t

Writing XCode*574*

This section describes how to write XCode. XCode allows you to access and define Prograph language elements in C. With XCode you can define and access persistents, class attributes, universal methods, simple class methods, class initialization methods and attribute get and set methods. You can also access (but not define) classes and instance attributes. XCode is compatible with compiled Prograph code but cannot be run by the Prograph interpreter.

A set of naming conventions is used to map between Prograph code and C code. For example, to define a universal method, myCmethod, in C, you would create a C function whose name was U_myCmethod. Conversely, to call, from C, a universal method, myPmethod, which is defined in Prograph, you would call the function U_myPmethod.

_____

NOTE: Names are case sensitive. Non-alphanumeric characters are represented by their ASCII code, preceded and followed by an underscore. For instance, U_2B_1 represents the universal method +1.

_____

NOTE: XCode cannot be run by the Prograph interpreter. The compiled? primitive can be used to prevent the interpreter from attempting to execute any references to XCode.

Function Return Values *575*

When a method completes without error, its C function should return PCF_TRUE for its integer return parameter. The only exception is a boolean method with no output root; such a method, if it completes successfully, conveys its boolean result by passing either PCF_TRUE or PCF_FALSE.

In compiled code values such as PRIMERR_ARITY, PRIMERR_VALUE and PRIMERR_TYPE should never be returned by methods.

Prograph Language Elements *575*

This section describe how to define and access each of the Prograph language elements in C. The conventions described in this section apply equally to THINK C and MPW C. These conventions will be slightly different if you are using the THINK C object extensions. These differences are discussed in a separate section.

Persistents*575*

In C, persistents are represented as global variables whose names begin with P_. The persistent, “myPersist”, defined in Prograph, is represented in C as a global variable `P_myPersist`.

C Representation:

```
P_id
```

Example:

```
  /* Assume myPersist exists in the Prograph source */

  extern C_integer *P_myPersist;

  DecUse( P_myPersist);
/* remove the existing value */
  P_myPersist = MakeC_integer( 12 );
/* set to a new value */
  ...
```

Example:

```
  C_integer *P_myPersist;
/* defines myPersist */
  ...
```

Universal Methods and Primitives [576]

In C universal methods and primitives are declared as functions of the form `U_id` which return a Nat2 value. The parameters of the function correspond to the roots and terminals of a Prograph operation.

No distinction is made between universal methods and primitives by the Prograph compiler. A universal method, “start-up”, defined in Prograph, is represented in C as a function `U_start_2D_up`. The primitive "join" is represented as `U__22_join_22_`.

You must use the SETARITY macro (described in the Arity Macros section) to set the arity before calling any method or primitive which has variable arity.

C Representation:

```
  U_id
```

Example:

```
  /* Assume string1 is "Elvis " and string2 is "lives" */
  /* string3 is uninitialized */

  C_string *string1, *string2, *string3;

  SET_ARITY( 2, 1 );
 /* "join" has variable arity */
  U__22_join_22_( string1, string2, &string3 );

  /* string3 is now "Elvis lives" */
  ...
```

Example:

```
/* The universal method swap-inputs will be callable */
/* from a compiled Prograph program */

Nat2 U_swap_2D_inputs( in1, in2, out1, out2 )
C_object *in1, *in2;
C_object **out1, **out2;

 IncUse( in1 );
 IncUse( in2 );
 *out1 = in2;
 *out2 = in1;

 return PCF_TRUE;
```

Class Identifiers "576"

Every Prograph class has an associated class identifier. Class identifiers are global variables of the C type Class and of the form C_id__. The class identifier of a Prograph class, “Person”, is represented in C as a global variable `C_Person__`.

Class identifiers are used with the supplied functions Member, Bless, New, NewN and the macro INCLASS to determine or set the class of a data item. All of these functions require the address of a class identifier.

Prograph classes and class identifiers cannot be defined in C.

C Representation:

```
C_id__
```

Example:

```
/* Use the macro INCLASS to test if */
/* anInstance is of class Person */

C_object *anInstance;
extern Class C_Person__;

if ( INCLASS( anInstance, &C_Person__ ) );

 ...
```

Default Instances "577"

Every Prograph class has a default instance which is accessible in C as a global variable of the form C_class__A_default. The default instance of a class, “Person”, defined in Prograph, is represented in C as a global variable `C_Person__A_Default`, of the type `C_Person*`.

C Representation:

```
C_class__A_default
```

Example:

```
/* Use the Duplicate function to create a new instance*/
/* of the class Person */

C_object *anInstance;


anInstance = Duplicate( C_Person__A_default, DEEP_COPY);

 ...
```

Class Attributes [577]

Class attributes are represented in C as global variables of the form C_class__A_id. Class attributes can be defined in C. The class attribute, “Person List”, of the class, “Person”, defined in Prograph, is represented in C as a global variable `C_Person__A_Person_20_List.`

C Representation:

```
C_class__A_id
```

Example:

```
/* Get class attribute "Person List" of class Person*/
C_Person *myPerson;
C_list *myList;


myList = C_Person__A_Person_20_List;
...
```

Example:

```
/* Define class attribute "First Person" of class Person */
/* This class attribute will be available in Prograph */

C_Person *C_Person__A_First_20_Person;
...
```

Instance Attributes [578]

Instance attributes can be accessed but not defined in C. To access the instance attributes of a Prograph class you define a structure in C which parallels the Prograph class definition. The C structure must begin with a two-byte type field followed by four-byte save and use fields. The remaining fields (four-bytes each) correspond in order to the instance attributes of the Prograph class. The structure name should be of the

form `C_id`. Field names are arbitrary but it is good practice to use the same names as used in the Prograph class.

Example:

```
  /* Assume a Prograph class Person with three attributes */

  /* Structure Declaration  */
  typedef struct


   Int2
  type;

   Nat4
  save;

   Nat4
  use;

   C_string
 *name;
   C_string
 *address;
   C_integer
*age;
   *C_Person;

  /* Accessing attributes */
  C_string *thisName;
  C_Person
*aPerson;

  thisName = (**aPerson).name;
  ...
```

Class Methods and Selectors[578]

In Prograph there are four types of class method: simple (plain), initialization, get and set. Class methods are represented as C functions with names of the form: C_class__M_id (simple), C_class__N_ (initialization), C_class__G_id (get), and C_class__S_id (set). In C, two other types of class methods may be defined: default get methods and default set methods which have the forms C_class__g_id and C_class__s_id respectively.

Class methods can be called directly or through what are known as method selectors. A method selector is a C function with the same name as a class method function preceded by an underscore. For example, the function name of a simple method selector has the form _C_class__M_id.

A method selector searches through the class hierarchy for the named method and then calls that method. The search starts at the class of the selector's first parameter and proceeds upward through the class hierarchy. If no method is found an error dialog will be displayed.

For example, assume an instance of class dog is passed as the first parameter to the selector _C_animal__M_talk. If the class dog has a talk method, the selector will call that method. If the class dog

does not have a talk method, the selector will examine each of dog's ancestor classes, in turn, until a talk method is found. If none of the ancestor classes has a talk method an error dialog is displayed.

Note that the class which is part of the selector name, animal in the example above, is not used in the search. (In future releases of the compiler this class name will be used to optimize method selection.) The class name should correspond to the declared class of the first parameter. For example:

```
/* class name animal is used because */
/* myAnimal is declared to be of class animal */

C_animal *myAnimal;

_C_animal__M_talk( myAnimal );
...
```

Get and Set method selectors behave slightly differently when no method is found in the class hierarchy. The search is repeated, this time for the default get or set method.

Selectors never have to be defined by the user - they are automatically created by the Prograph compiler.

Simple Class Methods *579*

Simple class methods are represented as C functions with names of the form C_class__M_id. The simple class method "birth" of the class "Person", defined in Prograph, is represented in C as a function `C_Person__M_birth`.

C Representation:

```
C_class__M_id
```

Selector:

```
_C_class__M_id
```

Example:

```
/* Call the method birth of the class Person */
/* Assume birth has one input and one output */

C_Person__M_birth( input, &output);

...
```

Example:

```
/* Assume the class Person has been defined in Prograph */
/* Define the simple method "Add Person" in C */

Nat2 C_Person__M_Add_20_Person( input )
```

```
  C_Person
*input;

   ...
 /* Do Stuff */
   return PCF_TRUE;
```

Initialization Class Methods "580"

A class initialization method is represented as a C function with a name of the form C_class__N_. Initialization methods always have one input and one output. The input value should be passed through as output. The initialization method of the class, “Person”, defined in Prograph, is represented in C as a function `C_Person__N_`.

To reproduce the function of a Prograph instance generator, first create an instance by making a deep copy of the default instance then call the selector of the initialization method.

C Representation:

```
C_class__N_
```

Selector:

```
_C_class__N_
```

Example:

```
/* Call the initialization method of class Person */
C_Person *myPerson;

C_Person__N_( New( C_Person__ ), &myPerson );
...
```

Example:

```
/* Assume the class Person has been defined in Prograph */
/* Define an initialization method for the class Person */

Nat2 C_Person__N_( input, output )

C_Person *input;
C_Person **output;

 ...
    /* Do initialization stuff */
 IncUse( input );
 *output = input;
 return PCF_TRUE;
```

Get Methods[581]

Get methods are represented as C functions with names of the form C_class__G_id. The get method of the attribute, “age”, of the class, “Person”, defined in Prograph, is represented in C as a function `C_Person__G_age`.

Get methods always have one input and two outputs. The input value should be passed through as the first output. The second output should be the value of the attribute.

C Representation:

```
C_class__G_id
```

Selector:

```
_C_class__G_id
```

Example:

```
/* Call the get method for attribute age of class Person */
C_Person *myPerson, *theInst;
C_integer *age;


C_Person__G_age( myPerson, &theInst, &age );
```

Example:

```
/* Assume a Prograph class Person with three attributes */
/* C Class declaration */

typedef struct


 Int2
type;

 Nat4
save;

 Nat4
use;

 C_string
*name;
 C_string
*address;
 C_integer
*age;
 *C_Person;
```

```
/* Get Method declaration */

Nat2 C_Person__G_age( input, output1, output2 )

C_Person *input;
C_Person **output1;
C_integer **output2;

 ...
   /* Do Get stuff */
 IncUse( input);
 IncUse( (**input).age );
 *output1 = input;
 *output2 = (**input).age;
 return PCF_TRUE;
```

Set Methods [582]

Set methods are represented as C functions with names of the form C_class__S_id. The set method of the attribute “age” of the class “Person” defined in Prograph, is represented in C as a function `C_Person__S_age`.

Set methods always have two inputs and one output. The first input value should be passed through as the output. The second output should be assigned the appropriate attribute of the input. Remember to decrement the use count of the attribute’s old value and to increment the use count of the new value.

C Representation:

```
C_class__S_id
```

Selector:

```
_C_class__S_id
```

Example:

```
/* Call the set method for attribute age of class Person */
C_Person *myPerson, *theInst;
C_integer *newage;


C_Person__S_age( myPerson, newage, &theInst);
```

Example:

```
/* Assume a Prograph class Person with three attributes */
/* C Class declaration */

typedef struct
```

```
   Int2
  type;

   Nat4
  save;

   Nat4
  use;

   C_string
 *name;
   C_string
 *address;
   C_integer
*age;
   *C_Person;

  /* Set method declaration */

  Nat2 C_Person__S_age( input1, input2, output )

  C_Person *input1;
  C_integer *input2;
  C_Person **output;

   ...
     /* Do Set stuff */
   IncUse( input2);
   IncUse( input1);

   DecUse( (**input1).age );
   (**input1).age = input2;
   *output = input1;
   return PCF_TRUE;
```

Default Get and Set Methods "583"

Default get and set methods may be defined in C only, not in Prograph. A regular get or set selector is called when the name of an attribute is preceded by a slash in a get or set operation. A default get or set selector is called when the attribute name is not preceded by a slash.

As described above default get and set selectors are also called by regular get and set selectors when no regular get or set method exists.

C Representation:

```
  C_class__g_id
  /* Default Get Method */
  C_class__s_id
 /* Default Set Method */
```

Object C Extensions "583"

XCode can be written using the THINK C object extensions. You should already have read the previous section which describes how to access and define Prograph language elements in non-object C. This section describes only the differences between object and non-object C.

NOTE: Object C declarations for the Prograph data types are given in the section on Prograph data types.

Class identifiers do not have to be declared as external in object C. Also the double underscore at the end of class identifiers is omitted.

Example:

```
  /* Use the macro INCLASS to test if */
  /* anInstance is of class Person */

  C_object *anInstance;

  if ( INCLASS( anInstance, C_Person) );

   ...
```

Instance attributes are accessed by defining a C object structure which parallels the Prograph class structure. Classes without ancestors in Prograph should inherit from the class C_object in C.

Example:

```
  /* Assume a class dog which inherits from class animal */
  /* has been defined in Prograph */

  struct C_animal :   C_object

   C_integer
*age;

  ;

  struct C_dog :   C_animal


   C_string
 *name;
   C_string
*owner

  ;

  /* Accessing attributes */
  C_integer
*theAge;
  C_string
 *theName;
```

```
  C_dog
  *aDog;

  theAge = aDog->age;
  theName = aDog->name;
  ...
```

Prograph class methods are defined in the same way as object C methods. The “C_class__” part of method names is omitted.

Example:

```
  /* Assume a Prograph class Person with three attributes */

  /* C Class declaration */

  struct C_Person :   C_object

   C_string
 *name;
   C_string
 *address;
   C_integer
*age;

   /* Method Declarations must appear in class definition */

   Nat2 M_Add_20_Person( );
  /* Class Method */

   Nat2 N_( C_Person** );
   /* Initialization Method */
   Nat2 G_age( C_Person**, C_integer** );
/* Get Method */
   Nat2 S_age( C_integer*, C_Person** );
 /* Set Method */



  /* Method Body Declarations */

  /* Simple method  - one input, no outputs */

  Nat2 C_Person:  :  M_Add_20_Person(  )

   ...

  /* Initialization method - one input, one output */

  Nat2 C_Person:  :  N_( output )

  C_Person **output;

   ...
```

```
    /* Do initialization stuff */
 IncUse( this );
 *output = this;
 return PCF_TRUE


/* Get method - one input, two outputs */

Nat2 C_Person:  :  G_age( output1, output2 )
C_Person **output1;
C_integer **output2;

 ...
   /* Do get stuff */
 IncUse( this );
 IncUse( this->age );
 *output1 = this;
 *output2 = this->age;
 return PCF_TRUE;


/* Set method  - two inputs, one output */

Nat2 C_Person:  :  S_age( input, output )
C_integer *input;
C_Person **output;

 ...
   /* Do set stuff */

 IncUse( input );
 IncUse( this );

 DecUse( this->age );
 this->age = input;
 *output = this;
 return PCF_TRUE;
```

Method selectors are called using the THINK C method calling mechanism. Note that the calling instance is automatically passed as the first parameter to a selector. Methods can be called directly using the non-object C form described in the previous section.

Example:

```
/* call the talk method of an animal */

C_animal *myAnimal;

myAnimal->M_talk( );
...
```

Building XCode in THINK C

u Create a new THINK C project.

u
Include the header file X_includes.h in each of your C source files.

u Add your source files to the THINK C project.

u Compile and build a library and save it in a file.

u Include that file in your Prograph project.

Building XCode in MPW C

u
Copy the folder XCIncludes into your MPW Interfaces folder.

u
Insert the following lines into your MPW start up script and execute them:

```
Set XCIncludes "MPWInterfaces:  XCIncludes:  "
 Export XCIncludes
```

u
Include the header file X_includes.h in each of your C source files.

u
Compile your source files with the command:

```
C mySource.c -i "XCIncludes"
```

u Include your object files and the library MPWLibrary in your Prograph project.