University of Amsterdam

Dept. of Social Science Informatics (SWI)
Roetersstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256786

A C++ interface for XPCE

Jan Wielemaker
Anjo Anjewierden
{jan,anjo}@swi.psy.uva.nl

XPCE is an object-management system and GUI toolkit designed for interactive, dynamically typed languages such as Prolog and Lisp. The C++ interface defined in this document aims at multi-lingual applications, where C++ provides raw performance and interfacing to external packages and an interactive symbolic language provides the application code and development environment.

This document describes version 0.2 of the C++ interface. The reader is supposed to be familiar with XPCE. The manual "Programming in XPCE/Prolog" [Wielemaker & Anjewierden, 1992] is a good starting point to learn about the XPCE library.

Contents

1 Introduction

XPCE has been developed as a GUI environment that can easily be connected to an arbitrary application language. Currently, XPCE is targeted first of all to Prolog. The C++ interface described in this document has been added to allow XPCE users using C++ for defining new, high-performance, XPCE classes, or XPCE encapsulation of (OS) features accessible from C/C++.

XPCE is a dynamically typed object-oriented system. (Graphical) objects may be manipulated using the generic operations new() to create an XPCE instance of an XPCE class, send() to invoke a 'procedure' method on an instance, get() for a 'function' method and free() to destroy an object.

The interface between XPCE and an application language is responsible for:

- Conversion of application-language data to XPCE data
- Conversion of XPCE data to application-language data
- Calling the XPCE generic operations
- Allowing XPCE to call functionality in the application-language

XPCE has been successfully connected to various Prolog and Lisp systems. These systems are also symbolically typed and the runtime environment can map predicate/function-names onto the implementation (i.e. the runtime environment can call a predicate/function given the name and arguments).

The C++ language has neither of these features: if a function is defined as untyped, there is no general way to find the type of the actual arguments at runtime. Also (within the language), C++ can't map the **name** of a function onto the address of the function.

The C++ **compiler** however can reason about types and the language allows for the definition of new types as well as the conversion between types. The interface described in this document exploits these features to define an interface that is equally robust and practical to use as the Prolog and Lisp interfaces.[1]

2 Motivation

Below are a number of issues to motivate an interface to C++:

- *XPCE may be used as a graphical library for C++*
  XPCE provides high-level support for dialog windows, but also for interactive graphics, text and Unix process interaction. In this respect XPCE is an alternative for other C++ connected (GUI) libraries with its own strong and weak points.

  The learning-curve for GIU libraries is generally long due to the large number of functions and options provided by such libraries. If XPCE can be connected to the language the application-programmer prefers for a particular application, s/he does not have to learn a new GUI library.

- *XPCE may be used as 'glue' in a multi-language application*
  XPCE can not only be connected to various different languages, it can also be connected to various programming environments at the same time.

[1] We also considered a C-interface. The C language both lacks runtime and compile-time type-detection. As a consequence the user has to specify the appropriate type conversions explicitly. This process is considered too error-prone.

Each of the programming environments can call XPCE behaviour without considering the language in which the behaviour is realised. For example, C++ can activate an XPCE class defined in Prolog and visa versa.

Time-critical and stable parts of the application may be modelled as C++ defined XPCE classes, while the overall control of the application may be defined in Prolog or Lisp.

- *C++ may be used to implement efficient XPCE classes*
  When communicating with Prolog or Lisp, a considerable amount of time is lost in data-transformations and control-transfer between XPCE and the Prolog or Lisp environment. With C++ this transformation is not necessary, making C++ the ideal language for defining performance critical XPCE classes.

- *Making other packages available to XPCE*
  Many systems and libraries provide an interface to C and/or C++. XPCE's C++ interface may be used as an efficient intermediate between XPCE and such an (operating-) system or library.

3 Capabilities of the interface

The interface between XPCE and C++ is fully transparent. It allows C++ code to create and manipulate XPCE objects. C++ functions can be called from XPCE code objects so that the message of a button object can call a C++ function **and** pass arguments to this function. Although, in principle, this interface suffices for defining XPCE classes in C++, a more friendly and efficient mechanism is provided for this.

The interface is built around the C++ class *PceArg*. An instance of this class represents an anonymous XPCE object. Class *PceArg* defines member functions that allow the C++ programmer to activate PCE methods on the object. Class PceArg also defines the type-conversion between raw C data-types and XPCE's dynamically typed data objects.

4 Argument-type conversion

4.1 C to XPCE

The C++ class *PceArg* defines constructors for translating raw C data-types into dynamically-typed XPCE data objects. The mapping defined by these constructors is given in table 1.

C	**XPCE**
char, short, int, long	int
char *	name object
float, double	real object

Table 1: Conversion of C data into XPCE data objects

4.2 XPCE to C

The C++ class *PceArg* defines cast operators to convert dynamically typed XPCE data objects into primitive C objects. These cast operators define the mapping given in table 2. If the requested type is char *, the value should be copied unless the converted object is **known** to be an XPCE name object.

requested C-type	**XPCE**	Remarks
int	int, number	PCE int only has 30 bits
int	real	rounded value
int	char_array	value if convertible
float, double	real	real is a float
float, double	int, number	converted value
float, double	char_array	value if convertible
char *	name	pointer to const char *
char *	string	pointer is valid as long as string lives
char *	int, number	'%d' formatted value in static area
char *	real	'%g' formatted value in static area

Table 2: Conversion of XPCE data objects to C

C++ interface functions that pass control to XPCE generally accept arguments of the type *PceArg*. The C++ type-conversion mechanism ensures that the C-primitive types are passed correctly to XPCE. The user should provide additional cast operators for application-types. Examples of using type-conversions are given in the subsequent sections.

4.3 Pointers

C-Pointers may be represented using the XPCE class *c_pointer*. On various occasions, C++ applications will have to store pointers to C++ data in XPCE, for example to pass the value of this to a *PceCall*. See section 7.1. The interface defines *PcePointer* in <pce/Pointer.h>. *PcePointer* is a derived class from *PceArg*. The value of a *PceArg* that represents a pointer is extracted using PceArg::pointer().

PcePointer::PcePointer(*void *ptr***)**

Encapsulate a C/C++ pointer into an XPCE *c_pointer* object and return a reference to the latter.

*void ** **PceArg::pointer()**

If the argument contains a *c_pointer* object, return the value of the pointer. Otherwise return NULL.

5 Invoking XPCE methods

Besides type-conversion, the C++ class *PceArg* defines the generic XPCE operations send(), get() and free() as member functions. The PCE send() and get() operations take a list of arguments. Up to 10 arguments may be specified using overloaded versions of the member functions send() and get() as defines on class *PceArg*. More arguments may be specified using sendv() and getv() that accept an array of arguments.

The following examples show how the x-coordinate of the *PceArg* object 'point' is changed and how the selection of a text-item is changed:

```
PceArg point, text_item;
...
point.send("x", 10);
text_item.send("selection", "gnat");
...
```

The member function PceArg::send() takes arguments of the type *PceArg*. The first argument of send() is interpreted as the selector, while the remaining arguments are interpreted as the arguments to the send-operation. The argument `10` is converted into an XPCE `int` datum, while the `"gnat"` argument is converted into an XPCE name object.

The member function PceArg::send() returns a value of the *enum* type `PceStatus`:

```
enum PceStatus { FAIL = 0, SUCCEED = 1 };
```

The member function PceArg::get() invokes the XPCE generic operation get(). The return value is made available as a *PceArg* again. The global object **TheDisplay** refers to what is called `@display` for the Prolog and Lisp interfaces. The following code fragment identifies XPCE:

```
void
identifyPce()
{ TheDisplay.send("inform", "I am XPCE version %s",
                  ThePce.get("version"));
}
```

Finally, the member function PceArg::free() invokes the generic XPCE operation free() to discard of the XPCE object. It is equivalent to `obj.send("free")`, except the free() may be used on objects that are already freed.

PceStatus **PceArg::send(***PceArg selector, ..., PceArg argN, ...***)**

Invoke an XPCE send operation. The first argument is used as the selector, and the remaining arguments are used as arguments to the method. Using the conversion of `char *` to *PceArg*, the selector can be written as, for example, `"append"`. If performance is critical, it is better to store the converted value in a global C++ object:

```
static PceArg PcNappend("append");

          ...,
          gnat.send(PcNappend, ...)
          ...,
```

The `PceEmacs` C++ mode contains commands to gather and replace all selectors in a file.

PceArg **PceArg::get(***PceArg selector, ..., PceArg argN, ...***)**

Similar to PceArg::send(), but implements the XPCE get operation. As the result is a *PceArg* as well, multiple get operations may be concatenated:

```
{ int screenwidth = TheDisplay.get("size").get("width");
  ...,
```

PceStatus **PceArg::free()**

Invokes →*free* on the receiver if the object exists. Succeeds if the object is not protected by 'object →*protected*'. Succeeds silently if the receiver is no object (i.e. an integer), or has already been freed.

6 Creating objects

The C++ interface defines derived classes of class *PceArg* with different constructors for creating XPCE objects. The most generic of this is the C++ class *PceObject*. The first argument (of type *PceArg*) defines the name of the XPCE class from which an instance is to be created. The remaining argument define the arguments given to the XPCE →*initialise* method. The following fragment defines a dialog windows saying hello to the world:

```
void
hello()
{ PceObject d("dialog");          // Create an instance of dialog

  d.send("append", PceObject("label", "message", "Hello World"));
  d.send("open");
}
```

Although the expression `PceObject("`⟨*classname*⟩`", ...` may be used to create objects, there are C++ header files for each of the built-in XPCE classes that define a derived class Pce⟨*CapitalisePceClassName*⟩. These classes define constructors for all possible arities (< 10) of the class' initialisation method. For example PcePoint (representing an XPCE point object) defines constructors with 0, 1, and 2 arguments as 'point →*initialise*' accepts two optional arguments.

Below is a rewritten version of the hello world program that also allows the user to delete the window.

```
#include <pce/Pce.h>
#include <pce/Label.h>
#include <pce/Button.h>
#include <pce/Message.h>
#include <pce/String.h>

void
hello2()
{ PceDialog d;                    // Create an instance of dialog

  d.send("append", PceLabel("message", PceString("Hello World")));
  d.send("append", PceButton("quit", PceMessage(d, "destroy")));
  d.send("open");
}
```

Some XPCE classes have names that cannot be represented as C++ symbols. Table 3 defines these names.

XPCE name	C++ Name	Summary
`*`	PceTimes	Multiplication expression
`-`	PceMinus	Subtraction expression
`+`	PcePlus	Addition expression
`/`	PceDivide	Division expression
`:=`	PceBinding	Name-value pair for argument-list
`<`	PceLess	Compare expressions on less-then
`=`	PceEquation	Identity between two expressions
`=<`	PceLessEqual	Compare expressions on less-or-equal
`==`	PceEqual	Test equivalence of arguments
`>`	PceGreater	Test arithmetic $>$
`>=`	PceGreaterEqual	Compare expressions on greater-or-equal
`?`	PceObtain	Invoke a get method
`\==`	PceNonEqual	Test non-equivalence of arguments

Table 3: C++ classnames for XPCE classes with a ‘symbol’ name

6.1 Global (named) objects

The C++ derived class *PceGlobal* allows access to existing named PCE objects (`@display`, `@arg1`, etc.) and allows for the definition of new named objects. The constructor arguments are:

```
PceGlobal::PceGlobal(PceArg name, [PceArg class, PceArg arg1 ...])
```

Some commonly used global objects are bound to globally defined instances of the C++ class *PceGlobal*. They are listed in table 4.

Prolog/lisp name	C++ name	Remarks
`@display`	**TheDisplay**	Represents the X-display
`@pce`	**ThePce**	Represents the environment
`@nil`	**TheNil**	Nil, nothing
`@default`	**TheDefault**	Default value
`@on`	**TheOn**	Boolean true
`@off`	**TheOff**	Boolean false
`@receiver`	**TheReceiver**	Receiver of event/method
`@event`	**TheEvent**	Current event
`@arg1` ... `@arg10`	**TheArg1** ... **TheArg10**	Forwarded arguments

Table 4: Global instances of PceGlobal

7 Callback to a C++ function

In the sections above, we have discussed how the C++ class *PceArg* and its derived classes are used to create an access objects in the XPCE world from C++. This section describes how a C++ function can be called from XPCE.

The C++ class *PceCall* is a derived class of *PceArg*. It creates an XPCE code object that, when executed, calls a C++ function. *PceCall* 'encapsulates' the C++ function in an object that may be activated from XPCE. The following example illustrates this:

```
#include <pce/Pce.h>
#include <pce/Dialog.h>
#include <pce/TextItem.h>

PceStatus
showInstances(PceArg cl)
{ PceArg c;

  if ( (c = ThePce.get("convert", cl, "class")) )
  { TheDisplay.send("inform", "Class %s has %d instances",
                    c.get("name"), c.get("no_created") -
                    c.get("no_freed"));
  }

  return SUCCEED;
}


PceStatus
pceInitApplication(int argc, char *argv[])
{ PceDialog d("Show #Instances");

  d.send("append", PceTextItem("class", "",
                               PceCall(showInstances, TheArg1)));
  return d.send("open");
}
```

The C++ class *PceFuncall* is similar to *PceCall*, but the return value of the C++ function is a *PceArg*. The resulting code object is an XPCE function object. Such an object may be used from another language (e.g. Prolog) or as part of other code objects for, for example, a button. The following example defines `@getenv` as a function to fetch a Unix environment variable.

```
#include <pce/Pce.h>
#include <stdlib.h>

PceArg
PceGetenv(PceArg name)
{ char *s = getenv(name);

  if ( s )
    return s;
  else
```

```
    return FAIL;
}


PceStatus
pceInitApplication(int argc, char *argv[])
{ PceFuncall f(PceGetenv, TheArg1);

  f.send("_name_reference", "getenv");  // name the object @getenv

  return SUCCEED;
}
```

The fragment below shows how this code is made available to Prolog:

```
1 ?- pce_load_cxx(getenv).

2 ?- get(@getenv, '_forward', 'USER', X).

X = jan
```

7.1 Callback to a C++ member function

If an application is written using C++ classes and member functions and one wishes to add XPCE behaviour to this class, it would be desirable to pass `this` and a pointer to a C++ member function to XPCE for later callback.

Such as system has been implemented in version 0.1 of this interface. This implementation relied on undefined implementation features of `g++`, where a C++ member function is basically a C function, where the first argument is a pointer to `this`. In `g++`, a pointer to a member function can be taken and casted to other types.

This implementation is not portable, and cannot even be guaranteed for future versions of `g++`, which is why we decided to drop the support for calling C++ member functions directly from XPCE.

If calling a member function cannot be avoided, use *PceCall()* to call a wrapper C++ function that calls the specific member function. The example `person.cxx` exploits this technique, which is summarised in the example below. *PcePointer* is used to pack the `this` pointer into an XPCE object of class *c_pointer*. PceArg::pointer() is used to extract the pointer value.

```
class person
{
  char *name;
public:

  setname(char *n) { name = n; }
  char *getname()  { return name; }

  void edit();
```

```
};


// The XPCE/C++ wrapper
PceStatus
setNamePerson(PceArg person, PceArg name)
{ (class person *)p = (class person *)person.pointer();

  p->setname(name);

  return SUCCEED;
}


// And the simple edit control
person::edit()
{ TextItem ti("name", name,
              PceCall(setNamePerson, PcePointer(this), TheArg1));

  ti.send("open");
}
```

8 Defining an XPCE class in C++

The previous sections illustrate how the XPCE library may be used from a C++ application. Although C++ functionality can be made available to users of other languages using *PceCall* and friends, this is often not the desirable way (c.f. the clumsy way to define and call the `@getenv` function on page 8).

A more natural way to make C++ functionality available to users of other languages is by defining a XPCE class from C++. The definition of a XPCE class consists of C++ functions implementing the XPCE methods, a C++ function to build the XPCE class object and finally a call to XPCE to let XPCE know the class can be built by calling this function.

8.1 An XPCE-method function

A C++ function that implements a method of an XPCE class returns a `PceStatus` for a send-method and an *PceArg* for a get-method. The first argument is an instance of the C++ class *PceReceiver*, a subclass of *PceArg*. All remaining arguments are instances of *PceArg*. *PceReceiver* defines member-functions for fast access to instance-variables:

PceArg **PceReceiver::fetch(***PceVariable var***)**

Fetches the value of the indicated instance-variable. *var* is normally stored in a global variable which is initialised by PceClass::defvar(). This function realises 'object ←*slot*'.

PceStatus **PceReceiver::store(***PceReceiver var, PceArg value***)**

Stores the value of an instance variable. If this function fails, it indicates that *value* could not be converted to the type of the instance variable. This function realised 'object →*slot*'.

PceStatus **PceReceiver::sendSuper(***PceArg ...***)**

Invokes a method using the definition of the super-class of the class on which the currently invoked method is defined. See 'object →*send_super*'.

PceArg **PceReceiver::getSuper(***PceArg ...***)**

As PceReceiver::sendSuper(), but for get-methods. See 'object ←*get_super*'.

The example below is a typical example, combining various of the above features:

```
PceStatus
initialiseApp(PceReceiver a, PceArg client)
{ a.sendSuper("initialise", "I am a toplevel window");
  a.store(VarClient, client);
  a.send("append", PceDialog());

  return TRUE;
}
```

8.2 Declaring the XPCE class

The C++ class *PceClass* is a derivate of *PceArg* for defining new XPCE classes. It defines the following member-functions:

*PceVariable** **PceClass::defvar(***name, group, summary, type, access, initial)***)**

All arguments are of the type *PceArg*. See XPCE manual on class variable and 'class →*instance_variable*' for details.

PceStatus **PceClass::defsendmethod(***name, group, summary, function, type ...)***)**

All arguments are of the type *PceArg*. function is a pointer to a function that returns a `PceStatus`, which first argument is a *PceReceiver* and which has the same number of *PceArg* arguments as there are 'types' provided (maximum 10).

PceStatus **PceClass::defgetmethod(***name, group, summary, rtype, function, type ...)***)**

Similar to PceClass::defsendmethod(). The argument *rtype* is a type defining the return-type. The return-type of the C++ function is *PceArg*.

Class *PceClass* defines two constructors: with a single argument it looks up an existing class object of the specified name. With 4 arguments it declares a C++ definition of a class. The arguments are: name of the class, name of the super-class, summary description and pointer to a function to build the class. This function returns *PceArg* and takes an (virgin) *PceClass* as argument. See the example below.

8.3 Example: Binary Tree

The example below exploits all the mechanisms described above to define an XPCE class *b_node*: a node in a binary tree.

```
#include <pce/Pce.h>

// References to XPCE instance variables of the class.

static PceVariable *bnode_left;
static PceVariable *bnode_right;
static PceVariable *bnode_key;
static PceVariable *bnode_value;

// Method to initialise the instance (like a C++ constructor)

static PceStatus
initialiseBNode(PceReceiver n, PceArg key, PceArg value)
{ n.store(bnode_key, key);
  n.store(bnode_value, value);

  return SUCCEED;
}


// Send method to insert a new key/value pair into the tree

PceStatus
insertBNode(PceReceiver n, PceArg key, PceArg value)
{ int c = strcmp(n.fetch(bnode_key), key);

  if ( c == 0 )                         /* replace value */
    return n.store(bnode_value, value);

  PceArg n2 = n.fetch(c > 0 ? bnode_right : bnode_left);
  if ( n2 != TheNil )
    return insertBNode(n2, key, value);

  return n.store(c > 0 ?  bnode_right : bnode_left,
                 PceObject("b_node", key, value));
}


// Method to find a node from a key.

PceArg
getNodeBNode(PceReceiver n, PceArg key)
{ int c = strcmp(n.fetch(bnode_key), key);

  if ( c == 0 )
    return n;
```

```
  PceArg n2 = n.fetch(c > 0 ? bnode_right : bnode_left);

  if ( n2 != TheNil )
    return getNodeBNode(n2, key);

  return FAIL;
}


// C++ function to build the XPCE class definition

PceStatus
makeClassBNode(PceClass cl)
{ bnode_left =
    cl.defvar("left", "tree", "Node to my left",
              "b_node*", "get", TheNil);
  bnode_right =
    cl.defvar("right", "tree", "Node to my right",
              "b_node*", "get", TheNil);
  bnode_key =
    cl.defvar("key", "table", "Key-name of the node",
              "name", "get", TheNil);
  bnode_value =
    cl.defvar("value", "table", "Value of the node",
              "any", "get", TheNil);

  cl.defsendmethod("initialise", "oms", "Create from key and value",
                   initialiseBNode, "key=name", "value=any");
  cl.defsendmethod("insert", "edit", "Add entry to the table",
                   insertBNode, "key=name", "value=any");

  cl.defgetmethod("node", "lookup", "Lookup node from key",
                  "b_node", getNodeBNode, "key=name");

  return SUCCEED;
}


// let a C++ global constructor declare the class

PceClass ClassBNode("b_node", "object", "Node of a binary tree",
                    makeClassBNode);
```

9 Lifetime of C++ objects and XPCE objects

All C++ classes involved in defining the interface define a single member: the `void *` 'self'. They represent no more than a **reference** to the XPCE object. Creation of a C++ *PceArg* instance does not automatically imply creation of an XPCE object and deleting the *PceArg* object has no consequences for the XPCE object.

Using constructors and destructors of PceArg objects can possibly be used to help the incremental garbage collector of XPCE. This is still subject of study.

10 Linking C++ code to XPCE

10.1 Incremental linking to Prolog

To link an piece of C-code to XPCE/Prolog, the XPCE/C++ is best compiled into a shared object (Unix) or a Dynamic Load Library (Windows), which can then be loaded into the running XPCE/Prolog system using Prolog's primitives for loading such files. Normally, no function needs to be called in the library.

The required system features are supported in Unix systems using the ELF binary format (Solaris 2.x, Irix 5.x, Linux) and Win32 (Windows 95/NT). Except for platform specific problems, the following sequence should suffice for creating a compliant `.so` file from an XPCE C++ class using `g++`.

```
% g++ -c -fpic -I$PCEHOME/include myfile.C
% g++ -o myfile.so -shared myfile.o
```

On Windows systems, the C++ files must be compiled and linked with the import library `xpce.lib` to a `.DLL` file. The Windows binary is compiled with `MSVC 4.2`.

On modern operating systems, C++ global constructors will automatically be executed when the shared object is loaded into the running application.

The XPCE/Prolog library library(`pce_loadcxx`) defines the predicate `pce_load_cxx/1` to load a shared object or DLL file:

pce_load_cxx(+*FileSpec***)**

Load the shared object holding an XPCE class. This predicate is provided for both SWI-Prolog and Quintus Prolog. XPCE/C++ code can also be part of normally Prolog foreign files and be loaded using the Prolog native foreign file loading predicates.

The following is a typical example of the header of a Prolog file using a C++ defined XPCE class.

```
:- module(mymodule, []).
:- use_module(library(pce)).
:- require([ pce_load_cxx/1
           ]).

:- initialization
   pce_load_cxx(foreign(mycxxcode)).
```

```
...
```

10.2 C++ stand-alone applications

Programs can be linked to a C++/XPCE stand-alone application by linking the `.o` files to the `-lXPCEmain` and `-lXPCE` libraries. The `-lXPCEmain` library contains a stub main() function that initialises XPCE, and calls pceInitialiseApplication(*int argc, char *argv[]*), passing the main arguments *argc* and *argv*. Finally it enters a loop waiting for user-events and dispatching them. This loop terminates if there are no more toplevel *frame* objects on the current display. The application can be terminated explicitly by invoking '`@pce` →*die*', or calling exit(). Below is the (Unix) sourcecode of `main.cxx`, the only member of `-lXPCEmain`. This file is available as ⟨*pcehome*⟩`/src/itf/pcemain.cxx`

```
#include <pce/Pce.h>
#include <pce/Chain.h>

static PceArg PcNframes("frames");
static PceArg PcNkind("kind");
static PceArg PcNtoplevel("toplevel");

extern "C" {
int     pceInitialise(int handles, char *home, int argc, char **argv);
int     pceDispatch(int fd, int timeout);
void    Cprintf(const char *fmt, ...);
}

int
main(int argc, char* argv[])
{ int frames = TRUE;

  if ( !pceInitialise(0, (char *)0, argc, argv) )
  { Cprintf("Sorry, failed to initialise XPCE\n");
    exit(1);
  }

  if ( !pceInitApplication(argc, argv) )
  { Cprintf("Failed to run pceInitApplication()\n");
    exit(1);
  }

  while(frames)
  { PceCell cell;

    pceDispatch(0, 1000);

    for(frames = FALSE, cell = AsChain(TheDisplay.get(PcNframes)).head();
```

```
          cell;
          ++cell)
      { if ( cell.value().get(PcNkind) == PcNtoplevel )
        { frames = TRUE;
          break;
        }
      }
    }

    exit(0);
  }
```

An XPCE/C++ stand-alone executable needs to be able to find the XPCE home directory for the system-required bitmaps, the resources and the PostScript header file. The path to the home directory is provided as second argument to pceInitialise(), or read from the environment variable `PCEHOME`.

10.2.1 Stand-alone applications in Win32

The above also works on Windows platforms. `main.cxx` contains a definition for WinMain() that parses the commandline arguments and calls main() as described above. If the stand-alone Windows applications wants to write or read to/from standard I/O, the system will automatically allocate a console using Windows AllocConsole() API.

11 Goodies

This section defines some member functions and types for speeding up some operations and/or providing shorthands.

11.1 Arithmetic on class PceArg

Class *PceArg* defines integer arithmetic by overloading the operators +, -, *, /, ++, –, +=, -=, *= and /=. These functions yield an error if one of the operands cannot be translated into an integer. The following example defines an XPCE callable function that draws an array of boxes.

```
#include <pce/Pce.h>
#include <pce/Box.h>
#include <pce/Point.h>
#include <pce/Picture.h>

PceStatus
drawBoxes(PceArg dev, PceArg x, PceArg y, PceArg w, PceArg h, PceArg n)
{ while(n-- > 0)
  { dev.send("display", PceBox(w, h), PcePoint(x, y));
    x += w;
  }

  return SUCCEED;
```

```
}

PceStatus
pceInitApplication(int argc, char *argv[])
{ PcePicture p;

  drawBoxes(p, 10, 10, 20, 50, 20);
  p.send("open");

  return SUCCEED;
}
```

11.2 Enumerating cells of a PCE chain

The following functions allow for enumerating the elements of a list similiar to 'chain →*for_all*' and friends.

PceCell **PceChain::head()**
: Returns a reference to the first cell of the chain. The C++ class *PceCell* is **no** derivate of *PceArg*.

PceArg **PceCell::value()**
: Returns the value of the cell.

PceArg **PceCell::operator ++()**
: Prefix ++. Modifies the *PceCell* object to point to the next cell in the XPCE chain.

PceArg **PceCell::operator ++(*int*)**
: Postfix ++. Modifies the *PceCell* object to point to the next cell in the XPCE chain, returns a new *PceCell* object pointing to the old cell. Use `++cell` if you can.

The following example lists the (sub-) directories and files in the argument directories. It uses the XPCE 'chain →*for_all*' for the directories, and C++ Cell methods for the files.

```
#include <stdlib.h>
#include <pce/Pce.h>
#include <pce/Chain.h>
#include <pce/Directory.h>
#include <pce/Message.h>
#include <iostream.h>

PceStatus
pceInitApplication(int argc, char *argv[])
{ int i;

  for(i=1; i < argc; i++)
  { PceDirectory d(argv[i]);

    PceChain dirs = AsChain(d.get("directories"));
```

```
    cout << "sub directories of " << argv[i] << endl;
    dirs.send("for_all", PceMessage(ThePce, "write_ln", TheArg1));

    PceChain files = AsChain(d.get("files"));
    cout << "files in " << argv[i] << endl;
    PceCell cell;

    for( cell = files.head(); cell; )
      cout << (char *)(++cell).value() << endl;
  }

  ThePce.send("die");
}
```

While *PceCell* references contain valid references to cells of XPCE chains, the contents of the chain may **not** be manipulated.

11.3 Debugging (pretty print)

char * **PceArg::pp()**

The member function PceArg::pp() returns a description of the argument in the same style as the the XPCE debugger. The storage is provided by a ring of 16 buffers. This implies that upto 16 arguments may use pp() without overwriting each others output. If information has to be kept it must be copied.

12 Status

The status of the XPCE/C++ interface should be regarded alpha. The basic concept will probably not change, neither will the mechanisms for creating objects, sending messages, etc. Various things are not yet provided for:

- Incremental garbage collector interface
  This can probably be integrated into the constructors and destructors of the various C++ classes. As omitting this just leaves garbage around for a little too long there is no reason to hurry.

- More 'goodies'
  It may prove useful to define more 'goodies', both to provide shorthands for commonly used operations and to provide low-level fast access to the XPCE virtual machinery.

References

[Wielemaker & Anjewierden, 1992] J. Wielemaker and A. Anjewierden. *Programming in PCE/Prolog*. SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1992. E-mail: jan@swi.psy.uva.nl.

Index