type for a very particular purpose. For example, you may have an OMAR program in which you frequently use the `vector` type to represent colors. You could create a type alias for the `vector` type named `color`. The `color` type would have the exact same properties as the `vector` type, except that it would have a different name and would use the syntax of a user-defined type.

Type aliases can also be used to represent arrays. This proves especially useful in the case of strings. If you define a type alias `string` that represents a `char[]`, you no longer need to use array brackets when using character arrays.

Figure 9-2: Type Alias Declaration Syntax

```
type  <type alias name>  is  <type name> <optional array brackets> ;
```

Example: Using Type Aliases

```
type color is vector;
type switch is boolean;
type string is char[];
type table is integer[,];

const boolean on is true, off is false;
integer i[1..5,1..20];

color type color = <.5 0 .5>;
switch type power_switch is on;
string type name = "Fred";
table type table is i;         // A valid assignment because the table type is equivalent to an integer[,]
```

Enumerated Types

Enumerated types are used to represent a situation where there are a known, fixed number of states. Some examples of things that would be well represented by enumerated types are days of the week, the seasons of the year, the suits in a deck of cards, and the nucleotide base types in a DNA molecule. If you didn't know about enumerated types, you would probably represent these types of situations using an integer to represent the state and defining a set of integer constants for the values of the states. For example, winter is season #1, spring is season #2 and so on. In languages like Java, where enumerated types do not exist, this is how the situation is handled.

Example: Using Integers to Specify a Variable with a Fixed Number of States

```
const integer straw = 1;          // definitions of the possible states
const integer wood = 2;           //
const integer brick = 3;          //

integer material = straw;
```

Using integers in this situation will work fine and you could implement things this way in OMAR as well; however it is really not a good idea for two reasons.

- An integer can take on values that are outside the range of the meaningful values that represent valid states. In the example above, you could conceivably assign the integer material the value of 47, which would be completely meaningless because there is no corresponding material for the value of 47. Although this assignment is not considered an error, errors may crop up in other areas of the code where it is assumed that the material will only take on the values in the defined range.
- Since all integers are considered to be the same type, just using integers to represent different things gives you no type-checking protection. If you have two integers that represent completely different things, then you can assign them to each other without causing an error because they are both integers.

Example: Missed Errors Caused by Using Integers Where Enumerated Types Should Be Used

```
const integer spring = 1;           // definition of the seasons of the year
const integer summer = 2;
const integer autumn = 3;
const integer winter = 4;

const integer hearts = 1;           // definition of the suits of cards
const integer diamonds = 2;
const integer spades = 3;
const integer clovers = 4;

integer season = fall;
integer suit = hearts;

// mistakes that are NOT caught by the compiler:
//
season = hearts;                    // oops - assigned enum constant to wrong "type"
suit = winter;                      // oops - assigned enum constant to wrong "type"
season = suit;                      // oops - assigned an integer "type" to another integer "type"
suit = 128;                         // oops - assigned value out of range
```

Fortunately, there is a solution to the problem: enumerated types. When you define an enumerated type, you list each of the possible states that variables of this type may have. Then, when you declare a variable of this type, the compiler assures that you only assign valid states to that variable. Also, the compiler will not allow you to assign two variables of differing types to one another, so you don't have to worry about making any of the mistakes illustrated above because the compiler will tell you if you do something wrong. In addition, an enumerated type may have the value of none, a keyword indicating that the enumerated variable does not have a valid state.

Example: Using an Enumerated Type to Specify a Variable with a Fixed Number of States

```
enum material is straw, wood, brick;            // definition of the possible states

material type material is none;
material is straw;
```

Example: Errors Caught when Using Enumerated Types instead of Integers

```
enum season is spring, summer, autumn, winter;   // definition of the seasons of the year
enum suit is hearts, diamonds, spades, clovers;  // definition of the suits of cards

season type season is fall;
suit type suit is hearts;

// mistakes that ARE caught by the compiler:
//
season is hearts;                   // compile error - assigned enum constant to wrong type
suit is winter;                     // compile error - assigned enum constant to wrong type
season is suit;                     // compile error - assigned a type to another type
suit is 128;                        // compile error - assigned value out of range
```

Structures

Another kind of user-defined type is a *structure*. OMAR structures are similar to structures in C++ or records in Pascal. Structures allow you to group together a set of related data into one conceptual unit.

Consider, for example, how you might represent something like the size and shape of a button on the screen. You could keep track of the horizontal and vertical location and the length and width of the button as a set of integer variables. But that's already four variables just for one button. In practice, you often find that you need to use a group of many more than just four variables to represent some concepts. Your button, for example, could also have a label, a color, and a font associated with it. If you had to explicitly declare and name variables for each of these properties, then it would soon become unmanageable.

Structures allow you to create a sort of data template for an idea and then once the template is declared, you can create multiple instances of this idea, each with its own individual set of data.

Example: Declarations of Two Buttons without the Use of Structures

```
char button1_label[];
integer button1_hcenter, button1_vcenter;
integer button1_hsize, button1_vsize;

char button2_label[];
integer button2_hcenter, button2_vcenter;
integer button2_hsize, button2_vsize;
```

Example: Declarations of Two Buttons with the Use of Structures

```
struct pixel has                    // A structure type declaration (pixel is a new structure type)
      integer H, V;
end; // pixel

struct button has                   // A structure type declaration (button is a new structure type)
      char label[];
      pixel type center, size;      // A structure variable decalaration (center and size are new
variables)
end; // button

button type button1, button2;       // A structure variable declaration (button1 and button2 are new
variables)
```

Accessing a Structure's Fields

Each of the data elements, or *fields*, in a structure has a name just like a regular variable. A field is accessed by giving the name of the structure variable, followed by a single quote and a letter s, followed by the name of the field. The single quote and letter s are together known as the *apostrophe s* operator, which is used to denote possession, as in English.

Example: Accessing a Structure's Fields

```
pixel type screen_center;

screen_center's H = 512;
screen_center's V = 384;
write "center of the screen = ", screen_center's H, ", ", screen_center's V, ;
```

Since structures may also contain other structures, you may have to dereference a structure multiple times to get at the field you need. For example:

Example: Accessing a Nested Structure's Fields

```
button type button;

button's label = "quit";        // a single field dereference
button's size's H = 40;         // multiple field dereferences
button's size's V = 10;
```

The With Statement

Sometimes you will find that you have a sizeable block of code that references the fields of a particular struct repeatedly. Since it is inconvenient to repeatedly dereference this structure to get at its fields, a shorthand notation is provided, the with statement. The with statement allows you to specify a certain block of code in which you are understood to be referring to a particular struct. Inside of this block, whenever the pronoun its is encountered, it is understood to be referring to the struct named at the top of the with statement block.

Figure 9-3: The With Statement

```
with <struct expression> do
        <declarations>
        <statements>
end;
```

With statements may be nested, in which case, the priority goes to the innermost block. If you are familiar with Pascal, note that OMAR's with statement differs from Pascal's because it requires the its pronoun to explicitly refer to a field instead of having any unresolved identifier implicitly refer to a field of the object. This makes OMAR's with statement much safer because you don't have to worry about a reference to a local variable inside of a with block referring to a structure member by mistake.

Example: Using a With Statement

```
button type button;

with button do
        its label = "quit";

        with its center do
                its H = 512;
                its V = 384;
                write "button's center = ", its H, ", ", its V, ;
        end; // with
end; // with
```

Assigning Structures

If two structure variables are of the same structure type, you can assign one to another just like normal, primitive variables by using the = operator. In the case of structs, as with arrays, the = operator actually makes a complete copy of one structure and assigns the copied structure. Therefore, the = operation can get to be quite expensive when dealing with large structures.

A more efficient method of assignment is assignment by reference, which uses the is operator. When you assign one structure variable to another using the is operator, you are in effect making both variables refer to the same structure. This means that a change to one variable effects both variables, as is the case with reference variables of primitive types. Creating multiple references to a structure is discussed more in a later section.

Example: Assigning Structures

```
pixel type pixel1
pixel type pixel2;
pixel type pixel3;

pixel1's H = 512;
pixel1's V = 384;

pixel2 = pixel1;                          // pixel2 is a copy of pixel1
pixel3 is pixel1;                         // pixel3 refers to pixel1

pixel1's H = 480;
write "H of pixel1: ", pixel1's H, ;      // writes out 480
write "H of pixel2: ", pixel2's H, ;      // writes out 512
write "H of pixel3: ", pixel3's H, ;      // writes out 480
```

Primitive Types and Reference Types

Structured types (structures, subjects, and arrays) are distinctly different from the primitive types, such as `integer` and `char`, for two reasons.

- They may be allocated and deallocated. When a structure is deallocated, then the memory that is used to represent the structure is freed up for use elsewhere. Since user-defined structures and arrays are more complex than the simple primitive types, it makes sense to be able to deallocate them when they are no longer needed.
- Structured types may be used to create dynamic linked data structures. This means that the user-defined types may contain links to other structures or objects that may contain links to yet other types and so forth. This allows you to build very complex and flexible data representations. Instead of being completely defined at the time the program is written, dynamic data structures may actually be assembled and disassembled as the program runs.

Since structures, subjects and arrays are all considered reference types we may henceforth use the term *object* to refer any one of these types when discussing memory allocation issues.

Unallocated Objects

In order to signify to the computer that an object is no longer needed and may be reclaimed, you can assign the reference to it a special value, called `none`, that indicates that the reference no longer refers to any structure instance. If a reference to a structure is `none`, then you may not access any of its fields because it is not valid. If you try to access a field of a structure that is `none` (deallocated, invalid), then a run-time error results. Unused objects are eventually reclaimed by a special mechanism, known as the garbage collector, and reused by the system.

Example: Freeing a Structure for Deallocation

```
button type button;
button's label = "quit";
button is none;
write "button's label = ", button's label, ;        // run-time error - button is none
```

Normally, when you declare a structure, the structure is automatically allocated for you, just as a primitive type such as an `integer` or `char` would be. Since objects may be created dynamically, as the program runs, you may wish to defer the creation of the object until later, when it is more appropriate to allocate the structure. This is done by initializing the structure to `none` inside of the initializer portion of the declaration.

Dynamically Creating Objects

Objects can be created in two different ways. The first place where they can be created is in their declarations. In this sense, they appear to be just like simple, primitive types such as integer or char. The second way to allocate structures and objects is to dynamically allocate them in a new statement as the program is running.

Suppose you create an unallocated object that you want to use at some later point during the execution of the program. In order to allocate the object, you pass the object to a special built-in procedure called new which does three things.

- It allocates memory space for the object.
- It executes the initializers to set the initial values of each field in the structure.
- For instances of a subject, it calls the constructor method, if one exists, to initialize the object (more on this in the section on object-oriented programming).

If you try to new an object that already refers to an allocated object, then a run-time error results.

Example: Dynamically Allocating Objects

```
button type button1 is none;

new button1;
button1's label = "quit";
new button1;                    // error - button is already allocated (not none)
```

Creating Multiple References to an Object

You can visualize a variable of a reference type (either a structure or a class) as a pointer that is either none, or else points to a block of data that contains the fields of the structure. This extra level of indirection allows several reference type variables to share the same block of data. You can think of the references as handles to the actual block of data. With structures and objects, you can have several variables throughout the program that grant access to the same piece of data. Although this can sometimes result in confusion, it is also a very powerful feature that allows linked and even circular data structures to be built.

Example: Creating Multiple References to a Structure

```
button type button1;                    // creates a structure referred to by button1
button type button2 is button1;         // creates another reference to button1
button type button3 is none;            // creates a reference to none

button3 is button2;                     // button2 and button3 share data (with button1)
button1's name = "quit";
write "button2's name = ", button2's name, ;     // prints the name of button2 (same as button1)
write "button3's name = ", button3's name, ;     // prints the name of button3 (same as button2)
```

Since arrays are also considered to be structured types, you may also create multiple references to arrays. Arrays are somewhat simpler than structures or classes, though, because you may never have circular references. For example, you may have a structure where one of the fields is actually a reference to the original structure. With arrays, this is not possible because the dimensions of the elements are always less than the dimensions of the original array.

Example: Creating Multiple References to an Array

```
integer i[1..10];
integer j[] is i;                 // array j shares data with array i

i[1] = 47;
write "value of j[1] = ", j[1], ;    // verify that the contents of j are the same as the contents of i
```

Freeing Objects

Remember that reference types are allocated either by their declarations or by a following new statement. At some point, when you are finished using these objects, you are going to want to return them to the system for reuse. If you are a C++ programmer, then you know that each call to `new` should eventually be followed by a `delete` command to deallocate each object.

In OMAR, as in Java, the programmer can not and need not deallocate objects. Objects that are no longer referenced by any variables are *automatically* collected for reuse in a process known as *garbage collection*. At first, the problem of figuring out when an object can be recycled seems straightforward enough. But remember that objects may have multiple references to them. Therefore simply setting a reference to `none` does not automatically cause its object to be freed, because there may still be other references to this object in other parts of the program.

In addition, there are circumstances where references to an object are implicitly removed.

- When you leave a procedure, all of its local variables disappear. Therefore, any objects or structures that are referenced only through a local variable are then inaccessible and may be recycled by the garbage collector.

- References to an object are also implicitly removed when you have a structure that contains references to other structures. When the references to the parent structure are removed, then any of its fields not referenced elsewhere will be freed up for recycling.

Listing 9-1: Implicitly Freeing Objects

```
do test;

integer i[];

verb do_stuff is
	// when this procedure ends, j and k disappear. Since the variable, i, also references the array,
	// [5 6 7 8], this array persists after the procedure is finished. The other array, [1 2 3 4],
	// is recycled because the only reference to it, j, has disappeared.

	integer j[] = [1 2 3 4];
	integer k[] = [5 6 7 8];

	i is k;			// let the external variable, i, share the array, [5 6 7 8]
end; // do_stuff

verb test is
	do_stuff;
	write "min, max i = ", min i, ", ", max i, ;
end; // test
```

Testing Object Allocation

Often, you need to check whether an object has been allocated before you perform some operation with it. One way to do so would be to use the following syntax:

Example: Testing Whether an Object Has Been Allocated

```
if button isn't none then
	{operations involving button}
end;
```

As an alternative to the peculiar double-negative syntax in the example above, OMAR provides a `some` operator, which works like a boolean question, taking an object variable as its parameter, and answering `true` if the object has been allocated. By using `some` where appropriate, you can make your code more readable and English-like:

*Example: Testing Object Allocation Using the **some** Operator*

```
if some button then
	{operations involving button}
end;
```

CHAPTER 10

Introduction to Object-Oriented Programming

This chapter introduces the basic elements of OMAR's object-oriented features.

The basic idea behind object-oriented programming (OOP) is that, instead of having the data and instructions for each conceptual entity spread throughout the program, or grouped together simply by convention, there ought to be an explicit, structured method of grouping together data and instructions that belong to a particular idea. In OMAR, as in other object-oriented languages, you can package data and instructions in conceptual units called *classes*. Each class serves as one complete computational description of an idea.

Class Components: Methods & Members

The two main components of a class are the data and the instructions that are used to represent an object. The data consists of a number of data fields, just like in a structure. The data fields that belong to a class are known as its *members*. The instructions that belong to a class are packaged as a series of procedures that only objects of that particular class may invoke. The procedures that belong to a class are known as its *methods*.

Class Declarations: Interface & Implementation

Class declarations can be fairly complex constructions, since they may be composed of a number of members and methods. When using a particular class, you ought to be able to view easily the information that tells how the class is to be used and skip the implementation details of the class. This idea should already be familiar from the procedure declarations that we have been using. You can use a procedure just by knowing the name and parameters that the procedure accepts, without knowing the details of the implementation. In a similar way, the class declaration lists the names and parameters of all available methods at the top, without including their implementations. The complete declarations of all methods including implementations are given in the last section of the class declaration, where all of the inner workings and details of the class must be defined.

Note that this organization is different from Java, where all of the code for the class is bunched together in once place, and you have to search through all of the code in the class to find which things are visible from outside the class and which things are merely private implementation details. Sometimes this shortcoming is alleviated by having sophisticated code development environments that let you hide the method implementations unless you need to see them. However, it is generally better to have this structure imposed by the language itself. In C++, the class interface is brought together in one place, as in OMAR, but the implementation may be scattered about, or even in multiple files.

Figure 10-1: The Basic Class Declaration

```
subject <name>
does
      <methods>              // The interface portion:
has                          // introduces the
      <members>              // methods and members.
is
      <implementation>
end;
```

```
subject circle
does                                    // Methods:
      verb print;
      verb move
            to vector location;
      end;
      scalar question circumference;
      scalar question area;
has                                     // Members:
      vector center = <0 0 0>;
      public scalar radius = 1;
is                                      // Implementation:
      verb print is
            write "circle with", ;
            write "     center = ", center, ;
            write "     radius = ", radius, ;
            write "     circumference = ", circumference, ;
            write "     area = ", area, ;
      end; // print

      verb move
            to vector location;
      is
            center = location;
      end; // move

      scalar question circumference is
            answer 2 * (3.14159) * radius;
      end; // circumference

      scalar question area is
            answer 3.14159 * radius * radius;
      end; // area
end; // circle
```

Class Instances: Objects

The class defines everything that the computer needs to know about a certain concept. The class doesn't actually do anything by itself, but merely defines how some kind of entity is to be represented inside the computer, what operations can be performed on that entity, and what, exactly, each of these operations will do when they are performed upon the entity. In order to actually do something with a class, you must make an ***instance*** of the class. This is much like creating a new structure variable.

An instance of a class is usually called an *object*. An object contains a set of data as defined by the class to represent the conceptual entity that is described by the class. Each time you create a new object, a completely new set of data is created. The class itself is like a template that describes how to create these collections of data, or objects.

Example: Creating Instances of a Class

```
circle type circle1;
circle type circle2, circle3;
```

Invoking an Object's Methods

In OMAR, to invoke an object's method, you simply state the name of the object followed by the method name and the method's parameters, if any. Note that the dot operator, or any other operator, is not required as it is in Java or C++. The name of the method simply follows the name of the object. In OMAR, when one of these objects is used in conjunction with one of its methods, then it is known as a *subject* because from the standpoint of English grammar, this is a more accurate description of its role in the statement.

Example: Invoking Methods: the Subject-Verb Format

```
circle type circle1;

circle1 print;                          // Call a subject's verb methods
write "area = ", circle1 area, ;        // Call a subject's question methods
```

If you are used to programming in a non-object-oriented language such as Pascal or C, it may seem strange that the circle's method, area, is able to say anything at all about the circle since it has no parameters. In fact, the method does have a parameter: an invisible parameter! Whenever a method is called by a subject in this way, the subject implicitly becomes the first parameter to the method call. All normal procedures listed as methods of a particular class have this implicit object parameter and may access the object's members. This will be covered more later in the section on objective methods.

Figure 10-2: Comparing OMAR with English

Grammatical Structure	English	OMAR
Verb	Stop!	quit;
Verb Object(s)	Destroy the document.	free hashtable;
Subject Verb	Fred writes.	hashtable initialize;
Subject Verb Object(s)	Fred writes documents.	hashtable enter thing as "Fred";

Accessing an Object's Members

An object's members are much like the data fields of a structure. To access an object's members, you use the same syntax as is used to access a structure's fields: the object name followed by the **'s** operator followed by the name of the member. Most well-designed classes require invoking the class methods to manipulate its data instead of allowing direct access to its members. There may be instances, however, when you want to access the members of an object directly.

Example: Accessing Members

```
circle type circle1;

circle1's radius = 10;
write "radius = ", circle1's radius, ;
```

From outside of a class declaration, an object seems a lot like a structure. Inside of the class method declarations, however, you may have noticed that there is something strange going on. Take a look at the implementation section of the circle class. Note that inside of a method declaration, the members of that class are referred to directly without any reference to the object to which they belong.

Example: Accessing the Members of an Object Inside of the Class Declaration

```
verb print is
    write "circle with", ;
    write "    center = ", center, ;                  // Center of what?
    write "    radius = ", radius, ;                  // Radius of what?
    write "    circumference = ", circumference, ;    // Circumference of what?
    write "    area = ", area, ;                      // Area of what?
end; // print
```

How does the compiler know what object to get these members from if there are no parameters to the method? The culprit here is once again the invisible parameter that was mentioned above. Remember that when a method is called, it is called from an object that becomes an invisible parameter to the method. So, the above code is actually compiled as:

Example: Making the Implicit Parameter in the Code Above Explicit

```
verb print
    circle type circle;           // Invisible implicit circle parameter
is
    write "circle with", ;
    write "    center = ", circle's center, ;                 // Member of invisible circle parameter
    write "    radius = ", circle's radius, ;                 // Member of invisible circle parameter
    write "    circumference = ", circle circumference, ;     // Call method with invisible parameter
    write "    area = ", circle area, ;                       // Call method with invisible parameter
end; // print
```

No this Parameter

OMAR's invisible parameter is very similar to the implicit this parameter in Java and C++. In OMAR, however, instead of making up an arbitrary name for the invisible parameter, it is simply given the name of the class, so if the class name is circle, then the parameter is a circle type circle. If the class name were square, then there would be a square type square parameter in every normal method declaration of the square class.

Constructors

In the example above, creating a new instance of the circle class was a simple matter of stating the type name followed by the instance name of the new object. For more complex objects, however, it might be desirable to perform some kind of initialization procedure every time a new instance is created. This initialization procedure is called a *constructor*. A constructor method is identified by the reserved name new. When you name a method in a class new, that method is invoked every time an instance of the class is created.

Listing 10-1: A Class with a Constructor

```
do test;

subject thing does
        verb new;                               // constructor
has
        public integer id;
is
        verb new is
                static integer counter = 0;
                counter = itself + 1;
                id = counter;
        end; // new
end; // thing

verb test is
        thing type thing1, thing2;

        write "thing1's id = ", thing1's id, ;          // Should write out "1"
        write "thing2's id = ", thing2's id, ;          // Should write out "2"
end; // test
```

Often, when you define a constructor, you find that some extra information must be provided to the constructor in the form of parameters. In this case, the syntax for the object declarations requires that the parameters be given immediately following the name of the new object instance variable. It's sort of like a variable declaration and method call combined into one.

Example: A Class with a Parameterized Constructor

```
subject circle
does
	// Methods
	//
	verb new
		at vector center = <0 0 0>;
	with
		scalar radius = 1;
	end;
has
	// Members
	//
	vector center = <0 0 0>;
	scalar radius = 1;
is
	// Implementation
	//
	verb new
		at vector center = <0 0 0>;
	with
		scalar radius = 1;
	is
		circle's center = center;
		circle's radius = radius;
	end; // new
end; // thing
```

Example: Instantiating Objects with a Parameterized Constructor

```
verb test is
	circle type circle1;                    // Use default values for constructor parameters
	circle type circle2 at <0 0 1>;         // Set first constructor parameter
	circle type circle3 at <0 0 1> with     // Set both constructor parameters
		radius = 5;
	end;
end; // test
```

If a class has a constructor that has mandatory parameters defined, then wherever an object of the class is created, appropriate parameter values *must* follow to satisfy the needs of the constructor. In the example above, if the circle class's constructor had mandatory parameters for the location and radius of the circle, then you could not instantiate a circle without including expressions for these parameters.

Destructors

Sometimes you may have a class of object for which you wish to perform some type of cleanup operations when an instance of the class is destroyed. For most normal objects, any necessary cleanup operations are carried out by the garbage collector, which frees up memory for any sub-objects that are used by the freed object. Certain other types of objects—for example, files or network sockets—may contain resources for which some special type of finalization is required when these objects are freed.

For these types of objects, you can provide a special method, known as a *destructor*, that is invoked on an object right before it is recycled. Just as constructors are methods that are always given the name new, destructors are methods named free. Destructors are invoked by the garbage collector, which is periodically invoked to sweep through the memory and recycle any unused objects. The exact time at which a destructor is invoked is not normally determined by the code because it depends upon when the garbage collector kicks in to free up memory. If you explicitly invoke the garbage collector, which you can do by using a renew statement, then you are insured that the destructor method will be called immediately.

Listing 10-2: Invoking the Garbage Collector and Destructors

```
do test;

subject thing
does
        verb new;                       // constructor
        verb free;                      // destructor
has
        public integer index = 0;
is
        integer counter = 0;            // counter of number of objects allocated

        verb new is
                counter = itself + 1;
                write "constructing thing #", counter, ;
                index = counter;
        end; // new

        verb free is
                write "destructing thing #", index, ;
                counter = itself - 1;
        end; // free

        write "initializing class", ;
        write "initial counter = ", counter, ;
end; // thing
```

Listing 10-2: Invoking the Garbage Collector and Destructors (Continued)

```
verb test is
        thing type thing is none;
        integer counter = 0;

        while true do
                new thing;
                thing is none;
                counter = itself + 1;

                // collect garbage every 10 iterations
                //
                if counter = 10 then
                        renew;      // do garbage collection
                        counter = 0;
                end;
        end;
end; // test
```

CHAPTER 11

Intermediate Object-Oriented Programming

This section examines two of the most important features of object-oriented programming: *inheritance* and *encapsulation.*

Code Reuse and Robustness

In large scale programming projects, you often run into two major problems. The first is that it is rather difficult to modify and reuse existing code. You can find existing code for doing just about every imaginable thing. Programs, however, are very complex, intricately connected things, like the spaghetti wiring inside of an old stereo system. If you want a particular set of features in your program, you often find that it is easier (and less error-prone and dangerous) to rewrite everything from scratch, rather than modify an existing program. Obviously, this is one reason why software creation is so time consuming and expensive.

The second problem is that it is difficult to incorporate existing code in a way that is safe and robust. Often, for example, you have situations where a set of variables must be initialized correctly before you can safely use a piece of code, or certain procedure calls must be made in a precise order, or certain variables may not be modified explicitly without breaking the code. OMAR's encapsulation primitives make it possible to create robust, bulletproof classes that can be used without being concerned about these issues. Object-oriented programming is an effort to make code more robust and more reusable through inheritance and encapsulation.

Inheritance

The basic idea behind inheritance is that you can create classes that are derived from existing classes, instead of writing each one from scratch. These derived classes inherit the functionality of the class from which they are derived (the parent class), but they may also extend or override the functionality of their parent class. If a class extends another class, that means that it has additional data and/or methods that the parent does not have. If a class overrides its parent class's methods, that means that it has defined methods that take the place of methods in the parent class and has new, different behavior.

Experienced object-oriented programmers know that one of the most difficult things about designing well-structured code is building these class hierarchies, so that each class neither does too much nor does too little. If the class does too much, then it is overly difficult and cumbersome to use. It may be inefficient because it contains harmless, but unneeded functionality. If the class does too little, then you end up with lots of little classes, each only slightly different and hard to distinguish from others, and the program once again becomes difficult to understand.

Extending a Class

To add new features to an existing class, you create a new class that extends the features of the old class. This new class is known as a *subclass* because it is a more specific version of the more general class that it was derived from. This terminology may be a little confusing because a subclass actually has a superset of the functionality of its parent class. A subclass is less general but has more 'stuff' than its parent class. To create a new subclass, use the following format:

Figure 11-1: The Basic Subclass Declaration

```
subject <name>
extends                         // Class which this class is
      <parent class>            // derived from
does
      <methods>                 // The interface portion:
has                             // methods and members
      <members>                 //
is
      <implementation>
end;
```

All classes are considered to be implicitly derived from a base class, the subject class. If you don't have an extends clause in your class declaration, there is an invisible extends subject clause that is implicitly added for you. Note that when you extend a class, you inherit not only the functionality of the superclass, but you also inherit the functionality of every superclass all the way to the root. The way to think about it is that if class B extends class A, then an instance of B IS an instance of A. For example, if you have an employee class that extends the class person then each employee IS a person. Likewise, if you extend the employee class by an executive class, then each executive is an employee and is also a person.

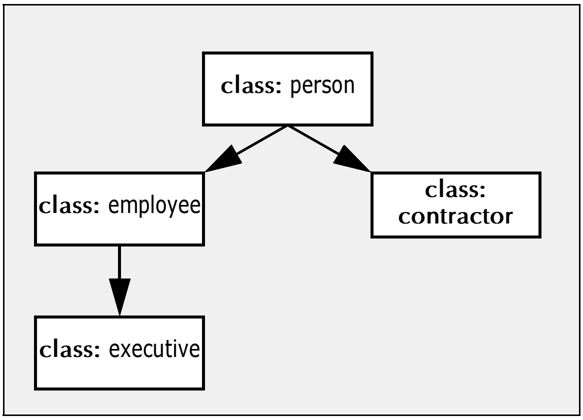
Example: Extending a Class

```
subject person
does
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0;
        end; // new

        // Methods to retreive private information
        //
        integer question get_birthyear;
has
        char name[];
        integer birthyear;
is
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0;
        is
                write "new person", ;
                person's name = name;
                person's birthyear = birthyear;
        end; // new

        // Methods to retreive private information
        //
        integer question get_birthyear is
                answer birthyear;
        end; // get_birthyear
end; // person
```

```
subject employee
extends
        person
does
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0, salary = 0;
        end; // new

        // Methods to retreive private information
        //
        integer question get_wages;
has
        integer salary;
is
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0, salary = 0;
        is
                // Call superclass constructor
                //
                person new named name with
                        birthyear = static birthyear;
                end;

                // Initialize employee members
                //
                write "new employee", ;
                employee's salary = salary;
        end; // new

        // Methods to retreive private information
        //
        integer question get_wages is
                answer salary;
        end; // get_wages
end; // employee
```

```
subject executive
extends
        employee
does
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0, salary = 0, stock_options = 0;
        end; // new

        // Methods to retreive private information
        //
        integer question get_wages;
        integer question get_perks;
has
        integer stock_options;
is
        // Constructor
        //
        verb new
                named char name[];
        with
                integer birthyear = 0, salary = 0, stock_options = 0;
        is
                // Call superclass constructor
                //
                employee new named name with
                        birthyear = static birthyear;
                        salary = static salary;
                end;

                // Initialize executive members
                //
                write "new executive", ;
                executive's stock_options = stock_options;
        end; // new

        // Methods to retreive private information
        //
        integer question get_wages is
                answer salary + stock_options;
        end; // get_wages

        integer question get_perks is
                answer stock_options;
        end; // get_perks
end; // executive
```

Figure 11-2: A Simple Class Hierarchy



In the example above, you have a simplified class hierarchy that might be used to represent a corporate hierarchy (for a payroll application, perhaps). The classes employee and contractor inherit the members and methods of class person. That means that an employee or contractor can do everything that a person can do (such as have a name, birthday, social security number etc.). In addition, the employee and contractor classes have additional capabilities that normal persons do not have (getting wages, for example). Members of the executive class have all of these capabilities plus more. For example, only executives can get perks and have stock options.

Constructor Chaining

When you create a new class by extending an existing class, it inherits all of the members of that existing class and therefore, you must make sure that the integrity of those members is maintained. This leads to the requirement that constructors must be called not only to initialize the members of objects of their own class, but also to initialize the members of objects belonging to any classes that are derived from this class. This is known as *constructor chaining.*

Example: Constructor Chaining

```
subject subclass
extends
      superclass                // This is a class with a constructor defined
does
      verb new;                 // This subclass must have its own constructor defined since
has                             // its superclass has a constructor defined
      integer stuff;
is
      verb new is               // The first line of the subclass's constructor must be a call
            superclass new;     // to its superclass's constructor to initialize its inherited members
            stuff = 0;          // After, you can initialize the subclass's noninherited members
      end;
end; // subclass
```

The way that constructor chaining is enforced is to require the first line of any constructor that belongs to a class with a superclass constructor to call that superclass constructor before doing anything else. In the example hierarchy above, since an executive is also an employee, which is also a person, whenever the executive's constructor is called, it must first call the employee constructor which must first call the person constructor.

Overriding Methods

Sometimes, when you create a new subclass, you find that an inherited method does not work quite as you want it to for the new class. In this case, you can provide a new definition for the inherited method that is more specific to the new subclass. This process of substituting a new method for one that has been inherited is known as *overriding*.

To override an inherited method, all you have to do is to define a new method for the subclass that has the same name and parameters as an inherited method. When an instance of this derived class calls the method of this name, the new method will be called instead of the inherited one.

In the example above, the executive class inherits the method get_wages from the employee class. Since the executive is also an employee, it makes sense that the executive should also be able to have this procedure. When you further defined the executive, however, you find that you can provide a more precise definition of this procedure that is specific to the executive class. In this case, an executive should also add in stock options when reporting wages. So, you can see how to use overriding to provide more specific versions of methods for subclasses when the more general inherited methods are not as precise.

Example: Using Method Overriding

```
executive type president named "bob" with                 // Instances of class executive
        birthyear = 1942;
        salary = 80000;
        stock_options = 100000;
end;
employee type manager named "sally" with                  // Instances of class employee
        birthyear = 1972;
        salary = 20000;
end;

write "Sally's wages = ", manager get_wages, ;            // Calls employee's get_wages method
write "Bob's wages = ", president get_wages, ;            // Call executive's get_waves method instead
                                                          // of the employee's get_wages method
write "Bob's perks = ", president get_perks, ;
write "Sally's perks = ", manager get_perks, ;            // Compile error - employees have no perks!!!
```

Dynamic Binding

When you create a new class using inheritance, you expect that if you have provided this class with a replacement method to override an existing method from its parent class, then the new method will be called instead of the old one. Dynamic binding is the technique that is used internally to make this happen. Using this technique of overriding methods and using dynamic binding to select the correct method to use, you can write very generic code and then let the objects themselves decide the appropriate method implementation to use.

Using the class hierarchy above, suppose that you had to write a program to compute the payroll from a list of employees. You could have a procedure that needs to get the wages of each employee. Remember that employees and executives have different methods for computing this, but that the executive method overrides the employee method. This means that all you have to do is to go down the list and call get_wages for each employee and this command is automatically interpreted differently, depending upon whether it was called from an employee or an executive. You, as the programmer, do not have to write special code everywhere that checks the type of the employee and calls the appropriate method depending upon the type because this is automatically taken care of for you in a way that is intuitive and elegant.

Example: Dynamic Binding

```
employee type secretary named "suzie", salesperson named "maggie", manager named "joe";
executive type president named "leo", ceo named "barb";
integer sum = 0;

// Table of references to employees
employee type staff[] is [manager president secretary ceo salesperson];

// Code to sum wages of staff
for each employee type employee in staff do
        // Here, get_wages will either execute the employee's implementation or
        // the executive's implementation depending upon the type of the employee
        //
        sum = itself + employee get_wages;
end;
```

Variable Shadowing

Each new class that you declare may have its own set of members that can be given any valid names that you choose. A class that extends another class also inherits the members of the parent class, so this presents a problem. If the new class declares a member of the same name as a member of the parent class, then the new member is said to *shadow* the inherited member. You can no longer refer to the member of the parent object just by stating its name, because the name has been appropriated by the new member.

You can, however, refer to shadowed members by giving the full name of the member. Recall that when referring to a variable inside of a class, if the variable name is not found in the method's local variables, then it is assumed implicitly to be a member of the current class. To specify that what you really want is the member of the class that you are extending, you just explicitly dereference the class that you wish to refer to.

Example: Accessing Shadowed Members

```
subject person
does
	verb print;
has
	char name[];
	integer id;
is
	verb print is
		write name, " with social security number = ", id, ;
	end; // print
end; // person

subject employee
extends
	person
does
	verb print;
has
	integer id;				// Shadows person's id
is
	verb print is
		// Implicitly refers to person's name (because there is no employee name member)
		write name;

		// Explicitly refers to person's id
		write " with social security number = ", person's id, ;

		// Implicitly refers to employee's id
		write "and employee identification number = ", id, ;
	end; // print
end; // employee
```

Method Overriding Vs. Member Shadowing

When you declare a new class, you may declare new methods with the same names as methods of the parent class. This is called overriding. You can also create new members with the same names as members of the parent class. This is called shadowing. Since these two situations arise from similar circumstances, it may be easy to get them confused. However, method overriding and member shadowing are resolved in distinctly different ways, so it is important to understand the difference.

When methods are overridden, the way that the correct method is determined is at run time depending upon the type of the object (this mechanism is called dynamic binding). In the case of shadowed variables, however, the determination of exactly which variable is referenced is determined completely at compile time. This is the crucial difference. Variables are determined at compile time and methods are determined at run time.

Listing 11-1: Method Overriding Vs. Member Shadowing

```
do test;

subject A
does
        integer question get_value;
has
        public integer i = 1;
is
        integer question get_value is
                answer i;
        end; // get_value
end; // subject A

subject B
extends
        A
does
        integer question get_value;
has
        public integer i = 2;
is
        integer question get_value is
                answer i;
        end; // get_value
end; // subject B

verb test
is
        A type A is none;
        B type B;

        write "B's i = ", B's i, ;              // References B's i, prints 2
        write "B's value = ", B get_value, ;    // Calls B's get_value, prints 2

        A is B;
        write "A's i = ", A's i, ;              // Still references A's i, prints 1
        write "A's value = ", A get_value, ;    // Calls B's get_value, prints 2
end; // test
```

Type Casting

Sometimes it is desirable to convert between objects of related types. This is useful when you need a piece of code to handle the most general possible case for a family of related types of object. For example, you could write a piece of code that stored objects of your `person` class into a directory. When you entered objects into this directory, you could enter `person`s, but you could also enter employees or executives because they are, after all, persons. When you removed the persons from the directory, however, you would have to explicitly convert them to the appropriate type, because, though all executives are persons, for example, not all persons are executives.

When you cast one type to another type, a check is made to make sure that the types are actually compatible. If this is not true, then a run-time error occurs. An object may only be cast to a related type in its class hierarchy. If an attempt is made to cast a type to a totally unrelated type, then a compile-time error is issued, since this may never be performed. A type cast is formed by placing the name of the type that you wish to cast to, followed by the keyword type, in front of the name of the object that you wish to cast. This returns an object of the desired type or issues a run-time error if the object is not of the desired type at the time the cast is performed.

Example: Implicit & Explicit Casting between Related Types

```
person type person is none;
employee type manager named "garth";
executive type ceo named "bill";
circle type circle is none;

person is manager;                    // OK because all employees are persons
person is ceo;                        // OK because executives are persons

ceo is person;                        // Compile error- a person is not necessarily
                                      // an executive - a cast must be used to check

ceo is executive type person;         // Cast is essential because all persons are not executives
                                      // This may cause a run time error if the person referred to
                                      // is not actually an executive - in this case it won't because
                                      // from the lines above, you can see that person refers to
                                      // an executive.

circle is circle type person;         // Compile error - a person may never be cast to type
                                      // circle since they are unrelated.
```

Type Querying

Sometimes, instead of attempting to blindly cast one object type to another object type, you would like to test the type of the object first to see if it is suitable for the operation you're attempting to perform.

When querying the type of an object, you can only test to see if an object is of a related class. If you try to test for an object being of a totally unrelated class, then a compile-time error results because this can never be true. For example, you can test to see whether an object of type person is of type executive because sometimes, a person may be also be an executive. If you try to test whether a person is of type circle, however, a compile-time error results since this can never be the case.

A type query returns a boolean value and is formed by placing the name of the type to query for, followed by the keyword type and the name of the object that you wish to query. The type querying function is very much like the instanceof operator in Java.

Example: Type Querying

```
person type worker is none;
employee type manager;
executive type ceo;

worker is manager;                    // Ok because employees are persons
worker is ceo;                        // Ok because executives are persons

if executive type worker then
      // Cast person to executive before calling executive method on person
      //
      write "executive perks = ", executive type worker get_perks, ;
end;

if circle type worker then                                // Compile-time type querying error
      write "person's radius = ", circle type worker get_radius, ;// Compile-time type casting error
end;
```

Encapsulation

One major source of bugs in programming is inadvertent misuse of variables. The basic aim of *encapsulation* is to shield data belonging to an object from unwanted tampering from other areas of the program. Since a class description knows which methods belong to the class, it can grant access to the object's private variables only to those methods that have a legitimate right to change those variables. This makes it possible to create solid, 'bulletproof' objects that may have arbitrarily complex inner workings but are immune to being broken by accidentally or purposefully changing variables that are only intended to be used in certain ways.

For example, suppose you have a symbol table object that has methods for storing and retrieving names and for telling you how many names it contains. In order to keep track of the number of names in the table, the table object needs a counter member that is incremented by one each time you add a name to the table. The only time you want this counter changed is when a name is added to the table; otherwise, this variable should be left alone. By creating a class, you shield this variable from unwanted tampering and don't have to worry about the possibility of another part of the program changing this counter by mistake. This is what encapsulation is all about.

Public & Protected Members

Each class has a collection of members that are used internally by the methods of the class, but may or may not be intended to be seen outside of the class.

By default, the members of a class are hidden from tampering from outside of the class because that is safer. These members are said to be *protected*. If you wish to grant access to a class's members from outside, you must designate those members as *public*. Public members may always be accessed and are therefore similar to the fields of a structure, which are always implicitly public.

Example: A Class with Public and Protected Members

```
subject circle
does
        verb new
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        end;
        scalar question get_area;
has
        public vector location;                 // This is a public member
        scalar radius, radius_squared;          // These are protected members (by default)
is
        verb new
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        is
                circle's location = location;
                circle's radius = radius;
                circle's radius_squared = radius * radius;
        end; // new

        scalar question get_area is
                answer (3.14159) * radius_squared;
        end; // get_area
end; // circle
```

Example: Accessing Members of a Class with Protected Field

```
circle type circle at <0 0 1> with
        radius = 10;
end;
circle's location = <0 0 10>; // Move circle
circle's radius = 20;         // Compile error -
                              // The circle's radius can not be accessed like this because it is protected.
                              // The reason for this member being protected is that whenever radius is
                              // changed, radius_squared should be changed also to keep the members
                              // of the circle consistent.
```

Although the radius member of the circle class may not be accessed from any arbitrary location in the program like the public members, there is one place outside of the class declaration where it can be accessed: inside the declarations of subclasses of the circle class. The scope of a protected member includes all classes that are derived from the class in which the protected member was declared.

Example: Accessing the Protected Field of a Parent Class

```
subject ellipse
extends
        circle
does
        verb new
                at vector location = <0 0 0>;
        with
                scalar major_radius = 1, minor_radius = 1;
        end;
        scalar question get_area;
has
        scalar minor_radius;
is
        verb new
                at vector location = <0 0 0>;
        with
                scalar major_radius = 1, minor_radius = 1;
        is
                ellipse's location = location;              // Access public member of circle class
                ellipse's radius = major_radius;            // Access protected member of circle class
                ellipse's radius_squared = radius * radius; // Access protected member of circle class
                ellipse's minor_radius = minor_radius;      // Access protected member of ellipse class
        end; // new

        scalar question get_area is
                answer (3.14159) * radius * minor_radius;
        end; // get_area
end; // ellipse
```

Private Members

In the example above, you can see how the radius member is hidden from general access but is allowed to be accessed from derivatives of the circle class. There are some cases, however, where you may have members that are only relevant to a particular class and will not be relevant even to subclasses that are derived from that class.

Imagine that you decide to precompute the circle area and store it in a member of circle. Although the circle radius is relevant to ellipses, the circle area is not really of any use to the ellipse class, so you might as well make it private to the circle class. Private members of a class are only visible inside of the declaration of their own class and nowhere else.

Figure 11-3: The Basic Subclass Declaration

```
subject <name>
extends
        <parent class>
does
        <methods>
has
        <public and protected members>
private
        <private members>
is
        <implementation>
end;
```

Note that the syntax for declaring private members is slightly different from the syntax for declaring public or protected members. The private members are set aside from the public and protected members in their own section. The reason for this is that there are no circumstances in which a person who is using a class needs to know about the private members of that class. If a person is simply using a class, then they need to know about only the public members and if a person is extending a class, then they need to know about the public and protected members. Only the original implementer of the class needs to know about the private members of that class, however, so these members are set apart from the true interface portion of the class.

Example: A Class with Public, Protected, and Private Methods

```
subject circle
does
        verb new
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        end;
        scalar question get_area;
has
        public vector location;                 // This is a public member
        scalar radius, radius_squared;          // These are protected members (by default)
private
        scalar area;                            // This is a private member
is
        verb new
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        is
                circle's location = location;
                circle's radius = radius;
                circle's radius_squared = radius * radius;
                circle's area = (3.14159) * radius_squared;
        end; // new
        scalar question get_area is
                answer circle's area;
        end; // get_area
end; // circle
```

Public & Protected Methods

Although it is generally less useful to restrict access to a class's methods than it is to restrict access to a class's members, you might occasionally find it convenient to do so. For example, sometimes a class provides special access functions that are useful to subclasses but that are not generally to be used by users of the class. Access to a class's methods is restricted in a way similar to class member restriction: by placing a visibility modifier before the declaration.

By default, the methods of a class are public. You may make them protected by placing the keyword protected before their declarations in the interface. Note that this default is the opposite of the default for members, which are protected by default. This makes sense because it is generally less important to protect methods than to protect members and also because methods are most often public, since they are the preferred way to interact with a class rather than to access the members.

Example: A Class with Protected Methods

```
subject circle
does
        verb new                                // A public method (by default).
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        end;

        // A protected method (note the 'protected' keyword).
        // This method may only be called inside of methods
        // of this class or of a class derived from this class.
        protected scalar question get_area;
has
        public vector location;                 // This is a public member
        scalar radius, radius_squared;          // These are protected members (by default)
is
        verb new
                at vector location = <0 0 0>;
        with
                scalar radius = 1;
        is
                circle's location = location;
                circle's radius = radius;
                circle's radius_squared = radius * radius;
        end; // new
```

<u>*Example: A Class with Protected Methods (Continued)*</u>

```
        scalar question get_area is
              answer (3.14159) * radius_squared;
        end; // get_area
  end; // circle
```

Private Methods

Private methods are methods that are used only in the implementation of a class and are of no concern to users of the class or even to subclasses of this class. Private methods are distinguished from public or protected methods of the class by being defined solely inside of the implementation portion of the class declaration, and have no forward declaration in the interface portion.

<u>*Example: A Class with a Private Member*</u>

```
subject circle
does
      verb new                                // A public method (by default)
            at vector location = <0 0 0>;
      with
            scalar radius = 1;
      end;
has
      public vector location;                 // This is a public member
      scalar radius, radius_squared;          // These are protected members (by default)
is
      verb new
            at vector location =<0 0 0>;
      with
            scalar radius = 1;
      is
            circle's location = location;
            circle's radius = radius;
            circle's radius_squared = radius * radius;
      end; // new

      scalar question get_area                // This is a private method because it does not
      is                                      // have a forward declaration in the interface
            answer (3.14159) * radius_squared;   // portion listed at the top of the class declaration
      end; // get_area
end; // circle
```

Summary of Access Levels

As a general rule, you should use the highest level of protection for a member or method that you can, without restricting the class's functionality. Here is a table summarizing the visibility of a class's methods and members under differing circumstances:

Table 11-1: Summary of Access Levels

Situation	Public	Protected	Private
visible inside of its own class	Yes	Yes	Yes
visible inside of a subclass	Yes	Yes	No
visible anywhere in the program	Yes	No	No

CHAPTER 12 Advanced Object-Oriented Programming

Objective Methods

Sometimes there are situations where you have methods belonging to a class that are not specific to a particular object. In the following example, the `circle` class includes a counter for the number of circles that have been created, and a method for returning the value of this counter. Since this method returns the value of a property of the class as a whole, and not of a particular object, then you have no need to call the method from a particular object. If you define the method as a normal method, then, as previously described, an implicit object reference would be passed to the method which would just be ignored. Although no harm would be done aside from the extra overhead needed to pass the reference, it is not really a good idea because it would make the method less versatile since you would always need to call it from a object.

The best solution is to designate these special case methods as taking no implicit parameters even though they belong to a particular class. These methods are referred to in Java as either class or static methods. In OMAR, class methods are called *objective* methods and are designated by preceding them with the keyword `objective` because when they are called, they require no subject instance. By contrast, you can think of the normal methods of a class as subjective methods because they depend upon the particular subject instance with which they are called.

The important thing to remember is that since objective methods do not take an object as an implicit parameter like normal methods, they may not refer to any of the class's members. They may, however, refer to any class variables listed in the implementation section of the class since these are not specific to each individual object but instead are instantiated once just like normal variables.

Example: A Class with Objective Methods

```
subject circle
does
	verb new with
		scalar radius = 1;
	end;
	objective integer question get_number;
	objective verb write_number;
has
	scalar radius;
is
	integer number_of_circles = 0;

	verb new with
		scalar radius = 1;
	is
		circle's radius = radius;
		number_of_circles = itself + 1;
	end; // new

	objective integer question get_number is
		answer number_of_circles;
	end; // get_number

	objective verb write_number is
		write "number of circles = ", number_of_circles, ;
		write "radius of circle = ", radius, ;		// Error - cannot refer to members of the object
	end; // write_number
end; // circle
```

Since these objective methods are not specific to an object, the question is: how are they called? To call an objective method, you must precede the method name with the name of the class, followed by the `'s` operator, instead of just the name of an object. The full name of any type consists of the name given to the type in the declaration followed by the keyword `type` to signify that it is the name of a type and not a variable.

Example: Calling Objective Methods

```
integer i;

circle type's write_number;		// calling a verb method
i = circle type's get_number;		// calling a question method
```

Reference Methods

Normally, an object reference is passed to each method in a class, which enables the method to access the members of the object. What if you want the method to change not just the object's members, but the actual object reference from which the method was called? If you simply change the reference that is passed to the method, it will only change the local copy of the reference that is used inside of the method and will not actually change the outside object reference from which the method was called. You need to use a *reference method*. A reference method has a link to the actual reference from which the method is called, instead of simply a copy of it. Using this external link, changes to the internal object reference are reflected outside of the method in the variable from which the method was called. This mechanism effectively makes the implicit object variable a reference parameter.

Example: Using a Reference Method

```
do test;

subject thing
does
        reference verb nuke;
        reference verb swap
                thing type reference thing2;
        end;
has
        public integer i;
is
        reference verb nuke is
                thing is none;
        end; // nuke

        reference verb swap
                thing type reference thing2;
        is
                thing type temp is thing;

                thing is thing2;
                thing2 is temp;
        end; // swap
end; // thing

verb test is
        thing type thing, thing1, thing2;

        // use reference method to set reference to none
        thing nuke;

        // thing should be none after above method call
        if thing is none then
                write "unallocated thing", ;
        else
                write "allocated thing", ;
        end;

        // use reference method to swap references
        thing1's i = 10;
        thing2's i = 20;
        write "thing1, thing2 = ", thing1's i, ", ", thing2's i, ;
        thing1 swap thing2;
        write "thing1, thing2 = ", thing1's i, ", ", thing2's i, ;
end; // test
```

Disabling Dynamic Binding

There are occasional instances when you need to be able to deactivate the dynamic binding mechanism so that you can explicitly tell the compiler which method implementation to invoke. For example, you might override a method with a new method, but still wish to call the old method from within the new method. In this case, any references to the old method will automatically refer to the new method because of the dynamic binding mechanism and infinite recursion will result. Static binding lets you bypass this mechanism to say exactly which method implementation to call. To force static binding, place the keyword static between the object and its method call.

Example: Disabling Dynamic Binding

```
subject figure
does
        verb init;
has
        vector color;
is
        verb init is
                color = <1 1 1>;
        end; // init
end; // figure
```

```
subject circle
extends
        figure
does
        verb init;
has
        scalar radius;
is
        verb init is
                circle's radius = 1;
                figure static init;        // without the 'static' keyword to force static binding, this
                                           // would call circle's init which would result in infinite recursion
        end; // init
end; // circle
```

Final Methods

Some methods have definitive implementations and should not be overridden. For example, if you have something like a vector or matrix class, then there is only one mathematical definition for the operations that may be performed on this kind of entity, so it is unlikely that you will ever want to override these methods.

In cases where you know that a method will not be overridden, there are certain optimizations that may be performed on the method calls. In particular, static binding may be used instead of dynamic binding which results in slightly speedier method invocation. In order to tell the compiler that a method will not be overridden, label the method as final. The final keyword comes directly before the name of the method. For example:

Example: Final Method Declaration

```
subject circle
does
	final scalar question get_area;
has
	scalar radius;
is
	scalar question get_area is
		answer (3.14159) * radius * radius;
	end; // get_area
end; // circle
```

Final Classes

Sometimes an entire class is the complete and definitive description of something and should not be extended by any subclasses. For example, if you have something like a vector or matrix class, then, because there is only one mathematical definition for these concepts, these classes need not be extended in any way. If an entire class is known to be the final definition, then every method in the class will be the final implementation. In this case, the class may be labelled as a final class and then every method in the class will be optimized as a final method.

Example: Final Class Declaration

```
final subject circle
does
	// all methods are implicitly final methods
	//
	scalar question get_area;
has
	scalar radius;
is
	scalar question get_area is
		answer (3.14159) * radius * radius;
	end; // get_area
end; // circle
```

Abstract Methods

Sometimes when you have a set of classes, you find that they share a common set of features. The correct thing to do in such a case is to factor out common members and methods into a base class from which the others inherit the common features. When the implementation of these common features is actually specific to each subclass, however, you can't really provide a valid implementation for the base class. In this case, you would like to specify an interface for these methods without providing an implementation with the understanding that any class that extends this class must provide its own implementation. Methods of this type are called *abstract methods* because they lack any implementation.

Abstract Classes

Any class that contains abstract methods must be labelled as abstract. Abstract classes may contain abstract methods and non-abstract methods. Abstract method declarations in a an abstract class's interface section are preceded by the keyword abstract. When a non-abstract class extends an abstract class, it must provide an implementation for each of the abstract methods.

Abstract classes are also different from normal classes because they may not be instantiated. It doesn't really make sense to create an instance of an abstract class because the class is not yet fully defined, since its methods have no implementations. You may, however, have references to an abstract class that can be directed to refer to instances of any non-abstract class that is derived from the abstract class. Abstract classes are meant to provide a mechanism for defining a specification for a set of features of a class without actually defining the details of the class.

Figure 12-1: An Abstract Class Declaration

```
abstract subject <name>
extends
	<parent class>
does
	<abstract and non-abstract methods>
has
	<members>
is
	<implementation of any non-abstract methods>
end;
```

Abstract classes might not have an implementation section. If all of an abstract class's methods are abstract, then there is no implementation section. If the abstract class contains some non-abstract methods, however, an implementation section is required to define the implementations of these methods.

For example, suppose you're creating a set of classes for defining geometric figures. You know that each figure must have methods for things like determining the area and circumference. Since these properties are unique to each kind of figure that you may choose to create, there is no general implementation of these methods for the class figure. This calls for an abstract class.

Example: An Abstract Class

```
abstract subject figure
does
	abstract scalar question get_area;
has
	const scalar pi = 3.14159;
end; // figure

subject square
extends
	figure
does
	scalar question get_area;
has
	scalar length = 1;
is
	scalar question get_area is
		answer length * length;
	end; // get_area
end; // square

subject circle
extends
	figure
does
	scalar question get_area;
has
	scalar radius = 1;
is
	scalar question get_area is
		answer pi * radius * radius;
	end; // get_area
end; // circle
```

Example: Instancing Subclasses of an Abstract Class

```
figure type figure is none;	// abstract class – no instances allowed
circle type circle;		// instances of non-abstract classes (circle and square)
square type square;		//	which extend the abstract class, figure

figure is circle;
figure is square;

write "figure's area = ", figure get_area, ;	// calling an abstract method
```

Interfaces

The ability of one class to extend another class allows new forms of expression in terms of specifying your data. Each class can only extend one other class, however. Sooner or later, however, you are going to run into the situation where you want a class to inherit functionality from two different sources. This presents a problem. If you were to allow a class to inherit methods and data from two different classes, it would bring up a number of issues. For example, when methods or members of the same name are inherited from both classes, how do you decide between them. Also, there are a number of implementation difficulties associated with what is called multiple inheritance. OMAR has chosen the same approach as Java in attempting to sidestep these issues.

There is a problem with inheriting methods and data from multiple sources, but it turns out that there is no problem with inheriting abstract methods or constants from multiple sources. Because of this implementation issue, OMAR has a special type of class, similar to an abstract class, which consists of only method interfaces (abstract methods) and constants. This is called an interface. An interface is declared similarly to a class.

Figure 12-2: An Interface Declaration

```
interface <name>
does
      <implicitly abstract methods>
has
      <constant members>
end;
```

Note that since all of the methods of an interface are implicitly abstract, there is never any need for an implementation section. Also, note that only constant members are allowed as members of an interface. When a class inherits from an interface, the syntax is similar to the mechanism for inheriting from a class, except instead of listing the interfaces in the `extends` clause with the parent class, you list them afterwards in a separate clause indicated by the keyword `implements`. Although only a single parent class may be listed after the keyword `extends`, multiple interfaces may be listed, separated by commas, in the implements clause.

Figure 12-3: A Class Inheriting from a Parent Class and Interfaces

```
subject <name>
extends
	<parent class>
implements
	<interfaces>
does
	<methods>
has
	<members>
is
	<implementation>
end;
```

Since interfaces only contain abstract methods, a class that implements an interface must provide its own implementation for each of these methods. You can also create references to interfaces, just like you can create references to abstract classes, even though you cannot create actual instances of these types. An interface is basically like a promise to implement a number of methods. Each class that implements an interface must carry out this promise and therefore is allowed to be considered as a subclass of this interface because it is guaranteed to be a superset of the interface's functionality.

Using Constants in Interfaces

Another useful application of interfaces is to package groups of constants. By putting the constants in an interface, you avoid the potential name clashes that you might have if you just had the constants declared as global variables. Since each class may implement as many interfaces as it likes, it can implement any number of these constant packages.

Example: Using Constants in Interfaces

```
interface math_constants
has
	const scalar pi = 3.1415926;
	const scalar e = 2.71828;
end; // math_constants

interface physics_constants
has
	const scalar planks_const = 6.626 * (10 ^ -34);
	const scalar boltzmanns_const = 1.381 * (10 ^ -23);
end; // physics_constants
```

```
subject thing
implements
        math_constants
is
        verb write_constants is
                // This constant can be accessed directly because this class
                // implements the math_constants interface
                //
                write "pi = ", pi, ;

                // This constant must be accessed through the name of
                // its interface because this class does not implement
                // that interface (physics_constants).
                //
                write "plank's constant = ", physics_constants type's planks_const, ;
        end; // write_constants
end; // thing
```

Static Initializers

You've seen how a class can specify a constructor method to initialize the members of an instance of a class whenever an instance is created. Sometimes, however, the class may have static data that needs to be initialized at the start of the program when the class declaration is encountered. The best way to accomplish this is with a static initializer. A static initializer is a section of code that is executed once, when the class is initialized at the start of the program. Static initializers are ideal for creating things like lookup tables that persist throughout the life of a class. The static initializer is a block of statements that is located after all other declarations in the class, so it is the very last section of the class declaration.

Listing 12-1: A Static Initializer

```
do test;

include "math.ores";

subject quiktrig
does
        objective scalar question quiksin
                scalar x;
        end;
is
        scalar table[1..360];

        objective scalar question quiksin
                scalar x;
        is
                answer table[trunc x mod 360];
        end; // quicksin

        // static initializer
        //
        for integer counter = 1 .. 360 do
                table[counter] = sin counter;
        end;
end; // quiktrig

verb test is
        write "quick sin of 30 = ", quiktrig type's quiksin 30, ;
        write "quick sin of 60 = ", quiktrig type's quiksin 60, ;
end; // test
```

Glossary

A

Algorithm

An algorithm is a set of instructions or steps that tell how to perform a certain task. Since computer programs are encoded as a series of explicitly defined, discrete steps, they are a good example of the use of algorithms. The term, *algorithm*, refers to the series of steps that are involved instead of the actual computer program itself. The computer program is a way of encoding the algorithm, although the algorithm could also be expressed in another computer programming language or a language such as English.

Argument

In a mathematical sense, *arguments* are the values passed into a function. For example, in the function call, sin (50), the argument of the function is 50. In OMAR, arguments to procedures are typically called *parameters*. *Program arguments* are the text commands passed into a program from a command line or HTML code.

B

Boolean

Boolean algebra is a system developed by George Boole to express logical relations. Boolean algebra takes as arguments Boolean values, which can be either true or false, and combines them using the operators, `not`, `and`, and `or` to yield Boolean results.

C

Conditional Statement

Conditional statements are statements that test if a certain condition is true and if so, performs some action. The test is done by evaluating a Boolean expression to yield a Boolean value and the statements are executed depending upon the value of the Boolean expression.

Complex Numbers

The complex number system was developed to provide for the represen-tation of the roots of negative numbers.

Typically, square roots are only defined for positive numbers, and the square root of a negative number is undefined because there are no real numbers which, when squared, yield a negative number. The complex number system introduces a new number, designated *i*, which is defined to be the square root of -1. Then, when you ask what the square root of -9 is, the answer is 3*i* because 3^2 is 9 and i^2 is -1, so $(3i)^2$ is -9.

All complex numbers may be expressed in the form (a + (b * ***i***)) where a is the *real* part and b is the *imaginary* part. This system may at first seem arbitrary and useless, but complex numbers find a number of uses in science and engineering and are the basis of the fractal images that come from the mathematical entity known as the Mandlebrot set.

Cross Product

The cross product is a mathematical operation that is defined between two three- dimensional vectors. The result is a new vector that is perpendicular to both of the other two vectors. The length (magnitude) of the cross product is equal to the product of the lengths of the two operands times the sine of the angle between them. The cross product is sometimes called the vector product because the result is a vector. The cross product is calculated as follows:

```
vector1: (a b c)
vector2: (d e f)

vector1 cross vector2 = (i j k)
where
i = (b f) - (e c)
j = (c d) - (a f)
k = (a e) - (b d)
```

D

Dot Product

The dot product is a mathematical operation that is defined for two vectors of any dimension. The result of a dot product is always a scalar, so the dot product is sometimes called the scalar product. Geometrically, the length of the dot product is equal to the product of the magnitudes of the operands times the cosine of the angle between them. The dot product of a vector with itself is therefore equal to square of its length. To compute the dot product, you compute the product of each component of one vector and the corresponding component of the other vector and sum them all together. For two three-dimensional vectors, you compute the Dot Product as follows:

vector1: $(u_1\ u_2\ u_3)$
vector2: $(v_1\ v_2\ v_3)$

vector1 dot vector2 =
$(u_1 * v_1) + (u_2 * v_2) + (u_3 * v_3)$

E

Expression

Expressions are combinations of variables using operators and functions to yield new values. An expression is like one half of a mathematical formula. For example, the expression, `(5 + (3 * 5))`, is an expression that evaluates to the value, 20. Expressions can always be evaluated to yield values. The type of data that is generated by the expression is often used to describe the expression, so you may have Boolean expressions, scalar expressions, etc. Expressions require that the proper type of data is used with the operators so some expressions are not valid. For example, the expression, `(true * 3)` is invalid because you can not use Booleans with the multiplication operator.

F

Function

A function, in its mathematical sense, is a relationship that maps elements from one set into elements of another set. For example, the sine function takes any scalar number an maps it to a corresponding value between -1 and 1 on the sine curve. A function, in the context of computer programming, has a similar meaning. In place of an abstract mathematical relationship, a piece of code performs some operations on arguments and returns a value. The value depends upon the values of the arguments and the sequence of operations that are encoded in the function.

G

Global Variable

A global variable is a variable that can be accessed anywhere in a program. The code of a program is normally divided into various procedures, each of which may have its own local variables that only it can access. Because all procedures can access global variables, global variables can be used to share data between different procedures. Generally, though, it's a good idea not to use global variables if they can be avoided. Because global variables can be changed anywhere in the program, it's more likely that their values could be accidentally changed, causing obnoxiously cryptic bugs.

I

Identifier

An identifier is any name that you supply for a variable, procedure, or any other user- defined entity requiring a name. The rules for creating new identifiers are that (1) they must not match a reserved word, and (2) they must begin with a letter and may be followed by letters, numbers, or the underscore character. In addition, the identifier must be unique, meaning that there are no other identifiers with that name that are declared in the same scope.

Imaginary Numbers

Imaginary numbers are numbers that are formed by taking the roots of negative numbers. All imaginary numbers can be expressed as the product of a real number times the number *i*, which is defined as the square root of -1. Imaginary numbers form the imaginary part of complex numbers. When imaginary numbers and real numbers are added together, the result is a complex number.

Integer
Integers are the set of all negative and positive whole numbers, including zero: -1, 0, 1, 2, 3, etc. Integers are sometimes called the counting numbers. Integers are a subset of real numbers and complex numbers. Real numbers (scalars) differ from integers in that real numbers may have a fractional part. Complex numbers differ from integers because complex numbers have an imaginary part that integers are not capable of representing.

K

Keyword
Keywords are identifiers that serve to indicate that something special is following the keyword. Keywords can be used in OMAR procedure calls to indicate that some parameter values are following. For example, in the procedure call `rotate by 50 around <0 0 1>`, the keyword `by` signifies that an angle measure follows and the keyword `around` signifies that the axis of rotation follows. The use of keywords can make procedure calls much more readable because they intersperse words with the numerical values.

L

Local Variable
A local variable is a variable that exists only inside the scope of a particular procedure. Local variables encourage better, safer programming because if a variables only live within a single procedure, you don't need to worry about other procedures inadvertently changing them. Almost all variables in a typical procedural language are local variables.

P

Parameter
A parameter is a number that defines an instance of an object. Parameters are passed into procedures, and, if the procedure does not rely on any global conditions, the parameters completely determine the behavior of the procedure. Similarly, if two objects are created with the same parameters, then they will be identical. Parameters function as a sort of interface to link various procedures together in a structured way.

Precedence
Precedence, in its mathematical sense, is the order in which operators are applied to data. Instead of simply being applied in a left-to-right order, some operators take precedence over others and are always applied first no matter where they come in the expression. For example, multiplication takes precedence over addition, so the expression (5 + 4 * 3) will be evaluated as (5 + (4 * 3)) = (5 + 12) = 17 instead of ((5 + 4) * 3) = (20 * 3) = 60. If two operators have the same precedence level, then they are applied in a left-to-right order.

Procedure
A procedure is a delimited section of a program that is designed to do one particular task. Procedures in OMAR come in a variety of flavors that each have certain restrictions and capabilities

to accomplish different tasks. The different types of procedures are verbs, questions, shaders, shapes, pictures, and anims.

S

Scalar
A scalar is any number with or without a fractional part. In mathematics, these are known as real numbers. Real numbers include the set of integers, or whole numbers; rational numbers, which can always be expressed as a ratio of whole numbers; and transcendental numbers, such as π or e, which can not be written as finite expressions of whole numbers.

SMPL
The programming language that was later called SAGE, and is now OMAR.

String
A string, in computer programming, is simply a set of characters grouped in an array. Strings are used to hold text.

V

Variable
In computer programming, a variable is a single storage unit for data, labelled by a unique identifier. Computations are done by performing various mathematical or logical operations on the variables and moving data around between the variables. All of the data that the computer program knows about must be stored in variables. There are different types of data that can be represented, such as Booleans, characters, integers, scalars, complex numbers, and vectors. Each variable is of a certain type that may not change, and variables of one type may not hold values of any other type.

Vector
A vector is a quantity that has both a magnitude and a direction. Vectors are useful for specifying directions and locations in space. Since space has three dimensions, OMAR vectors have three scalar components, although vectors with more dimensions could be defined. Operators such as addition and multiplication are also defined for vectors as well as some new operators such as dot product and cross product. Since human eyes are sensitive to three primary colors, vectors can also be used to define colors.

Index