5 O_2C Code

METHOD, FUNCTION AND PROGRAM BODIES

The syntax you use for defining a body is a superset of standard C, with an extended type system and additional operations to support the O_2 data model.

This chapter describes the code part of O_2C:

- O2C variable declarations
- Atomic types
- Structured types
- Manipulating objects
- Type compatibility
- Calls
- Transaction instructions
- Building schema libraries
- Using O2APi in O2C
- Compiler options
- O2C language syntax

5 O2C Code

5.1 O_2C variable declarations

You can declare ordinary C declarations and O_2C declarations of O_2 data types together.

All O_2 type declarations begin with the keyword **o2** with the syntax:

```
o2 type_spec variable_name [= initial_value]
[, variable_name [= initial_value]]...;
```

- ***Atomic type***

  To declare O_2 atomic type (**integer**, **real**, **char**, **string**, **bits**, or **boolean**) and C declarations:

```
o2 integer n;
o2 char* s;
o2 string nickname = "Fifi";
```

- ***Structured type***

  To declare O_2 structured type (**tuple**, **set**, **unique set** or **list**):

```
o2 set (integer) lucky_numbers = set (4,27,888);
o2 tuple (name: string, age: integer) student;
```

- ***Class name***

  With an O_2 class name the variable represents an object rather than a value. For example, assuming that **Person** and **City** are defined classes:

```
o2 Person sophie, tom, rose;
o2 City x;
```

- ***Type***

  In the construction **type (***Class_name***)**, the variable represents a value rather than an object, having the data type of the class *Class_name*.

```
o2 type (Person) a, b;
o2 type (City) y;
```

- ***Named type***

  You can also use an O_2 named type. The defined variable represents a value with the structure of the named type. Note that you cannot use the C **typedef** command with O_2 types.

5.2 Atomic types

The six atomic types defined in O_2C are **integer**, **real**, **boolean**, **char**, **string** and **bits**. You declare all these types using the **o2** declarator and manipulate their values using operators conforming to C.

Integers

O_2C integers behave the same as C **int** variable with identical value ranges:

```
o2 integer i, j = 4, k;
k = i + j;
```

The O_2C variables **i**, **j** and **k** are declared as integers with **j** initialized to 4, and **i** and **k** initialized to the default zero. In expressions, the C **int** variable and O_2 **integer** are freely interchangeable and mixed. Any assignment, arithmetic and comparison operations on **int** variables also apply to **integer** variables.

Real numbers

O_2C reals behave in the same way as C **double** variable with identical value ranges:

```
o2 real x, y, z;
x=0.02;
y = x*y;
if (z == x)
....
```

The O_2C variables **x**, **y** and **z** are declared as real numbers with **x**, **y** and **z** all initialized to the default value zero. In expressions, the C **double** variable and O_2 **real** are freely interchangeable and mixed. Any assignment, arithmetic and comparison operations on **double** variables also apply to **real** variables.

Boolean values

You use O_2C booleans in the same way as C conditional expressions:

```
o2 boolean a = true, b;
...
b = (x == y);
...
if (b)
...
```

The O_2C variables **a** and **b** are declared as booleans with **a** initialized to **true** and **b** initialized to the default value **false**.

5 O2C Code

Characters

In O_2C **char** variables behave in the same way as **char** variables in C:

```
o2 char a ="K", b;
```

The variables **a** and **b** are declared as characters with **a** initialized to the letter **K** and **b** initialized to the default null character (ASCII zero). In expressions, the C **char** variable and O_2 **char** are freely interchangeable and mixed. Any assignment and comparison operations apply to both C and O_2 **char** variables.

Bits and strings

You can declare **string** as follows:

```
o2 string m, message = "This is a string.";
```

This defines the string variables **m** and **message** as strings with **m** initialized to the default empty string and **message** initialized to **"This is a string."**. You can declare **bits** as follows:

```
o2 bits q;
```

This defines the bits variable **q** as a bit strings. It is initialized to the default string of length zero.

Both strings and bits end with a zero byte.

Note

The difference between a character string and a bit string is that any byte of a bit string can contain any value including zero and can therefore include one or more embedded zeros. To include embedded zeros in a bit string, use the **strncpy** function described below.

The following operations may be performed on both string and bits variables:

- ***Assignment: x1 = x2***

  x1 = x2 copies the string or bits variable **x2** into **x1**.

- ***String length: count (s), count (b)***

  count (s) returns the length of string **s** and does not include the zero byte that ends the string.

  count (b) returns the exact length of the bit string **b** including any embedded zero bytes.

- ***Comparison: x1 == x2, x1!= x2***

  x1 == x2 is **true** if the string or bits **x1** is equal to **x2**.

  x1!= x2 is **false** if the string or bits **x1** is equal to **x2**.

- ***Concatenation: x3=x1 + x2, x1 += x2***

  X3 = X1 + X2 creates a new string or bits by concatenating **x2** and **X1**.

  X1 += X2 or **X1** = **X1** + **X2** appends **X2** onto the end of **X1**.

- ***Update: x[i]= a_char***

  Replaces the ith character by a character.

- ***Access: x[i]***

  X[i] returns the ith character in string or bits **X**. Note that numbering starts at zero. The first character in the bits or string is **X[0]** and the last is **X[(count(X) -1)]**

- ***Membership: (p in x)***

  (**p in X**) gives the ordinal number where **p** first occurs in the string or bits **X** or a negative number if **p** does not occur in **X**.

- ***Sublist: x[i:j]***

  X[i:j] returns a sublist of the string or bits **X**, from the ith to the jth characters. **X[i:]** is the same as **X[i:(count(X)-1)]**, and **X[:j]** is the same as **X[0:j]**.

- ***Replacement: x1[i:j] = x2***

  X1[i:j] = X2 replaces the ith to the jth characters of the string or bits **X1** with **X2** (**X1** and **X2** do not have to be the same length). If **i** and **j** are the same integer then only a single character of **X1** is replaced by **X2**.

- ***Clearing: x[i:j] = ""***

  X[i:j] = "" removes the ith to the jth characters of the string or bits **X**. If **i** and **j** are the same only a single character is removed from **X**.

- ***Insertion***

  X1[:i] += X2 inserts **X2** into **X1** *after* the ith character of **X1**.

  X1[i:i] = X2 + X1[i:i] inserts **X2** *before* the ith character of **X1**.

- ***Iteration: for (c in x) {instructions}***

  for (c in X) {*instructions***}** repeats the *instructions* for each character in the strings or bits **X** and the char variable **c** takes on the value of each character of **X** in turn. You can add a **where** modifier to limit the number of iterations:

  for (c in X where *condition***) {***instructions***}**

  The *instructions* are carried out each time the *condition* is met.

- ***C library functions***

String and bits variables behave differently from C **`char *`** strings. However, you can pass a string or bits variable to any C library function that calls for a **`const char *`** parameter, but a library function cannot alter the contents of the string or bits.

For instance, a **`string`** can be used:

- As the second argument to **`strcat`**, and **`strncat`**.
- As either or both of the arguments to **`strcmp`**.
- Anywhere in a **`printf`** function even as the format string.

There are two special functions which manage the conversion between **`char *`** and **`string`**: **`strcpy`** and **`strncpy`**. You use these functions to copy a C **`char *`** string or **`bits`** variable to an O_2C string variable, and back again.

In the following example, the **`strcpy`** function copies the value of the character string pointed by **`p`** to the first zero byte in the string **`str`**:

```
o2 string str;
char *p = "This is a string.";
strcpy (str, p);
strcpy (p, str); /* do the reverse */
```

In the example:

```
o2 bits b;
char *p= "This is a \0 bit string.";
strncpy (b, p, 20);
strncpy (p, b, 20); /* do the reverse */
```

The function **`strncpy (b, p, n)`** copies the value of the substring of length **`n`** pointed by **`p`** up to the first **`n`** bytes including all zero bytes. For example, **`strncpy (b, p, 20)`** copies the first twenty bytes of **`p`** into **`b`** including zero bytes i.e., "**`This is a \0 bit stri`**".

Note

The overloaded **`strncpy`** is not identical to the C standard as it does not stop at the first zero byte that it encounters.

Using a string as a parameter of a C function which has a side effect on it can lead to catastrophic failures!

5.3 Structured types

The structured types defined in O_2C are **tuple**, **set**, **unique set** and **list**. You declare all these types using the **o2** declarator and manipulate their values using operators conforming to C.

Tuples

Tuple attributes are initialized in the same way as atomic values. Integers and real numbers are initialized to 0, characters, bits and strings to null, booleans to false and sets and lists to empty. Nested tuples are initialized recursively. Objects are initialized to the nil object.

For example:

```
o2 tuple (name: string, age: integer) you;
```

The variable **you** is an instance of the tuple.

As with C structures, you can extract or assign tuple attributes individually using the dot operator:

```
you.age = 54;
```

An object with the tuple type structure behaves in the same way as a pointer to a C structure. You can extract an attribute using the **->** operator (according to the encapsulation level):

object->attribute

For example:

```
o2 tuple (last_name: string, first_name: string) name1, name2;
o2 Person foo, bar;

 foo = new Person;
 foo->name = tuple (last_name: "Spade", first_name: "Sam");
 name1 = tuple (last_name: "Tirebiter", first_name: "George");

 bar = new Person;
 bar->name = name1;

 name2 = foo->name;
 foo->name.first_name = "George L.";
 name2.first_name = foo->name.first_name;
```

5 O2C Code

Sets

A set is an disordered collection of objects of the same (super) class or a collection of values of the same type.

For example,

```
o2 set (integer) lucky_numbers;
```

This defines the variable **lucky_numbers** as a set with the default value of the empty set. You can also declare the set with specific values, e.g.

```
o2 set (integer) lucky_numbers = set (54, 73, 77, 77);
```

In this type of ordinary set, the same object can appear several times. However, you can also declare a unique set in which the same object can only appear once.

For example,

```
o2 unique set (Person) kids;
```

The variable **kids** is a unique set of objects of class **Person**, which is initialized to the default empty set **unique set()**.

The following operations are possible on set values where **X**, **Y** and **Z** are sets:

- ***Assignment: x = set (9, 9, 10)***

  X = set (9, 9, 10) specifies that X is an ordinary set containing 3 elements. A set can be converted to unique set with any duplicate elements being automatically removed, e.g.

  ```
  o2 set(integer) X=set(5, 7, 5, 6, 5);
  o2 unique set(integer) Y;
  Y= (o2 unique set (integer)) X;
  ```

  Y is the **unique set (5, 7, 6)**.

- ***Comparison: x==y x!=y***

  X == Y is **true** if **X** and **Y** contain exactly the same elements and exactly the same number of elements; otherwise it is **false**.

  X!= Y is **false** if **X** and **Y** contain exactly the same elements and exactly the same number of elements; otherwise it is **true**.

- ***Union: z = x + y***

  Z = X + Y makes **Z** a set containing the elements of both **X** and **Y**. If **Z** is a unique set, any duplicate elements are removed.

- ***Size: count (x)***

  count (X) returns the number of elements in set **X**.

- ***Intersection: z = x * y***

  Z = X * Y makes **Z** a set containing the elements common to **X** and **Y**.

- ***Difference: z = x - y***

  Z = X - Y makes **Z** a set containing the elements of **X** which are not in **Y**.

- ***Membership: (p in x)***

  (p in X) is the boolean **true** if **p** is an element of set **X** and **false** if not.

- ***Insertion: x += y or x = x + y***

  X += Y or **X = X + Y** add the elements of **Y** to **X**. If **X** is a unique set, any duplicate elements are removed. You must write **x+=unique set(p)** to add a new element.

- ***Deletion: x -= y or x = x - y or x*= y***

  X -= Y or **X = X - Y** remove any elements of **Y** from set **X**.

  If **x** is an ordinary set, elements are removed only as they occur in set **Y** and you may find that duplicates of the same element remain in **x** after this removal operation.

  For example, if **X** is **set (1,2,3,3)**, the operation **X -= set(3)** removes only one of the elements **3** from **X**.

  X *= y removes all elements from **x** except those elements of **y**.

- ***Iteration: for (q in x) {instructions}***

  for (q in X) {*instructions*} repeats the instructions for each element of set **X**. The variable **q**, of the same type as the elements of **X**, takes on the value of each element in turn. You can add a **where** modifier if you want to limit the number of iterations to a subset of **X**:

  for (q in X where *condition***)** {*instructions*}

  The *instructions* are carried out once for each element of **X** satisfying the boolean *condition*..

- ***Singleton extraction: element (x)***

  If **x** contains only one element, it returns this element otherwise an exception is raised.

5 O2C Code

Lists

A list is an ordered collection of objects belonging to the same super class, or it is a list of values of the same type. A list can contain the same element more than once. You can also declare an empty list, **`list()`**, as in the following example where the variable **`group`** is a list of tuples initialized by default to the empty list.

```
o2 list (tuple (name: string, age: integer)) group;
```

A list is initialized by default to an empty list. You can also initialize parts of a list using the syntax:

```
number: value
```

For example, if you take an integer **`p`** with value 2:

```
list (1, 2, p+5, 4:0, 3*p:9, 100)
```

you obtain the following list:

```
<1,2,7,0,0,0,0,9,9,9,9,9,9,100>
```

The following operations are possible on list values:

- ***Assignment: x = y***

  `X = list(1, 2)` specifies that the list **`X`** contains two elements **`1`** and **`2`**.

- ***Comparison: x == y, x!= y***

  `X == Y` is **`true`** if **`X`** and **`Y`** contain the same elements in the same order; otherwise it is **`false`**.

  `X!= Y` is **`false`** if **`X`** and **`Y`** contain the same elements in the same order; otherwise it is **`true`**.

- ***Concatenation: z = x + y, x += y***

  `Z = X + Y` gives list **`Z`** containing elements of **`X`** followed by elements of **`Y`**.

  `X += Y` (or **`X = X + Y`**) appends the elements of list **`Y`** to list **`X`**.

- ***Size: count (x)***

  `count (X)` returns the number of elements in list **`X`**.

- ***Direct access: x[i]***

  Each list element has an ordinal number with which you directly access the element. **`X[i]`** returns the value of the ith element of list **`X`**. Note that numbering starts at zero so that the first element in **`X`** is **`X[0]`**. The last element is **`X[count(X)-1]`**.

- ***Update: x[i] = element***

  Replace the ith element of the list.

- ***Membership: (p in x)***

  (p in X) gives the ordinal number where **p** first occurs in list **X**. If **p** is not an element of **X**, a negative value is given.

- ***Sublist: x[i:j]***

  X[i:j] returns a sublist of **X** from the ith value through to the jth value.

  X[:j] is the same as **X[0:j]**, and **X[i:]** the same as **X[i:(count(X)-1)]**.

- ***Replacement: x[i:j] = y***

  X[i:j] = Y replaces the ith through to the jth elements of **X** with the list **Y**. (List **Y** can contain more or less elements than the sublist being replaced). If **i** and **j** are the same integer then only one element of **X** is replaced by the elements of **Y**.

- ***Clearing: x[i:j] = list()***

  X[i:j] = list() removes the ith through to the jth elements from list **X**. If **i** and **j** are the same then only one element of **X** is removed.

- ***Insertion: x+= list(p)***

  X += list(p) adds the element **p** to the end of list **X**.

  X[:i] += list(p) inserts the element p *after* the ith element of **X**.

  X[i:i] = list(p) + X[i:i] inserts the element **p** *before* the ith element of **X**.

- ***Iteration: for (p in x) {instructions}***

  for (p in X){*instructions*} repeats the *instructions* for each element of **X**. The variable **p**, of the same type as the elements of **X**, takes the value of each of these elements in turn.

  You can add a **where** modifier if you want to limit the number of iterations to a sublist of **X**:

  for (p in X where *condition***)** *{instructions}*

  The *instructions* are carried out once for each element of **X** satisfying the boolean *condition*.

5.4 Manipulating objects

This section is divided up into the following subsections:

- Object creation - new and init
- Object assignment - obj1 = obj2
- Object comparison - obj1 == obj2, obj1 != obj2
- Object values
- Object disencapsulation: * operator
- Object properties: -> operator
- Object sharing
- Copying values
- Object persistence
- Object deletion

Object creation - new and init

You create objects in the O_2 database in method and program bodies using the **new** command, e.g.

```
tintin = new Person;
```

This creates a new object of the class **Person**. The **new** command initializes the object because it automatically calls a special system supplied method called **init** which in turn initializes the value encapsulated by the new object. By default this value is initialized to the simplest form of its type:

- Integers and real numbers to 0.
- Strings and bits to the null string.
- Char to the null character.
- Boolean to false.
- Sets and lists to empty.
- Tuples follow the same rules for each of their attributes.
- Nested tuples are initialized recursively.
- Objects to the **nil** object.

Every class has its own **init** method which has no parameters and which you cannot delete. The **init** method is also the only O_2 method which cannot be inherited. You can, however redefine an **init** method with or without parameters. This then overrides the system-supplied **init** method:

```
method body init(name:string, age:integer) in class Person
{
self-> family_name=name;
self-> age=age;
}
```

Use the **new** operator to give the parameters which are then sent to **init** e.g.

```
tintin = new Person ("Tintin", 23);
```

As **new** returns the object, a method can be sent immediately to the object:

```
(new Person ("Captain Haddock", 37)) -> display;
```

You can avoid the implicit call to the **init** method using the **void** keyword:

```
tintin = new void Person;
```

You can always delete this redefined **init** method in which case the system-supplied **init** takes over again.

Object assignment - obj1 = obj2

The expression:

```
obj1 = obj2;
```

sets the object variable **obj1** to represent the same object as the object represented by **obj2**.

Important

This assign sets the value of the object to be shared and any later modification to one of the objects is automatically reflected in the other object.

If **obj1** referred to some other object before the assignment, this is overridden and it no longer refers to that object.

Object comparison - obj1 == obj2, obj1 != obj2

The expression:

obj1 == obj2 or **obj1 != obj2**

contain operators that compare object identities.

For example:

```
if (obj1 == obj2) {
...
}
```

This conditional statement is only true if obj1 and obj2 are the same object. If they are two different objects even with values that are completely equal, the statement is false.

Object values

Operations on the values encapsulated by objects are only allowed if the object encapsulation is adhered to. Only methods attached to the object class of the object or subclass can read or write private attributes.

However, any method, program or function can read read-only attributes but only those methods attached to the object class or

subclass can modify their values. Public attributes can be read or modified anywhere.

Object disencapsulation: * operator

An object is not the same as a value. An object encapsulates a value and its behavior. The relationship between an object and its value loosely corresponds to the idea of pointers and the values to which they point in C. To obtain the value of an object you use the operator *****.

For example, take the following class:

```
class Person
public type tuple (name: string,
                   age: integer)
end;
```

and the following code:

```
{
o2 Person tintin = new Person ("Tintin", 23);
o2 tuple (name:string,
          age:integer) value;
value = *tintin;
}
```

The * operator separates the value from its object enabling you to reach the object value. You can get the attribute value using the dot operator. For example, for the value of the **name** attribute use **: value.name** (**"Tintin"**) and the **age** attribute: **value.age** (**23**).

However, instead of using **value**, you can also use the expression ***tintin**, e.g.:

```
(*tintin).name
```

In O_2, the ***** operator is said to disencapsulate the object.

Disencapsulation can always occur in a method attached to or inherited by the object class or one of its subclasses.

In any other method, or in any application program or function, disencapsulation can only occur with the public or read tuple attributes of the object's value.

In any other circumstances, disencapsulation violates the encapsulation rules between data and behavior in object-oriented systems, and is thrown out by the compiler.

Since the value of an object is usually a structured value, all of the structured-value operations are allowed on the disencapsulated value.

Object properties: -> operator

To reach the field of a tuple type object you use the operator **->**.

```
object -> attribute.
```

Taking the class Person you can use the -> operator in the following way:

```
{
o2 Person tintin = new Person ("Tintin", 23);
tintin -> age += 1
}
```

Consequently **tintin -> age** is 24.

You can also invoke a method attached to an object's class using the **->** operator:

```
obj1->method_name
obj1->method_name (argument [, argument]...)
```

If the method returns a value, the method expression takes on the data type of the returned value.

Object sharing

An object can be shared but a value cannot.

```
class Person
    type tuple (name: string,
                age: integer,
                mother: Person,
                father: Person,
                spouse: Person,
                children: list(Person))
end;
```

An object of the class **Person** contains several other objects of class Person as some of its tuple attributes. Any one of these objects can be a component of several other objects.

For example, an object representing a person John appears as the **spouse** attribute of the object representing his wife and as a **father** attribute for his children and as an element of the **children** list of his parents.

All these references are to the same physical object John. If the object John is modified, e.g. his age, in the object representing his father then the object John is automatically updated in all the other objects.

Copying values

You cannot share values but you can copy them.

For example, if objects **p1** and **p2** of class **Person** are set by **p1->name = p2-> name = "Joe"**, the two objects have the same value. But if later **p2->name = "Dupont"**, the two objects now have different values.

Important

Objects have an identifier and can be shared, values do not have an identifier and cannot be shared.

Object persistence

Objects and values do not automatically remain in the database after the program that created them has terminated.

Programs can create temporary objects and values which are discarded when the program ends. To store objects and values in the database, you must make them persistent.

- ***Persistence***

  An object or value is persistent if:

  - It has a specific name in the schema.
  - It is attached to a named value or a named object in a program.

- It is attached to another persistent object in a program.

In other words, any object reachable from a persistent object is persistent. No other objects or values are persistent

Once you have created an object using the **new** instruction, you can make it a component of a named object or named value. This associates it with a schema name and thereby makes the object persist in the database.

For example, take a named value **People**, defined as a set of objects of class **Person**. The following O_2C instruction renders an object variable **p** of class **Person** persistent:

```
People += set (p);
```

- ***Constant names***

The **constant name** is equivalent to C **const** qualifier as applied to a pointer.

The value pointed to can change but not the pointer itself. The only way to modify such a pointer is to use the **modify constant name** command.

For example, you can add a year to the value of the **age** attribute for the constant name **Tintin**:

```
Tintin->age += 1;
```

However, you cannot set the object **Tintin** to be equal to another object:

```
Tintin = Joe;
```

For a named object, therefore, any operation is allowed on the object value but not on the object itself, i.e it cannot appear on the left-hand side an assignment.

For a named set value, you can only use the following operations:

- Insertion of elements: **X += Y**
- Removal of specific elements: **X -= Y** or **X *= Y**
- Removal of all elements: **X = set()**

For a named list value, you can only use the following operations:

- Appending of elements: **X += Y**
- Insertion of elements: **X[:j] += Y**
- Replacement of specific elements: **X[i] = e or X[i:j] = Y**
- Removal of all elements: **X = list()**

A **constant name** value cannot appear at the left-hand side of an assignment in any other circumstances.

Object deletion

To delete an object, you must remove all sources of its persistence.

This means that you must delete all name associations, whether direct or indirect.

For a named object, you must delete the name using the **`delete name`** command.

If the object is a member of a persistent set or list, it must be removed in the same way that it was added.

For example to remove the object **`p`** from **`People`**:

```
People -= set(p);
```

If the object is a component of any other persistent object, or an attribute of any persistent tuple value, then you must break these links by assigning **`nil`** to them.

For example:

```
John -> spouse = nil
```

5.5 Type compatibility

In O_2C, you can often replace one structured type by a more specialized subtype, without affecting the functioning of the method or program concerned. This is called type compatibility.

In assignment expressions with O_2 structured values, the value type on the right of the = sign can be a subtype of the value on the left side. For example:

```
o2 tuple (name: string, address: string, phone: string) x;
o2 tuple (name: string, address: string) y;
...
y = x;
```

The type of **x** is a subtype of the type of **y**. The **name** and **address** attributes of **x** are copied into the corresponding attributes of **y**, and the **phone** attribute of **x** is ignored. However, the assignment **x = y** is rejected as **y** is not a subtype of **x** and the compiler does not know how to create the **phone** attribute of **x**.

Sometimes you do not know at compile time what the type of your object will be at runtime. If you consider two objects: **x** of class **Employee** and **y** of class **Person**, with **Employee** a subclass of **Person**, the following assignment works:

```
*y = *x;
```

The type of ***x** looks like it is a subtype of ***y**. However, at run-time, the object **y** can turn out to be an instance of any subclass of **Person**, and its type can have attributes unknown to class **Employee**. The system therefore must check at run-time. The assignment is legal if and only if ***x** is of the same type or a subtype of ***y**. Otherwise, an exception occurs.

In both cases, information transferred from **x** to **y** is incomplete with the subtype's additional specialized attributes being ignored.

Note

With objects, this information loss does not occur. If you take the same two objects, **x** of class **Employee** and **y** of class **Person**, then the assignment: **y = x;** simply makes the two variables **x** and **y** refer to the same object **x** of class **Employee**. All attributes defined for the class **Employee**, including the specialized ones, are available to both variables.

Casting

Casting in O_2C syntactically resembles casting in C but semantically, there is an important difference. The syntax of a cast is:

(o2 *type_spec*) *expression*

At runtime, a cast object must be of the type specified by the cast, otherwise a run-time exception occurs. Casting is useful in making a method more generic and therefore more reusable. There are three types of casting.

- ***General casting situations***

  You must cast an expression when the compiler cannot infer its type. Since the O_2C keyword **nil** refers to any object, in some contexts the compiler needs to know the class of the object.

  For example, if **s** is declared as set of objects of class **Person**, to specify that this set contains three nil objects, you must use the following cast:

  ```
  s = set ((o2 Person) nil, nil, nil);
  ```

  More generally, an aggregate construction with a set or a list has a static type determined by its first element.

  For example, take a hierarchy where **Employee** and **Student** are subclasses of the class **Person** and the variables:

  ```
  o2 Employee e;
  o2 Student s;
  ```

  The aggregate **set(e, s)** does not compile because the compiler cannot infer the type from it. You must indicate the type by casting the first element of the aggregate **set((o2 Person) e, s);** The compiler then knows that its type is **set(Person)**.

- ***More complex casting***

  Other situations also require casting. Consider the following example:

  The first is **Group**, defined as a set of objects of class **Person**. There is a method called **leader** which returns the person who is the group leader.

```
class Group
   type set (Person)
   method leader: Person
end;
method body leader: Person in class Group {
    ...
  for (p in *self) {
    ...
  }
  return p;
 }
```

The other class is **Team**, a subclass of **Group**. A team is a set of objects of class **Employee** (which is a subclass of **Person**). This class is given below.

The class **Team** also has a method **leader**, but we want to specify that the leader is an object of class **Employee**, not just of class **Person**.

```
class Team inherit Group
   type set (Employee)
   method leader: Employee
end;

method body leader: Employee in class Team {
  o2 Person boss;
  boss = self->Group@leader;
  return (o2 Employee) boss;
}
```

The **leader** method of class **Team** calls the **leader** method of class **Group** (using the **@** notation, which returns an object of class **Person**).

However, since the **leader** method of class **Team** returns an object which is of class **Employee**, it casts it as such before returning it. This allows the code of the **leader** method of **Group** to be called (reused) instead of repeated within the same method of class **Team**.

- ***Situations requiring an O_2 type***

  Finally, C expressions must be cast when used in situations requiring an O_2 type. Consider a list of integers: `o2 list(integer) L;`

  You can add an element to the list in the following ways:

```
int x;
o2 integer y;
L+= list((o2 integer)x);
L+= list(y);
```

  or

```
int x, y;
o2 integer a, b;
L+= list((o2 integer)(x+y));
L+= list(a+b);
```

5.6 Calls

You call programs and functions within your application according to the following rules.

Program call

Programs within an application can call each other, pass parameters but cannot return values to the caller.

All programs can call functions and they can call public methods.

You do not need to re-declare application variables as they are known to all programs associated with the application. However, they can be passed as parameters to methods called by the programs.

Application variable values can be accessed across transaction boundaries but they do not persist after the application has terminated.

- ***Syntax***

One program calls another by using the same syntax as if it were calling an ordinary C function:

```
travel_information();
add_reservation (ticketB);
```

However a program does not return a value.

Note

Brackets are required when calling an O_2 program from O_2C code, even if the program takes no parameters.

```
program_name ([arg_name [, arg_name]...])
```

Function call

Functions are parts of program code that exist in a schema which are not methods and are not included in any applications.

You can call a function from anywhere within the schema in which they are defined: from methods, application programs or other functions.

Functions can call public methods and access public values, and are useful for defining operations on values as values have no methods attached to them.

- ***Syntax***

The syntax for calling an O_2 function is identical to that for an ordinary C function.

However, O_2 functions are known to a schema and therefore do not have to be explicitly linked to the calling code.

A function can have arguments and they must be specified between brackets, just after the function name:

```
function_name (arg_name [, arg_name]...)
```

Note

As with C functions, brackets are required when calling an O_2 function from O_2C code, even if the function takes no parameters.

For example, a function which sorts a set of people and returns a list:

```
function psort (group: set (Person)): list (Person)
```

A function may or may not return a value to its caller.

5.7 Transaction instructions

When an application starts up, it opens by default a “read-only” transaction. To program transactions the following O_2C instructions are available:

- **`transaction`**: starts a new transaction.
- **`validate`**: commits the updates carried out by the current transaction and then releases all the locks on which other transactions may be waiting.
- **`commit`**: is the same as the “validate” transaction except that it also frees all temporary memory thereby invalidating all temporary variables and interrupting the current execution flow. A “restart procedure” is then undertaken.
- **`abort`**: cancels all the updates carried out by the current transaction, interrupts the execution flow and undertakes a restart procedure.

Restart

The restart program takes a first argument provided by the system, of type **`integer`**.

The file **`o2_event.h`** contains the definitions of constants providing the possible arguments for a restart:

- **`O2_COMMIT`** — after a commit instruction
- **`O2_ABORT`** — after an abort instruction
- **`O2_ERROR`** — after a run-time error
- **`O2_DEADLOCK`** — after an abort caused by a database deadlock

The restart program may take more arguments which the are global variables for a transaction.

Note

For more details on the restart program, refer to Section 4.2.

Handling exceptions

O_2 uses the standard UNIX exception handling mechanism. However, there are two **o2_event** which are useful in an O_2 application: **O2_WAIT** and **O2_DEADLOCK**. If a resource conflict occurs between two transactions, by default, the system decides which transaction should sleep until the resource is free. The system can also kill a transaction to avoid a deadlock. Before either of these events the system raises an O_2 exception that allows the application to take control and catch these events. For example, it can print a message to the user and restart at some point previously set for this purpose. To catch these exceptions in O_2C, you must include the file:

```
#include "o2_event.h"
```

At this point the function **o2_event** is available. This function allows you to define an exception handler in order to manage **O2_WAIT** and **O2_DEADLOCK** events. It takes two arguments, the event name and the exception handler address. For example, to set the handler **do_not_sleep** for the **O2_WAIT** event, you must write in O_2C:

```
#include "o2_event.h"
...
   O2_handler save_previous;
   save_previous = o2_event(O2_WAIT, (o2 void) do_not_sleep);
...
```

The O_2C function **do_not_sleep()** is defined in the current schema. After **o2_event** is called, if a resource conflict occurs, the system automatically calls the function **do_not_sleep** instead of putting the transaction to wait until the resource is free. It is a good programming practice to save the previous handler and restore it (by **o2_event**) when the handler set is not used anymore. When called, an exception handler receives one parameter which gives the event name. Consequently, **do_not_sleep** is defined as follows:

```
function body do_not_sleep(what: integer){
#include "o2_event.h"
... switch(what){
          case O2_WAIT:....
          case O2_DEADLOCK:... }
}
}
...
```

The function can then decide to resume (by return) and therefore go to sleep, or commit or abort and therefore trigger restart, or do a long jump inside the application.