Version: 2/6/2002 4:23 AM

An Introduction to Programming using OIL2

An Introduction to Programming using OIL2

FARGOS Development, LLC
757 Delano Road
Yorktown Heights, NY 10598
http://www.fargos.net
mailto:support@fargos.net

Notice of Rights

Trademarks

FARGOS/VISTA, FARGOS/SolidState and FARGOS/SolidConnection are trademarks of FARGOS Development, LLC.

Abbreviations

FARGOS Development, LLC is a Limited Liability Company registered with the State of New York. It is required to identify itself as such in its name, hence the ", LLC" suffix. For purposes of readability in this document, the ", LLC" suffix is sometimes dropped. The phrase "FARGOS Development" always denotes "FARGOS Development, LLC" and is not intended to suggest any alternate form of organization.

Notice of Liability

Information in this document is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this document, FARGOS Development, LLC shall **not** have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained within this document or by the computer software or hardware products described in it.

Contents

1. Preface

Most FARGOS/VISTA-related documentation assumes a modest level of programming and system administration expertise. While truly expert programmers receive significant[1] productivity gains by using FARGOS/VISTA technologies, it can be argued that, ultimately, productivity gains are the sole benefit that such expert programmers obtain. This is because, given sufficient time, they could write all of their applications as handcrafted assembler language. As a practical matter, however, no one has that much free time and productivity gains enable the implementation of applications that would otherwise be impossible due to time constraints.

The majority of programmers, however, are not experts. One of the great features of the FARGOS/VISTA suite of technologies is how even novice programmers are able to quickly implement distributed applications that would otherwise be beyond their abilities. For such programmers, the issue is not merely productivity but instead the reduction in the complexity of a problem by such a dramatic amount so as to make its solution feasible.

Object Implementation Language 2 is one of the cornerstones of the FARGOS/VISTA suite. Programmers with C/C++ experience will be able to extract the nuances of the language by reading the OIL2 Reference manual and can dispense with this guide, which is intended for individuals who have never written a program before or consider themselves inexperienced with C or object-oriented programming.

This document is based on course materials development for and used by an EECS 181 (Introduction to Computers) class in the mid 1980's at the University of Michigan. It was originally oriented around the C programming language, but it has been revised and extended to now use examples written in Object Implementation Language 2. In contrast to most manuals, an informal style has been adopted in keeping with the tutorial nature of this document's contents. The reader is often directed to take certain actions, such as think about a particular issue or run an example program.

[1] Usually 6- to 10-fold improvements, so what might normally take a year of development can be done in a month or two.

2. An Introduction to Programming Using OIL2

As suggested by its name, Object Implementation Language 2 (OIL2) is an object-oriented programming language. Programmers use it to create applications that are implemented as *objects* that interact with each other. Objects are instances of *classes*. If you have not had exposure to these concepts before, they can sound quite technical, so we should agree on some common definitions:

Data: an arbitrary piece of information that is stored or manipulated by a computer. It can be a number (e.g., 2, 3.14, -1, etc.), letters or arbitrary characters (someone's name, a paragraph of text from a novel, etc.), or some arbitrary content. We use the word *data* whenever we want a generic term to talk about information in the computer.

Type: semantic information associated with data that allows its meaning to be determined. Some types are found in almost all programming languages because computers have been designed to manipulate them directly. One of the most common simple types of data is that of an integer (e.g., …, -2, -1, 0, 1, 2, …). Real numbers (often called floating-point due to the way in which they are represented by the computer) are another math-oriented example. Character data (commonly referred to as strings) is yet another primitive type.

Class: a programmer-defined type that defines two things: a collection of data that is maintained together and a set of operations that can be performed against such data collections. The grouped data items are referred to as *instance variables* and the operations are referred to as *methods*. OIL2 programmers write class definitions that describe the types of data to be stored and the programming instructions that implement each method. Virtually every programming example in this document will be of a class.

Object: a distinct physical instance of a collection of data that defined by a class. A class can have zero, one or many object instances. All instances of a class have the same methods. As a crude real-word analogy, consider the process of making decorative sugar cookies. A class definition would correspond to the cookie cutter that cuts a cookie from a sheet of dough. An object would correspond to a cookie. One cookie cutter might be used to create several cookies. While very similar in appearance, each cookie is physically distinct from the others.

Application: a generic term for a complete computer program that performs some (hopefully useful) task. Some applications are very simple (such as the examples found in this guide); others are quite complex. Programmers create complex applications by creating smaller pieces and combining them. These smaller pieces are called subroutines in some languages; in OIL2, they will be implemented as *methods* of a *class*.

What is an Object?

You would be correct if you suspected that the meaning of the word *object* is an important concept for the discussion at hand. An object is an instance of a distinct collection of data that is associated with a particular type definition, which is called a *class*. The class defines what data is grouped together and the operations that can be performed against it. Determining what data is best grouped together is something of an art form: as a rule, a programmer should try to minimize the

number of distinct items that are grouped together as this will tend to keep the class well-focused.
As an illustration of the concept, consider the implementation of a checking account as a class. There are two obvious candidates for instance variables:

- The checking account number which uniquely identifies the account
- The current balance

As far as methods (i.e., operations) go, two should come immediately to mind:

- Deposit funds
- Withdraw funds

Consider: are there any other useful operations associated with a checking account?[2]

Note the separation between the data (such as the account balance) and the operations (like deposit). This separation is one of the key characteristics of object-oriented programming. While object-oriented programming was invented in the 1960's, it only started to become popular in the late 1980's and achieved something of a cult status in the 1990s. Prior to this, most programmers used procedural-oriented languages that imposed little formal structure. The results were programs that were hard to enhance and yielded very little reuse of previously written code. The other major paradigm of programming languages is functional languages. Functional programming languages are well suited for certain problems; however, they require a certain mindset that is not easily grasped by the average practicing programmer.

Defining Classes

Some object-oriented languages, such as C++, enable, yet do not require, the development of object-oriented programs. Thus they still permit programmers to write in the procedurally oriented style so common for over two decades but now frowned upon. In contrast, OIL2 mandates the use of an object-oriented programming style: no half-measures are permitted. Therefore, we have to begin immediately with a discussion of the process of defining a class instead of encountering this issue near the very end of an introductory text or omitting its discussion altogether. There are four attributes associated with an OIL2 class:

- the class name
- the instance variables
- the inheritance hierarchy, which must eventually inherit from the class **Object**.
- the methods, which must include the methods **create** and **delete**.

Class Names

In OIL2, all classes have a name that is defined by the programmer. It is considered good programming practice for the name of a class to have some relationship to its purpose, but this is not a requirement. Thus, we might choose a name like *CheckingAccount* for a class that implemented the checking account example discussed above.

OIL2 coding convention: it is recommended that all OIL2 class names begin with a capital letter.

---

[2] A balance inquiry will turn out to be a valuable addition.

Namespaces and Version Ids

For the sake of completeness, we note that OIL2 classes are actually identified by the combination of three elements:

- a name space
- the class name
- the class version Id

Namespaces solve the potential problem of two programmers accidentally choosing the same name for a class that they developed independently (e.g., *MyClass*): if both programmers place their new class into their organization's respective name space, the two implementations can be distinguished (e.g., *CompanyA.MyClass* vs. *FirmB.MyClass*). Namespaces can also be used to allow locally implemented versions of facilities to replace default versions. Class version Ids address the problem of updating the implementation of code into already running systems or when persistent objects are in use. FARGOS/VISTA-based systems can support truly non-stop operation, so these features are required; most systems would require a restart of the application with the new code.

We will ignore these capabilities and just use their default values in our examples.

Instance Variables

Instance variables define the data that is to be maintained by each individual object. Each instance variables is identified by a unique (to the class) name and have a type declared. OIL2 supports several native types. As defined in the OIL2 reference manual, these types are:

Table 1

Keyword	Type	Comments
nil	A null value	A special value to indicate no value set
int	32-bit integer	A normal integer; int and int32 are synonymous.
int32		
int64	64-bit integer	Sometimes, 32 bits are not enough.
float	32-bit floating point	Single precision floating point.
double	64-bit floating point	Double precision floating point.
fixed	fixed point decimal	Arbitrary precision arithmetic, useful for currency.
string	octet string	Can contain embedded nulls; also keeps track of character set (e.g., ASCII, EBCDIC, Unicode, binary); reference counted implementation in FARGOS/VISTA.
oid	object Id	Reference to an object.
array	sparse array	Can be subscripted by non-contiguous int32 values; reference counted implementation in FARGOS/VISTA.
assoc	associative array	Subscripted by strings (binary data OK); reference counted implementation in FARGOS/VISTA.

set	set	Really a list that preserves order; reference counted implementation FARGOS/VISTA.
nlm	native language message	Internationalized messages.
any	any type	Any of the above types can be used.

The type **any** is quite useful since the corresponding variable can hold any kind of data and is not supported by many programming languages. If one does not know what the type of a variable will be, the variable must be declared as being of type **any**. One might ask why an OIL2 programmer should not always use the type **any**. The reason is that declaration of the exact type of a variable enables the compiler to catch some programming errors at compile time and generate code that is more efficient.

Recommendation: if you know the type of a variable, declare it correctly rather than use the type **any**.

A variable declaration always begins with a type name. The type name is followed by a list of one or more identifier names. Some examples:

```
int         j, k;
float       angle, balance;
string      s, mess;
any         v;
```

Inheritance Hierarchy

Inheritance is an extremely powerful facility that enables the construction of reusable components and elegant designs that are easy to maintain and enhance. Unfortunately, making effective use of inheritance requires appropriate models. It takes some practice for programmers to be able to create useful class hierarchies. That said, the basic concepts are not very difficult to understand.

Formally, inheritance provides support for polymorphism, which means that an object may appear to be of more than one class. The key concept is that a class provides a specialization of a class from which it inherits. As an example, consider a hypothetical class *WheeledVehicles*, which represents information about vehicles that have wheels. Specialized classes *TwoWheeledVehicles* and *FourWheeledVehicles* implement further specialization of the class *WheeledVehicles*. Each of those classes can be further specialized: *Bicycle* and *Motorcycle* could be specializations of the class *TwoWheeledVehicles*; *Automobile* and *PickupTruck* could be specializations of the class *FourWheeledVehicles*. This hypothetical hierarchy illustrates polymorphism: given an instance of *Automobile*, an application can treat it as either an *Automobile* object or a *FourWheeledVehicles* object or a *WheeledVehicles* object. It cannot, however, be viewed as a *PickupTruck* object, or *TwoWheeledVehicle* object.

Some object-oriented languages are restricted to only supporting single inheritance, thus a class can directly inherit from only one class. In contrast, OIL2 supports multiple inheritance. Multiple inheritance permits a class to inherit the capabilities of more than one class. Using the prior hypothetical example, one could introduce a class *MotorizedVehicle*, which could then be used by *PickupTruck*, *Motorcycle* and *Automobile*. Note, if only single inheritance was supported, this focused

implementation could not be used: one would have to implement *MotorizedTwoWheelVehicles* and *MotorizedFourWheelVehicles* to handle the situation created by human-powered vehicles like *Bicycle*.

Most object-oriented languages do not require that a class inherit from some other class; however, OIL2 does require that all classes must eventually inherit from the ultimate base class **Object**. Therefore, all OIL2 classes inherit from at least one class, which often is just **Object**.

Class Definition Syntax

Class definitions are introduced by the keyword **class**. The prototype appears below:

```
   class [NameSpace . ]ClassName[(versionID)] {
         typeName    var1 [, var2 …];
   } inherits from ClassName1 [, className2 …];
```

As one can see, the class definition permits the specification of the name space, class name, version Id, instance variables, and the class from which the new class inherits. The declaration of the *CheckingAccount* class that was previously discussed appears below:

```
class CheckingAccount {
   string      accountId;
   float       balance;
} inherits from Object;
```

Figure 1

It is worth noting that the definition is quite concise, yet readable.

Methods

As explicitly noted by its name, Object Implementation Language 2 is an implementation language, not merely a definition language. The key feature of a programming language is to describe operations that a computer should perform. In OIL2, these instructions are written as methods of classes. Every OIL2 needs to define at least two methods, which are **create** and **delete**. The **create** method is executed when a new object of the class is created; the **delete** method is executed when an object of the class is deleted. Quite useful classes can be implemented that only have these two methods; however, most classes implement additional methods that provide useful class-specific functionality.

While not necessary, most methods accept some arguments at runtime that affect their operation. Many programming languages expect a method to take a fixed number of arguments and require the programmer to assign a name to each argument so that it can be addressed. In contrast, OIL2 makes it trivial to work with routines that take a variable number of arguments. One way in which OIL2 enables working with variable number of arguments is that it always makes available the arguments to a method in the predefined array *argv* and it provides a count of the total number of arguments in the predefined variable *argc*. For methods that do expect a predetermined number of arguments, it is possible to declare each positional argument. This has the benefit of making it more convenient to access a given argument and it provides the compiler with type information that it would otherwise not have.

A method definition has the following prototype:

```
ClassName:methodName[(typeName arg1 [, typeName arg2 …])]
{
   typeName    var1 [var2, …];
   executableStatement;
   …
}
```

Although each programming language is different, one can expect to find certain common features and capabilities in each. For example, it should not come as a surprise to find that programs have a starting point. We cannot have the computer start executing our program at just any spot, thus a language needs some means of indicating where is the program's beginning. In OIL2, methods start execution at the top and terminate when the flow of execution reaches the bottom of the method. Methods can also be explicitly terminated by the OIL2 **exit** or **return** statements.

Continuing with our illustrative example of a checking account (review Figure 1 for the class definition), the **create** method will probably set the account's Id and opening balance. The deposit method increases the available balance by the amount of the deposit, whereas the **withdraw** method should probably only process the request if sufficient money is available.

```
CheckingAccount:create(string acctId, float openingBalance)
{
   accountId = acctId;
   balance = openingBalance;
}

CheckingAccount:delete() {} // null method since nothing needs to be done

CheckingAccount:deposit(float depositAmount)
{
   balance += depositAmount;
}

CheckingAccount:withdraw(float desiredAmount)
{
   if (balance >= desiredAmount) {
         balance -= desiredAmount;
         return (desiredAmount);
   }
   return (0);
}
```

Figure 2

The majority of this document deals with the use of OIL2 executable statements. Despite not having yet covered these statements, the source code above (Figure 2) should be understandable with a little effort. The notations "+=" and "-=" mean add or subtract a value, respectively, from the variable on the left hand side. The expression "*balance >= desiredAmount*" returns a true value if the value of the variable *balance* is greater than or equal to the value of the variable *desiredAmount*.

Compiling OIL2 Source Code

The best way to learn or get a feel for a programming language is to actually see some examples. For the remainder of this document, we will be working with complete examples that you will probably want to try out on your computer. OIL2 source code is stored as plain text files and it is suggested that OIL2 source files be named with a *.oil* suffix. Developers should use the text editor of their choice to create and edit OIL2 source files; the compiler can handle source code created on platforms with incompatible notions of end-of-line markers.

Configuring the Development Environment

Before you can compile and run applications written in OIL2, the system administrator of your computer must install the FARGOS/VISTA Software Development Kit. Your organization may have purchased a FARGOS/VISTA license that permits their developers to generate native object code; however, we will focus only on the generation and use of OIL2 Architecture Neutral Format (OIL2 ANF) object code. Your system administrator should have set the **VISTA_ROOT** and **VISTA_UNAME** environment variables appropriately:

- **VISTA_ROOT** must be set to point at the location where the software was installed.
- **VISTA_UNAME** is set to identify the type of the underlying system (Windows for Microsoft Windows variants or the result of the **uname** command on Unix variants).

Details of the installation process are provided in the FARGOS/VISTA Installation Guide. For convenience, the directory identified by *$VISTA_ROOT/$VISTA_UNAME/bin* should be added to your executable search **PATH**.

Invoking the OIL2 Compiler

The OIL2 compiler executable is named **oil2_parse**. There is also a script, named **oil2**, which invokes the **oil2_parse** program after running a source file through the local C preprocessor. The **oil2** script permits a programmer to use conventional C preprocessor directives, but it is wasted overhead if an OIL2 source file does not use any such directives. Also, if the local system does not have a C preprocessor, the script will not work. For those reasons, none of the examples in this document will use C preprocessor directives and it is suggested that the **oil2_parse** program be used directly. The OIL2 compiler is instructed to generate OIL2 Architecture Neutral Format object code by the –oil2 option:

```
    oil2_parse –oil2 srcFile.oil
```

The resulting object code file will be generated with the *.oil* suffix of the source file name replaced with *.o2o* (for OIL2 Object).

The Hello World Program

From its appearance in the book the "C Programming Language", a simple program that displays the message "Hello, world" has become one of the most common introductory examples of a language. Consider the following:

```
%include <OMEcore.o2h>

class HelloWorld {
} inherits from Object;

HelloWorld:create()
{
   display("Hello, World!\n");
}

HelloWorld:delete() {}
```

Figure 3

The directive **%include** is directly supported by the OIL2 compiler and is conceptually identical in behavior to the **#include** direction recognized by C preprocessors. The OIL2 compiler will attempt to locate and insert the indicated file at that point in the parse stream. Most OIL2 programs will use the illustrated directive to include definitions of the standard OIL2-callable interfaces provided by the FARGOS/VISTA Object Management Environment. The angle-brackets ("**<**" and "**>**") are used to indicate that the file should be found in a special place used by everybody (normally, the directory *$VISTA_ROOT/oil2Include*).

The output of information is a crucial issue in every program. If a program never outputs a single item of data, it serves no purpose except to use up CPU time. In OIL2, interactions between objects are performed by sending messages using the **send** statement. The standard class **IOobject** provides the means to access a variety of I/O devices, like files and network communications links, by representing them as objects.

For efficiency, some facilities are provided by the FARGOS/VISTA Object Management Environment via functions. The **display()** function used above is one such example. This is used for debugging purposes: it converts and displays on standard output any data that is passed as its arguments. The ability to call functions is part of the specification of OIL2 as a language; any available functions are either part of the FARGOS/VISTA Object Management Environment or locally available enhancements.

If we examine the program, we see the expected declaration of the **create** method. We also see a set of braces ("**{**" and "**}**") that enclose the body of the method. Braces are used in OIL2 to denote blocks of statements. They are the equivalent of **begin** and **end** in languages such as Pascal. For every "**{**", there must be a matching "**}**" (as you would hope in order to preserve symmetry). The body of this method is quite short, consisting of a single statement, namely the call to the **display()** function. Note the semicolon (";") after the statement. All statements in OIL2 are terminated by a semicolon. There are no exceptions, no special cases.

You may have wondered what is the meaning of "**\n**"? The **display()** function does not automatically move to a new line after printing. This is good—we can do a lot more because of this, but to get text to start on a new line, we must explicitly output the new line character. We cannot type this character, nor do we know what the coded value is on a particular system (not that we are really interested). There are several special characters that are not readily typed—the new line character is just one of them. OIL2 allows us to specify these characters using "escapes". The escape character is the backslash ("****"). What follows the backslash has a special interpretation. In this case, "\n" represents the new line character. Therefore, the "\n" in the displayed string will cause subsequent output to start on a new line.

If the source shown above in Figure 3 is placed into a file named *hello.oil*, it can be compiled into an OIL2 ANF file named *hello.o2o* by issuing the command:

```
oil2_parse –oil2 hello.oil
```

Running an OIL2 Application

Once an OIL2 program is compiled, one typically wants to use it. Applications written in OIL2 run inside a FARGOS/VISTA Object Management Environment. The standard version of this system is made available via the **vista** executable. The **vista** executable processes a file that directs it to create a set of objects as part of its initialization process. These files are often referred to as "rc" files (rc stands for

run commands). To enable their use on desktop graphic user interfaces, it is recommended that such files be saved with a *.vrc* suffix, which should have been associated with the **vista** executable.

To run the "hello world" program described above in Figure 3, a two line *rc* file will do the necessary work:

```
LoadOIL2File file:hello.o2o
HelloWorld
```

If the above was placed into a file *hello.vrc*, then the program could be run by issuing the command:

```
vista hello.vrc
```

Output similar to the following should be seen:

```
FARGOS/VISTA Object Management Environment
Copyright © 1999 – 2002 FARGOS Development LLC.  All rights reserved.
Hello, World!
```

While OIL2-based applications can be deployed in quite sophisticated scenarios, including dynamically loaded from remote systems, the basic development scenario holds:

1. An OIL2 programmer places source code into a text file with an *.oil* suffix.
2. The OIL2 source is compiled using the OIL2 compiler to generate some form of object code, such as OIL2 ANF or native object code.
3. The generated object code is placed into a FARGOS/VISTA Object Management Environment, often by dynamically loading the object code into a running **vista** process.
4. At some point, the code is utilized by creating an object of a class implemented in the object code file.

With this background, we can move on to the discussion of executable statements.

Exercise 1

Write an OIL2 program that prints your name out. Compile and run it. Modify it so that you use two calls to **display()**: one to print your first name, one to print your last. Compile and run this one too.

3. Executable Statements

The real work of an OIL2-based application is achieved through executable statements. The discussion below will explore the use of many of the executable statements available in OIL2.

A Temperature Conversion Chart

Let's enter a program that will print out a table of Fahrenheit/Celsius equivalents. You may recall that to convert a Fahrenheit temperature to its Celsius equivalent, you subtract 32 from the Fahrenheit temperature and multiply the result by 5/9 ths. This may be expressed by the equation:

$$C = (5 / 9) (F - 32)$$

Let's write an OIL2 class to print out a conversion table for the integral Fahrenheit temperatures from 0 to 100. How would we do this by hand?

First, we would start with the Fahrenheit temperature being 0.

Next we would do the calculation C = (5/9) (F-32) and write the result down.

Finally, we'd add 1 to the current temperature on which we were working. If the result was less than or equal to 100, we'd do the next calculation.

An OIL2 class to do this follows:

```
/* Fahrenheit to Celsius conversion */
%include <OMEcore.o2h>

class FtoCtable {
} inherits from Object;

FtoCtable:create(int start, int last)
{
   int fahr;
   float celsius;

   fahr = start;
   while (fahr <= last) {
         // 5.0, 9.0 to denote real
         celsius = 5.0 / 9.0 * (fahr - 32);
         display(fahr, "\t", celsius, "\n");
         fahr = fahr + 1;
   }
   send "deleteYourself" to thisObject;
}

FtoCtable:delete() {}
```

Figure 4

If you put the program in a file *ftoc.oil*, it can be compiled using the command:

```
   oil2_parse –oil2 ftoc.oil
```

Test out the result by placing the following into the file *ftoc.vrc*:

```
LoadOIL2File file:ftoc.o2o
FtoCtable 0 100
```

Test the application:

```
   vista ftoc.vrc
```

This program demonstrates quite a few things. First, we had two variables in this program, namely "*fahr*" and "*celsius* ". We declared both variables in the very beginning of the program, before we used them. We declared *fahr* as an integer ("**int** "), because we knew it would always be an integer. We did not declare *celsius* to be an integer because we knew that it was going to take on real values. Why? Because of the multiplication by 5/9ths.

Arithmetic Precision

We normally represent real numbers in a computer using a format called floating-point. That's why *celsius* is declared as "**float**". Floating-point numbers are stored using a limited number of significant digits along with an exponent. This is very much like scientific notation: 12500 is the same as 1.25×10^4. OIL2 supports two kinds of floating-point numbers: single precision, which is selected by using the type **float**, and double precision, which is selected by using the type **double**. As you may

have suspected, double precision values take up more space, but allow additional significant digits to be maintained.

While floating-point values enjoy a very significant advantage with respect to the speed of calculations, they suffer from one notable restriction, namely that they have a finite number of significant digits. Consequently, very large or small numbers lose digits. As an illustration, consider a floating-point representation that can only maintain three digits of precision and uses base 10 exponents. It could thus represent the number 1.23 as 1.23 E0, the number 12.30 as 1.23 E1 and 123 as the number 1.23 E2. Unfortunately, the number 123.45 would still be represented as 1.23 E2, thus losing the .45. If the number was 12345.67, it would be represented as 1.23 E4, an error of 45.67.

OIL2 also supports an alternative representation of real numbers that provides for arbitrary precision. This type is called **fixed** for fixed-point. Since the underlying architecture of most modern computers natively supports floating-point arithmetic, the types **float** and **double** yield faster calculations. If arbitrary precision is required then the type **fixed** is available, although such calculations are much slower.

We now know how to declare variables that are going to only hold integers (use **int**) and those that are to hold real numbers (use **float**). Note once again the semicolon (";") after each and every statement. We call statements such as "int fahr;" *declaration* statements.

We also see three *assignment* statements (e.g., "fahr = 0;"). Assignments in OIL2 take this format: variable = expression. Note the use of the equals sign ("=").

There is also a looping construct used in this program (the "**while**" statement). It takes the following form:

```
while (condition) statement;
```

More often than not, we want the **while**-loop to control more than one statement. To do this, we enclose the statements in question between braces ("**{**" and "**}**"). Thus, whenever a statement can be used, we could also group a collection of them:

```
while (condition) statement;
```

or

```
while (condition)
{
      statement1;
      statement2;
      /* more */
      statementn;
}
```

There is no semicolon after the closing brace ("**}**") because the brace is not a statement. What does the **while**-loop construct do? As long as the conditional expression in the parentheses is true, the statements in the block enclosed by the braces are executed[3]. Loops are of critical importance in programming. Without

[3] What is truth? In OIL2, truth is actually represented by a zero value. Normally one is never concerned with this: we write conditional expressions like "a < b" and let the compiler generate the required code to do the test; however it is possible to take advantage of this to write more efficient code at the expense of readability.

loops it would be easier to do the work by hand: you would have to type 100 equations and **display()** statements to solve this problem if we could not use a loop.

Comments are also illustrated. OIL2 supports two styles of comments. Multi-line comments are enclosed between "**/*** " and "***/**". Any text between these two sets of characters is ignored by the OIL2 compiler. The other style treats all characters remaining on a given line as a comment. Such single line comments are introduced by the characters "**//**".

Finally, we see some real (pun intended) math: the expression 5.0/9.0 is evaluated, and multiplied (the "*" operator denotes multiplication here) by the quantity *fahr* minus 32. Why 5.0 / 9.0 instead of 5/9? If we wrote 5/9, the compiler would treat this as an integer divided by an integer and would produce an integer result, which would be zero. The 5.0/9.0 tells the compiler to divide real numbers and produce a real result (which is stored in the computer as a floating-point number).

We also see some more features of the **display()** routine. This time we pass several arguments. Once again, we see a "**\n**" to denote a new line and add the use of "**\t**" to indicate a tab character.

Method Arguments

For the first time, we have taken advantage of the ability to pass arguments to methods. All methods have two special variables that are pre-declared and always available:

```
int   argc;
array argv;
```

The number of arguments that were passed to a method is available in the variable **argc**, thus if no arguments were passed, its value is zero. Likewise, if one argument was passed, its value is one and so on. The actual arguments provided are made available in the array **argv**. Unfortunately, our introduction to arrays does not take place until later (see), so a fuller understanding will have to be deferred. For the time being, note that the first argument provided is stored in **argv**[0], the second argument is stored in **argv**[1], the third in **argv**[2], etc. OIL2 permits such arguments to be declared, thus assigning them a type and alternative name. In the example above, the first argument declared to be an integer (**int**) and called *start*. Thus *start* is the same as **argv**[0]. The second argument in the example was declared to be *last*, thus it serves as an alias for **argv**[1].

The ability to reference arguments using **argc** and **argv** is an extremely powerful facility that makes it trivial to implement methods that handle a variable number of arguments. This is something that is extremely difficult to do in C: it's fair to say that most C programmers have never implemented a routine that took a variable number of arguments because of its complexity. It is also convenient to be able to assign a name to a positional argument, which makes the code more readable, easier to maintain and permits the opportunity to give the compiler a hint as to the expected type of the argument.

Sending Simple Messages

Our first use of the OIL2 **send** statement also appears the example shown in Figure 4. The **send** statement is one of the most important OIL2 statements (arguably, only the assignment statement is more critical). In the OIL2 object model, the only way that two objects can interact is by sending a message and the OIL2 **send**

statement provides this critical functionality. Two parts of every **send** statement are required:

- the name of the method to be invoked
- the destination to which the message should be sent

In the example above, the name of the method being invoked is "**deleteYourself**", which is a method implemented by the ultimate base class **Object**. The **deleteYourself** method requests that the object perform the process of deleting itself. The actual name of the method can be computed at run-time, but the most frequent usage is use a string constant. The destination of the message is the object can be specified as either an object Id or a string that indicates the name of a registered service. If a string is used, the indicated name is automatically looked up and converted to an object Id. In this example, the destination was specified as *thisObject*, which is a special pre-declared variable. The variable *thisObject* holds an object Id that refers to the object upon which the active thread is working.

Exercise 2

Modify the Fahrenheit-to-Celsius program to print out equivalents from 200 to 212 degrees Fahrenheit. Increment by 1/2 degree.

Exercise 3

Write a Celsius to Fahrenheit conversion program, and produce the table for the integral Celsius degrees 0 to 100. The formula you need:

F = 9/5 * C + 32

Strings

We have dealt primarily with number crunching up to this point. Admittedly there is a strong tendency to associate extensive calculations with computers, but is this a realistic bias? Not particularly. There are many computer applications that do not do a lot of work with numbers. A notable example is word-processing: it is concerned with the manipulation of text, not numbers. Many programs used in commercial data processing work must manipulate text as well as numbers (e.g., your name on a bill).

How is this done? We use a data type called **string**. In OIL2, strings are a sequence of characters. A string constant is declared using double quotes and we have seen string constants used in every one of the prior examples. One of the simplest things we can do with an OIL2 string is copy it:

```
string      s1, s2;
s1 = "abc";
s2 = s1; //s2 now is "abc"
```

The OIL2 runtime makes a string copy extremely efficient: even if the string's length is several megabytes, copying a string requires only a very small number of bytes being moved or altered. The exact number of bytes moved or altered depends on the memory-addressing model in use (32-bit vs. 64-bit), but it should be 16 or less.

One of the simplest string-related functions is the **length()** function, which returns the number of characters in a string:

```
int         l;
string      s;

s = "abc";
l = length(s); // l is equal to 3
```

In the OIL2 runtime, the length of a string is always maintained with the string, so a **length()** call is very efficient and takes a constant amount of time. In contrast, the equivalent **strlen()** function in C or C++ takes time proportional to the length of the string to compute its result. In OIL2, strings can be concatenated using the addition operator:

```
s = "hi" + " " + "there!"; // s = "hi there!"
```

Strings can also be compared using the relational operators, such as "**==**", "**!=**", "**<=**", etc. The equals and not-equals operators are particularly efficient. For example, since OIL2 strings always know their length without requiring any computation, strings of dissimilar length can be immediately proven to be not equal by merely comparing their respective lengths.

In addition to the **length()** function, there are two other fundamental functions associated with strings: **midstr()**, which extracts a portion of a string, and **midchar()**, which extracts a single character from a string.

It is often necessary to determine if a given string is found within the contents of another string, and if so, where it starts. Thus, "*cd*" is found at offset 2 in the string "*abcdef*". How might we go about writing such code? We could start by looking for the first character of the substring in the larger string and scan right until we found it or reached the end of the larger string. If we find the character, then we check the second character. If it does not match, we move right again and start over with looking for the first character of the substring. When all of the characters of the substring match, we have found it and can stop. This gives us:

```
%include <OMEcore.o2h>

class Local . Substring {
} inherits from Object;

Substring:create(string s1, string s2)
{
   int   i, done;
   int   i1, i2;

   i = 0;
   done = 0;   // not done

   while ((done == 0) && (i < length(s2))) {
         i1 = 0;     // start at beginning of s1
         i2 = i;
         while ((i1 < length(s1) &&
             (midchar(s1, i1) == midchar(s2, i2)))) {
               i1 += 1;
               i2 += 1;
         }
         if (i1 == length(s1)) done = 1; // reached end of s1
         else i += 1;      // lookat next character
   }
   if (done) {
         display("Found ", s1, " at offset ", i, " within ", s2, "\n");
   } else {
         display("Could not find ", s1, " within ", s2, "\n");
   }
   send "deleteYourself" to thisObject;
}

Substring:delete()   {}
```

Figure 5

The algorithm above seaches for a substring *s1* within a string *s2*. The presented algorithm was intended to be easily understood and it is not the most optimal implementation of such an algorithm. A best of breed algorithm that searches for a substring is used to implement the **findSubstring()** function and OIL2 programmers should use it rather than create their own implementation. **Note:** the implementation of **findSubstring()** is optimized in several ways and recognizes several special cases that can be handled by a single CPU instruction.

Simple Input/Output in OIL2

We have mentioned that if a program does not do any output, then it serves no purpose. Well, if we can't do any input, we will also be restricted in our capabilities. At first, most people are surprised at just how much you can do by just reading a file starting at the beginning and reading until the end, but it turns out one can do quite an incredible amount of useful work by doing just that. More than 100 Unix utilities just do this, each with a specific purpose.

We are going to write a class that performs the same function as the **wc** utility, found on computer systems running the Unix operating system. The program **wc**, which stands for "word count", reads an input file and prints out the following information: how many characters, how many "words", and how many lines are in it (thus three separate items). This may not seem too useful, but it really is. For example, **wc** would be used in conjunction with the Unix **who** command to determine how many people are currently logged onto the system.

How can we read the file? We will first create an **IOobject** that will act as the interface to the file. An **IOobject** is able to deal with a wide variety of devices, such as local files and network communication via stream and datagram sockets. The create method of **IOobject** takes an argument that indicates what device should be opened. These always begin with a scheme prefix and, by design, are very similar to URLs one might encounter while browsing the Internet with a web browser. Several schemes are supported by the FARGOS/VISTA Object Management Environment core and a given system may be extended with additional capabilities. Some of the commonly supported schemes are:

- file:
- tcp:
- udp:

For the purposes of this example, our focus will be on the **file:** scheme. Its prototype is similar to:

file:filename[,{r|w|t|a|c|e}+]

The file name is mandatory, but it can be qualified with some command options. These are provided by separating the file name and the list of options with a comma (","). If no command options are provided, the default operation is to open the file for read. There are several recognized flags:

Table 2

Command Flag	Meaning
R	open for read
W	open for write
T	truncate the file
A	append and open for write
C	create if the file does not exist
E	the file must exist

Thus, to open the file "*abc.txt*" for reading, the following specification could be used:

file:abc.txt,r

In a similar fashion, the file "def.txt" could be created (if needed), truncated if data is already present and opened for writing using the following specification:

file:def.txt,cwt

Object Creation

Objects are created by sending a **createObject** request (or equivalent) to the **ObjectCreator** object. This special object is automatically created as part of the boot procedure of each FARGOS/VISTA Object Management Environment process. Its object Id is always made available via the special pre-declared variable *ObjectCreator*. In OIL2, object Ids are a special type named **oid**.

Note: each FARGOS/VISTA Object Management Environment process is uniquely identified by the object Id of its respective **ObjectCreator** object.

The **createObject** method takes two mandatory arguments:

- the name of the class of the object to be created
- an access control list for the new object

Any additional arguments that are passed are made available to the **create** method of the new object's class. The access control facilities for objects residing within a FARGOS/VISTA Object Management Environment are quite flexible, but the most commonly used access setting is to permit complete access to the user who creates the object and deny access to all others. The FARGOS/VISTA Object Management Environment provides a standard OIL2-callable function to create such an access control list: **makeDefaultACL()**. This function and others like it are described in the FARGOS/VISTA Object Management Environment Programmer's Reference. The method fragment below illustrates the creation of an object:

```
assoc    acl;
oid      fileObj;

acl = makeDefaultACL();
fileObj = send "createObject"("IOobject", acl, fileName) to ObjectCreator;
```

Figure 6

Remote Procedure Call-style Sends

Often an application will want to send a message to another object and continue processing. We saw such an example in Figure 4 when the **deleteYourself** message was sent. There are many other situations, however, in which an application wants to send a message to another object in order to obtain some information. In such a situation, the application wants to wait until a response is received that contains the information of interest. This common requirement is conveniently supported by the OIL2 compiler using what is called an RPC-style **send** statement. An example of such usage is found in Figure 6, where an object is created by sending a **createObject** message to the *ObjectCreator* object. In this particular example, the application waits until the object Id of the new object is returned by the **createObject** method and stores the result in the variable *fileObj*.

The **readBytes** method of class **IOobject** reads data from a stream device, such as a file or TCP connection. The **readBytes** method normally takes a single argument that specifies the maximum number of bytes desired. It returns all of the available data up to the specified limit. Upon reaching end-of-file, the special value nil is returned. Because data returned from **readBytes** can be of either type string or nil, the variable used to hold the return value should be declared as type any.

Special Operators

Up to this point, we have seen assignment constructs that have occurred time and time again, one of which is a special case. There have been a lot of statements like:

var = var + expression

as in

```
i = i + 1;
sum = sum + list[i];
```

OIL2 has a nice way of expressing this, using the "+=" operator. The above examples would then become:

var += expression

as in

```
    i += 1;
    sum += list[i];
```

You can see how this would save typing. It can also reduce errors, as you would not have to repeat a complicated subscript expression. As a simple example:

```
    i = c - '0';    // get value of digit
    n_digits[i] = n_digits[i] + 1;
```

Normally we would write:

```
    n_digits[c-'0'] += 1;
```

and thus the extra variable *i* would not be needed.

There are several "*op*= " operators, such as "***=**". In the program of Figure 12, instead of:

```
    for (j=1;j<=i;j+=1) x_to_i = x_to_i * x;
```

we wrote:

```
    for (j=1;j<=i;j+=1) x_to_i *= x;
```

Reading Files

For the first version of our example, we will read data a byte at a time until the end of the file is reached. To reduce the complexity of the example, only characters and lines will be counted in this first version. So, when we read a character, what do we do? First, we want to keep track of the number of characters in the file, thus after reading each character we should increment a count. How do we tell if we are at the end of a line? Each line is terminated by a new line character ("\n"). Thus, we will check to see if the character read is a new line, and if so, increment the count of the number of lines.

```
%include <OMEcore.o2h>

class WC1 {
} inherits from Object;

WC1:create(string fileName)
{
        string  fileSpec;
        oid     fileObj;
        assoc   acl;
        int     characterCount, lineCount, wordCount;
   any   data;

        acl = makeDefaultACL();
        fileSpec = "file:" + fileName + ",r";

        fileObj = send "createObject"("IOobject", acl, fileSpec)
                to ObjectCreator;

        characterCount = 0;
        wordCount = 0;
        lineCount = 0;

        data = send "readBytes"(1) to fileObj;
        while (data != nil) {
                characterCount += 1;
                if (data == "\n") lineCount += 1;

                data = send "readBytes"(1) to fileObj;
        }
        display("characters=", characterCount, " words=", wordCount,
                " lines=", lineCount, "\n");
        send "deleteYourself" to fileObj; // close input file
        send "deleteYourself" to thisObject;
}

WC1:delete() {}
```

Figure 7

Note the use of "**!=**" to express the "not-equals" comparison in the conditional part of the **while**-loop and the "**+=**" assignment operator, which adds the right hand side of the expression to the left hand side and stores the result back into the left hand side argument.

What does this code fragment do? It reads the first character in the file. If the file is empty, the special value **nil** will be returned at this point. Next, the condition in the **while** statement is evaluated. If the value **nil** has not been returned, then we process the character. After processing the character, we read the next character in the file, and jump to the top of the **while**-loop. Eventually, the end of the file will be reached and the special value of **nil** will be returned, which will cause the loop to be exited. Finally, the computed totals will be printed.

We see here our first "**if** " construct. It takes the form of:

```
   if (condition) statement
```

Note the use of "**==**" to represent the "equals" comparison. Be careful when typing: using "**=**" does not do what you want as "**=**" is the assignment operator. If you typed:

if (c = "\n"**)** ...

this would have the effect of changing the variable *c* and setting it equal to "*n*"—not what we intend. What we need is for the variable *c* to be compared to the string corresponding to the new line character, which is represented by "*n*" (the compiler

understands this special sequence), and if *c* is equal to that value, then increment the variable *lineCount*. Using a single equals-sign instead of two is a very common typing mistake and can be a hard error to track down.

Finally, how do we handle words? Let us make things simple and say that words are separated by a space, tab or new line character. We will need to keep track of whether or not we are in a word. We'll do this by using a variable, call it *inWord*, which will be equal to 1 if we are in a word, and equal to 0 if we are not. If we are in a word, and reach the end of a line (signaled by reading a new line ("\n") character) or read a space, then we have reached the end of the word. Increment the word count and set *inWord* equal to zero to denote that we are no longer in a word.

If we are not in a word, and read a character that is not a space, tab or a new line character, then we should set *inWord* equal to one to denote that we are now in a word. Thus, we have:

```
%include <OMEcore.o2h>

class WC2 {
} inherits from Object;

WC2:create(string fileName)
{
        string  fileSpec;
        oid     fileObj;
        assoc   acl;
        int     characterCount, lineCount, wordCount, inWord;
   any   data;

        acl = makeDefaultACL();
        fileSpec = "file:" + fileName + ",r";

        fileObj = send "createObject"("IOobject", acl, fileSpec)
                to ObjectCreator;

        characterCount = 0;
        wordCount = 0;
        lineCount = 0;
        inWord = 0;

        data = send "readBytes"(1) to fileObj;
        while (data != nil) {
                characterCount += 1;
                if (data == "\n") lineCount += 1;
                if ((inWord == 1) && ((data == "\n") ||
                        (data == " ") || (data == "\t"))) {
                        // then was inside a word and reached the end
                        wordCount += 1;
                        inWord = 0;
                } else if ((inWord == 0) && (data != " ")
                         && (data != "\n") && (data != "\t")) {
                         inWord = 1;
                }

                data = send "readBytes"(1) to fileObj;
        }
        display("characters=", characterCount, " words=", wordCount,
                " lines=", lineCount, "\n");
        send "deleteYourself" to fileObj; // close input file
        send "deleteYourself" to thisObject;
}

WC2:delete() {}
```

Figure 8

All sorts of good stuff appears here, such as the "**&&**" and "**||**" operators (which represent the logical AND and logical OR functions). We once again make use of the braces ("**{**" and "**}**") to enclose groups of statements. Note the use of braces in the third **if** statement ("**if** (inWord == 0...)"). In this case, the braces are not required. Why? Because there is just a single statement, namely the "inWord = 1;". So why use the braces? It just makes it easier to insert additional statements if it turns out later that they are needed.

So, what does the first new **if** construct say? It asks if the variable *inWord* is equal to 1 (i.e., are we in a word?), and if the variable *data* is equal to "*n*" (the new line character), a space or a tab. If true, then we have reached the end of the word and we will increment he word count and reset the *inWord* variable. Note the use of parentheses around the OR clause. The parentheses behave in the manner you would expect. The value of the expression is the result of the OR operation and this result is used as the second operand of the AND operation.

We also see our first "**else** ". If the condition of the first **if** is evaluated as false, then the **else** clause is executed instead. In this case the clause is another **if** statement that asks if we have reached the beginning of a new word.

Note: the order in which certain things are evaluated is normally not too important, but occasionally it is. OIL2 has a few built in "precedence" rules. For example, it will do multiplication and division first, and then do addition or subtraction. This is just as you did in your math classes. This order of evaluation can be modified by the use of parentheses, again just like normal math.

As another example, consider a utility that will strip trailing blanks from lines in a text file.

```
%include <OMEcore.o2h>

class Local . StripBlanks {
} inherits from Object;

StripBlanks:create(string fileName)
{
   oid   fileObj;
   assoc acl;
   string      fileSpec;
   int   rc, i, char;
   any   data;
   string      line, stripLine;

   fileSpec = makeAsString("file:", fileName, ",r");
   acl = makeDefaultACL();
   fileObj = send "createObject"("IOobject", acl, fileSpec)
         to ObjectCreator;

   rc = send "getID" to fileObj;
   if (rc == -1) { // could not open...
         display("Could not open ", fileName, "\n");
         send "deleteYourself" to fileObj;
         send "deleteYourself" to thisObject;
         exit;
   }
   line = "";
   data = send "readBytes"(1) to fileObj;
   while (data != nil) {
         if (data == "\n") {     // end of line
               i = length(line) - 1;
               while (i >= 0) {
                     char = midchar(line, i);
                     if ((char != ' ') && (char != '\t') && (char != '\r')) break;
                     i -= 1;
               }
               stripLine = midstr(line, 0, i + 1);
               display(stripLine, "\n");
               line = "";  // reset for next line
         } else {
               line += data;
         }
         data = send "readBytes"(1) to fileObj;
   }
   send "deleteYourself" to fileObj;
   send "deleteYourself" to thisObject;
}

StripBlanks:delete() {}
```

Figure 9

Exercise 4

The **IOobject** class provides a **writeBytes** method that is a complement to the **readBytes** method. It is called as shown in the following code fragment:

```
   int   bytesWritten;

   bytesWritten = send "writeBytes"(data) to obj;
```

Write a simple class in which the **create** method takes two arguments that indicate the name of a source file and an output file. Duplicate the source file's contents to the destination file and, for efficiency, read and write the data in blocks larger than 1

byte. The resulting application will be a close equivalent to the Unix **cp** command or MS-DOS **COPY** command or the standard FARGOS/VISTA class **SendFile**.

Exercise 5

One of the things the C preprocessor normally does when it reads a source file is to strip comments so that the resultant output from the preprocessor contains only characters that a compiler needs to see. Write an OIL2 class that performs this function of the C preprocessor. It should read a C/C++/OIL2 source file and write the equivalent source program into an output file with no comments in it. **Note:** comments do not nest.

Exercise 6

Add the required statements to the **WC2** class in Figure 8 so that it will also count and display the number of paragraphs in the file. Let us say that a paragraph is denoted by a sequence of two or more new lines in a row.

Exercise 7

Write a simplified version of the Unix utility "**prep** ". The **prep** command prepares a file for statistical processing. Your **prep** will read a source file and write its contents, with one word-per-line, to an output file.

Arrays

Arrays are a very important concept. Up to this point, we have set aside memory locations for storing data by declaring specific variables (e.g., "*int c*;", "*float celsius*;"). This has worked fine for previous problems; however, we can anticipate several kinds of problems where this would not be feasible. As an example, consider a word-processing program. We cannot call each distinct memory location by a different name. Similar problems arise in math: we often need more variables than we can uniquely name. In math, we use subscripts to solve this problem: one can write X_i (pronounced as X "sub" i). You can probably guess that we will do something similar in programming.

The primary storage of a conventional computer system is organized as an array of memory locations, each of which has a unique address. These addresses begin at 0, thus the second memory location in the computer is address 1. The mathematically-oriented can make the analogy between this and a vector. Indeed, arrays are sometimes referred to as vectors by some programmers.

One point that (while not hard to see) often causes difficulty is the fact that if the array (or vector) subscripts are numbered starting at 0, and the array has *n* elements in it, then the subscript of the last element is *n - 1*, not *n*.

In general, we use arrays for two purposes: 1) we have a problem requiring too many variables to be named uniquely, or 2) we want to select a variable based on computations performed at runtime. The special pre-defined OIL2 method variable *argv* is an example of the latter: while methods have a small number of arguments and can easily be assigned unique names, being able to access the parameters via an array permits the handling of a variable number of arguments.

Let's write a program before we are too bogged down in words. We will write a simple calculator program that processes the arguments handed to the **create** method.

```
%include <OMEcore.o2h>

class Calc1 {
} inherits from Object;

Calc1:create()
{
        int     i;
        string  operand;

        if (argc < 3) {
                display("Not enough arguments\n");
                send "deleteYourself" to thisObject;
                exit;
        }
        result = argv[0];
        i = 1;
        while (i < (argc - 1)) {
                op = argv[i];
                operand = argv[i + 1];
                if (op == "+") result += operand;
                else if (op == "-") result -= operand;
                else if (op == "*") result *= operand;
                else if (op == "/") result /= operand;

                i += 2;
        }
        display("result=", result, "\n");
        send "deleteYourself" to thisObject;
}

Calc1:delete() {}
```

Figure 10

We see here how we reference an array, using square-brackets: *array-name***[***subscript***]**.

The for**-Loop**

The **while**-construct we already know was used to iterate over the over array elements. This type of looping sequence is extremely common and OIL2 provides an alternative construct, the **for**-loop, which can be used to achieve the desired effect in less lines. The above loop can be expressed in terms of a **for**-loop:

```
        for (i=1;i < (argc – 1);i += 2) . . .;
```

The illustration above shows that the **for**-loop construction is written:

for (init**;**cond**;**mod**)** statement**;**

It is equivalent to:

```
init;
while (cond)
{
        statement;
        mod;
}
```

"*init*" is an expression that is evaluated once before anything else is done; typically, this is an assignment. "*cond*" is an expression that is evaluated before each execution of each pass of the loop. As long as "*cond*" evaluates to a non-zero (true)

result, the loop will be executed. "*statement*" is the statement (or block of statements) to be executed during each pass. "*mod*" is an expression that is evaluated at the conclusion of each pass; typically, this is an increment of some variable that is tested by the "*cond*" expression. Once "*cond*" is evaluated as zero (false), execution falls through to the next statement—the loop "*statement*" is not executed.

Complex Data Types

Many times programmers must deal with complicated data structures that cannot be represented by single instances of simple types, such as integers, strings and real numbers. OIL2 supports data types that address such problems: sparse arrays, associative arrays and sets.

Sparse Arrays

Arrays in many programming languages are dense, meaning that all the subscript indices must be contiguous (e.g., 0, 1, 2, 3, ...), and they must have their size pre-declared so that storage can be allocated ahead of time. In contrast, OIL2 arrays have the ability to be sparse, which means that there can be arbitrary gaps between subscript indices, and they never have a size declared. An OIL2 array can be subscripted by any 32-bit integer value. OIL2 arrays are declared using the type name **array**.

Often we want the data in an array to be in some kind of predefined order. The most obvious example would be to place the data in ascending order, so that the smallest element appears in the first element of the array, and the greatest element appears in the last. There are many ways to sort data, some much better than others. We'll look at an easily understood algorithm that really isn't that great with respect to performance.

We can sort an array by looking for the smallest element in the portion of the array we haven't yet sorted. When we find it, exchange it with the bottom of the unsorted portion. Repeat the process on the remaining elements (there is one less now) until all of the elements have been placed. The following program illustrates this:

```
%include <OMEcore.o2h>

class Local . Sort1 {
} inherits from Object;

Sort1:create()
{
   array data;
   int   i, j, n;
   int   min, pos;

   n = 10;     // number of elements
   for (i=0;i<n;i+=1) data[i] = n - i;
   display("Original:\n");
   for (i=0;i<n;i+=1) {
         display("data[", i, "] = ", data[i], "\n");
   }
   // now sort
   for (i=0;i<n - 1;i+=1) {
   // n - 1 because we don't need to do anything if
   // only one element is left
         min = data[i];
         pos = i;
         // check remainder of array for a smaller element
         for (j=i+1;j<n;j+=1) {
               if (data[j] < min) { // found a smaller element
                     min = data[j];    // remember it
                     pos = j;
               }
         }
         // pos now points to element with smallest value
         // in the unsorted portion of the array.  Exchange
         // with element i, the base of the unsorted
         // portion.
         data[pos] = data[i];
         data[i] = min;
   }
   display("Sorted:\n");
   for (i=0;i<n;i+=1) {
         display("data[", i, "] = ", data[i], "\n");
   }

   send "deleteYourself" to thisObject;
}

Sort1:delete() {}
```

Figure 11

Exercise 8

A good cryptographer finds ciphers trivial to solve. A cipher is essentially a coding scheme like ASCII or EBCDIC—a one-to-one correspondence of letters or symbols. These are simple to solve because cryptographers know that certain letters are used more often than others are and they can analyze the frequency of use of each symbol and make reasonable guesses as to what each represents.

Write a program that reads a file and counts how many times each letter is used. Treat upper case and lower case alike. When the end of the file is reached, print out the counts for the letters.

Associative Arrays

OIL2 supports another form of array that uses strings as subscripts instead of integers. These are called associative arrays and declared using the type name **assoc**. Associative arrays make it very easy to implement tables whose key is not a number. An example of such a table would be a telephone directory that maps a person's name to their phone number.

Sets

In OIL2, sets are a bit of a misnomer: they are more properly viewed as ordered lists rather than mathematical sets. Elements can be added or removed from sets. In OIL2, the individual elements of a set are processed using the **for**-**do** statement.

```
for var in setExpr do statement
```

The **for**-**do** loop accesses each element of the set *setExpr* in order. The value of a set element is copied to the variable *var* and then *statement* will be executed. As an example, consider the following fragment that finds the maximum element in a set:

```
int   minVal, v;
set   elements;

elements += 1;
elements += 3;
elements += 2;
// elements = { 1, 3, 2 }
minVal = 0;
for v in elements do if (v > minVal) minVal = v;
// v = 3
```

Functions and Methods

We have already been making extensive use of several functions, such as **display()** or **length()**. We have also invoked methods that returned a value, such as **readBytes** or **createObject**.

Why do we use functions or methods? A fundamental reason is to benefit from the efforts of other programmers.

One reason is because somebody has figured out how to do something we want to do, but don't know how. As an obvious example, consider the **display()** function: we don't actually know how to compose the required code to write to the standard output, but we can utilize the efforts of someone who did. In a similar vein, we can write a procedure or function to solve a common, often needed problem, such as calculating the square root of a number. By making a useful function available to other programmers, we make their jobs easier.

Frequently, programmers create a function or method that will be only used by their application. They do this to break a large problem up into smaller problems that are more easily understood and maintained.

Some languages make a distinction between procedures and functions. The difference is that a function implies that a value is returned to the caller, whereas a procedure does not. In the examples presented earlier in this document, we have used **length()** as a function (it returns the number of character in a string) and **display()** as a procedure (it returns the number of elements displayed, which was not of interest to our applications). OIL2 does not make the distinction between functions and procedures—they are all viewed as functions, but you can ignore the result that was returned.

In contrast, methods are special functions or procedures that are associated with a specific class. Some methods are able to return a value (and thus act as a function) while others do not (and thus act as a procedure). If a method is invoked with *fromObject* set to **nil**, then the return of a result is always inhibited. A method is normally invoked using a separate thread of execution; however, OIL2 also permits a method to be called as a function.

To illustrate a method that returns a value, let's write one that will calculate integral powers of an integer. The first problem is to come up with a name. How about "*power*"? Our function will take two arguments, say "*x*" and "*i*". We will return *x* raised to the *i*th power or x^i.

```
%include <OMEcore.o2h>

class Exponent {
} inherits from Object;

Exponent:create(int x, int i)
{
   int   result;

   result = call "power"(x, i);
   display(x, "**", i, " = ", result, "\n");
   send "deleteYourself" to thisObject;
}

Exponent:delete()    {}

Exponent:power(int x, int i)
{
   int   x_to_i, j;

   x_to_i = 1;
   for(j=1;j<=i;j+=1) x_to_i *= x;
   return (x_to_i);
}
```

Figure 12

The example in Figure 12 contains the first appearance of the "**return** " statement. If a method body does not return a value, then it exits either by "falling off" the end of the code or by encountering an explicit **exit** statement. In contrast, method bodies that return a result use the **return** statement. The value in the parentheses is returned to the routine that called the method body and execution of the called method body ceases.

Exercise 9

Write a method "*min*", which takes two integer parameters. It should return the smaller of the two parameters.

The code in Figure 12 deals with integers. What if we wanted to do the same thing for floating-point numbers? There are two approaches we can take in OIL2. The first is to take advantage of the **any** type and leave it to the runtime environment to perform the appropriate actions.

```
%include <OMEcore.o2h>

class Exponent {
} inherits from Object;

Exponent:create(int x, int i)
{
   any   result;           // was int, now any

   result = call "power"(x, i);
   display(x, "**", i, " = ", result, "\n");
   send "deleteYourself" to thisObject;
}

Exponent:delete()    {}

Exponent:power(any x, int i)     // was int x, now any x
{
   any   x_to_i;           // was int, now any
   int   j;

   if (typeOf(x) == int) { // add check for argument type
         x_to_i = 1; // integer
   } else {
         x_to_i = 1.0;     // floating-point
   }
   for(j=1;j<=i;j+=1) x_to_i *= x;
   return (x_to_i);
}
```

Figure 13

The code in Figure 13 is compact and easy to maintain. As a consequence of these attributes, it is the most commonly used approach by OIL2 programmers. The disadvantage is that additional CPU cycles will be required at runtime to make the appropriate determinations. The second alternative is to implement several methods with the same name but different types of arguments. This is called overloading in some languages. The correct method implementation will be automatically selected at runtime based upon the types of the arguments that are passed. In OIL2, overloaded methods are enabled by qualifying a method with the **unique** keyword. Because the type of the method's arguments are known at compile time, more efficient code can be generated for the method body.

```
%include <OMEcore.o2h>

class Exponent {
} inherits from Object;

Exponent:create(any x, int i)
{
   any   result;

   result = call "power"(x, i);
   display(x, "**", i, " = ", result, "\n");
   send "deleteYourself" to thisObject;
}

Exponent:delete()    {}

Exponent:power(int x, int i) unique
{
   int   x_to_i, j;

display("power int argv=", argv);
   x_to_i = 1;
   for(j=1;j<=i;j+=1) x_to_i *= x;
   return (x_to_i);
}

Exponent:power(float x, int i) unique
{
   float x_to_i;
   int   j;

display("power float argv=", argv);
   x_to_i = 1.0;
   for(j=1;j<=i;j+=1) x_to_i *= x;
   return (x_to_i);
}

Exponent:power(double x, int i) unique
{
   double      x_to_i;
   int   j;

display("power double argv=", argv);
   x_to_i = 1.0;
   for(j=1;j<=i;j+=1) x_to_i *= x;
   return (x_to_i);
}
```

Figure 14

Which style should one use? Most programmers do not have experience with languages in which the type of a variable or argument can be inquired at runtime, so they would tend to use overloaded methods out of habit. The disadvantage is that there can be significant duplication of code required. While such duplication increases the size of the executable, the critical disadvantage is that more effort is required of the programmer during initial development and future maintenance. Given that CPU speeds continue to increase at an incredible rate while human abilities show little improvement, OIL2 programmers typically avoid overloaded methods and write a single method that uses **typeOf()** and **if**-statements when special handling of different argument types is required.

Exercise 10

Write a method "*sort*" which takes two parameters: one an array, the other an integer denoting how many elements are in the array. The array should be the first

parameter. Your method should sort the array into ascending order and return the sorted array. The code in Figure 11 should be helpful.

Note: rather than force a programmer to maintain a variable that indicates how many elements are in an array, OIL2 applications would call the function **elementCount()**.

Exercise 11

OIL2 arrays have the intrinsic ability to be sparse. The functions **nextIndex()** and **indexExists()** are used to iterate over all of the elements both sparse and associative arrays. Write a method that takes a single argument and returns the value of the largest element in the array.

Exercise 12

Write a method "*revString* " which will reverse the contents of a string and return the computed result. Thus "123456" would become "654321".

Exercise 13

Write a method "*substitute*" which will take three string arguments. It will look for the first string in the third: if found, the portion corresponding to the first string will be replaced with the second. **Note:** this routine is a simplified version of the **substituteText()** function.

Exercise 14

Write a simplified version of the Unix "**fgrep**" (fixed generate regular expression pattern) utility. The **fgrep** utility accepts a string argument and then reads a file. It prints out on the standard output each line of the file that contains the string. This is a very useful utility. Your version will take two arguments: a string to search for and the name of the file to open. Read from the file using an **IOobject** and display lines that match using the **display()** function. Use of the **findSubstring()** function is suggested.

Exercise 15

Write a much-simplified version of the Unix utility "**sed**" (stream editor) that implements only a variation of the substitute command. The **sed** program reads a file and writes an edited version on the standard output. The **sed** directive "1,$s/string1/string2/" replaces the first occurrence of "*string1*" in a line with "s*tring2*". The modified line would be written to the standard output after the substitution. If *string1* does not occur in the line, the line is written out unchanged. Your version of **sed** will take two arguments: a string specifying a substitution pattern and the name of a file to be read using an **IOobject**. The substitution pattern should take the form "*#string1#string2#*". The "#" represents any given character. Thus the following are all valid and equivalent substitution patterns: "*Xstring1Xstring2X*", "*:string1:string2:*", or "*/string1/string2/*". Output should be performed using the **display()** function.

Global Variables

Up to this point, we have mostly been using local variables for computation. A local variable declared in one method is invisible to another. Since OIL2 variables are passed by value and not by reference, a method cannot modify the local variables of another method. Whenever we have needed to remember something between invocations of a method, we saved the information in an instance variable of the class. This permitted methods of the class to share information associated with a particular object and this is sufficient for the vast majority of situations. There are, however, some special occasions where we want to share information between all instances of a class. In OIL2, this capability is obtained by defining a variable within a **global** section. As an illustration, consider:

```
global {
   const string srcID = "$Id$";  // track RCS or CVS version Id
   int   totalObjects;           // a global count
};
```

Any methods within the source file that reference "*totalObjects*" will refer to the same variable. The two most common reasons for using a global variable in an OIL2 program are:

- maintaining a count
- performance-sensitive sharing of read-only information between two objects

Note: global variables have a purpose, but programmers are strongly cautioned to not use them unless necessary. Programmers who are not yet comfortable with object-oriented programming may gravitate towards using global variables, when they should have been declaring instance variables for their class.

Exercise 16

A stack is a very useful data structure. The classic example of a stack is that of a pile of plates. You "push" plates onto the top of the stack and likewise "pop" off the top plate when you remove one. This kind of behavior is termed last-in, first-out (LIFO). Stacks are used extensively in programs. Write a method "*push*" that takes an argument and a corresponding method "*pop*". These two methods should maintain a stack using instance variables. Thus the following code fragment:

```
   call "push"(2);
   call "push"(3);
   call "push"(1);
   a = call "pop"();
   b = call "pop"();
   c = call "pop"();
   display("a=",a, " b=", b, " c=", c, "\n");
```

would produce output looking like:

```
   a=1 b=3 c=2
```

If an attempt is made to pop a stack that is empty, **display()** an error message.

Preprocesor Directives

We have already made use of file inclusion to include the definitions of standard functions provided by the FARGOS/VISTA Object Management Environment. The "*%include*" directive tells the OIL2 compiler to read the specified file. If a file name is surrounded by "**<**" and "**>**" the compiler looks in a set of special locations for the include file. If it is surrounded by normal quotes, then the file name is used as-is.

Normally files that are included contain definitions of external functions or definitions of constants.

OIL2 source programs can be run through preprocessors. The **oil2** shell script provided with the FARGOS/VISTA Software Development Kit passes OIL2 source code through the C preprocessor before invoking the OIL2 compiler (the **oil2_parse** executable). Because OIL2 source can be compiled to OIL2 Architecture Neutral Format object code without requiring a C++ compiler, some developer machines may not have a C preprocessor available. Thus, if guaranteed universal portability is required, C preprocessor macros should be avoided. Many developers, however, write code only for their own organizations and are able to dictate the environment in which they work. If they know a C preprocessor is available on their system, then they should feel no constraint regarding their use of the **oil2** shell script. For those users, a discussion of some of the C preprocessor features is provided below. It can be safely skipped by anyone not interested in utilizing such facilities.

Macro substitution is a powerful feature of the C preprocessor. In its simplest form, it consists of normal text substitution. More advanced macros can be written that take parameters. Macros are defined using the "*#define*" preprocessor directive. Historically, the most common use of this feature in C programs was to define constants that may be changed. For example, a buffer may be declared to be an arbitrary size, and the routines that work on it must know how big it is. It is considered bad practice to "hardwire" these constants into a program, because it would be difficult to change the buffer size if desired. The way around this is to define a macro that represents the size of the buffer:

```
#define BUF_SIZE 200
```

A C/C++ program can then use the macro "*BUF_SIZ*" any place it wants to talk about the size of the buffer:

```
char buffer[BUF_SIZE];
```

or

```
   if (bfr_pos >= BUF_SIZE) printf("Buffer overflow!\en");
```

By convention, C programmers use all capital letters when naming macros that do not have arguments (symbolic constants). They then stand out in a program and can't be mistaken for variables. You should always follow this practice.

Note: rather than use *#define*'d constants, OIL2 programmers should use the **const** declaration that is part of the language. For example:

```
   const int BUF_SIZE = 200;
```

Because of the **const** declaration in OIL2, simple substitution macros have little utility in OIL2 source code. The macros supported by the C preprocessor can also take arguments. Such complex macros can be of use to OIL2 programmers because they can write macros that look like functions or procedures.

```
#define addOne(x) ((x) + 1)
```

Note that the macro definition above did not put a semicolon on the line because we expect the "user" to do that when he uses the macro. Using macros instead of a true function call can result in considerable savings of CPU time because an actual function call is not performed when the program runs.

Finally, the C preprocessor supports conditional compilation. There are several directives that can be used, but we'll illustrate the most general. To motivate this

discussion, consider the process of debugging a program. Normally an OIL2 programmer puts in a lot of **display()** statements to print out the contents of variables at different points in the program. When the program is finally debugged, these statements must be deleted somehow. More often than not, after the programmer thinks everything is OK, another bug is found and the **display()** calls have to be restored. The C preprocessor lets us do the following: we define a simple macro/constant, say "*DEBUG*" and enclose the **display()** calls in between conditional directives:

```
#define DEBUG
/* lot of code here */
#ifdef DEBUG
display("This is a debugging line\en");
#endif
```

The #*ifdef* directive asks if the argument is defined. So in this example it is asking if *DEBUG* is defined. It was defined at the very top of the program. Since the parameter was defined, all of the source code until the #*endif* is used. If not, then that code is ignored. There is also a "#*else*" directive available, which is illustrated below:

```
#ifdef DEBUG
display("This is a debugging line\en");
#else
display("This is the normal output line\en");
#endif
```

Thus by commenting out the #*define DEBUG* line at the top of the program, the compiler only sees the final program. At any time we can uncomment the #*define DEBUG* line, recompile, and have the debug statements work. **Note:** the **debugDisplay()** function yields similar functionality, but is more powerful because its usage does not require a recompile.

This is just one use of conditional compilation. Historically, the most popular use is to permit programmers to write portable programs—they can use #*ifdef*'s and be able to write code that will run under a variety of operating systems. Depending on what operating system the code is to run under, different macros will be defined and thus different segments of code will be compiled. Given the architecture neutral characteristics of the OIL2 runtime, such usage is rare indeed with OIL2 source.

Multi-Dimensional Arrays

We have found arrays to be very useful, but up to this point we have only used one-dimensional arrays. It is not always convenient to be restricted to using one-dimensional arrays. In OIL2, an array element can hold arbitrary data, including another array. This is how multi-dimensional arrays are supported in OIL2.

Not yet processed

Up to this point, we have said that the local variables we declared were created when a subroutine is entered and destroyed when the routine exits. This meant that there was no way for a subroutine to remember something from one call to another unless it used a global variable. Global variables necessarily always keep their values: they are created at the time the program is loaded and disappear only when the program finishes execution.

By default, local variables exist only for the life of the current method invocation.

We can think of global variables as having being static in the sense that they are never destroyed. We know that when we declare a global variable in a source file, the compiler then knows that the variable is global when we reference later in the source file.

Global arrays are useful for other reasons. Recall that if an array is global, it is never destroyed--it also stays in the same spot. Let's write a function that will take a character, fill a string with its two-digit hex value and return a pointer to that string:

```
/* lib hexconv:  convert character to hex representation */
char *hexconv(c)               /* returns pointer to a string */
char c;
{
        /* allocate 2 digits, mark end with '\e0' character */
        static char array[] = {'?','?','\e0'};
        static char digits[] = {'0','1','2','3','4','5','6','7',
                                '8','9','A','B','C','D','E','F'};
        int i;

        i = c / 16;             /* i = c divided by 16 */
        array[0] = digits[i];   /* get first digit */
        i = c % 16;             /* i = c MOD 16 */
        array[1] = digits[i];   /* get second digit */
        return (array);         /* return pointer to string */
}
/* endlib:  hexconv */
```

We could not do this if "\fBarray\fR" was not static: when the routine returned the array would be destroyed. Note the use of "%" to denote the MOD operation. This is sometimes useful, such as in this example. We can illustrate the convenience of returning a pointer to a string with the following program:

```
main()
{
        char *hexconv();        /* declare function */
        char a;

        a = '\en';              /* let's find out about newline */
        printf("decimal=%d, hex=%s\en",a,hexconv(a));
}
```