ObjectStore® PSE Pro™

PSE Pro for C++ Tutorial

Release 6.3

PROGRESS SOFTWARE | Real Time Division

PSE Pro for C++ Tutorial, Release 6.3, October 2005

September 2005

Contents

Preface 5

Chapter 1 Benefits of PSE Pro for C++ 9

Chapter 2 Description of the Data Model for the Tutorial Example . . 11

Required PSE Pro Header File 15

Chapter 3 Writing an Application to Use PSE Pro 21

Establishing Fault Handlers 21

Initializing PSE Pro. 21

Creating and Opening Databases 22

Using Database Roots as Entry Points 22

Saving Databases 23

Closing Databases 23

Chapter 4 Building Applications That Use PSE Pro 25

Creating the Schema Source File. 25

Generating the Schema 26

Running the **pssg** Utility. 26

Regenerating the Schema Object File 26

Default Compiler Switches 26

Schema Generation in the VC++ IDE 27

Compiling the Code 27

Linking the Components 27

Typical Link Errors Related to Schema Generation 28

Static Builds. 28

Schema Generation and Static builds 29

Using MSDEV IDE 29

Index 31

Preface

Purpose

PSE Pro for C++ Tutorial provides an introduction to writing and building PSE Pro applications.

Audience

This book is for programmers responsible for writing database applications that use PSE Pro for C++. No previous knowledge of PSE Pro for C++ or the C++ interface to ObjectStore is required. It is assumed that you are experienced with Visual C++.

Scope

This book supports Release 6.3 of PSE Pro for C++.

How This Book Is Organized

Chapter 1, Benefits of PSE Pro for C++, on page 9 describes the advantages of using PSE Pro to maintain persistent data.

Chapter 2, Description of the Data Model for the Tutorial Example, on page 11 provides sample code for a simple application.

Chapter 3, Writing an Application to Use PSE Pro, on page 21 uses the sample application described in Chapter 2 to illustrate the steps you must follow to write a PSE Pro application.

Chapter 4, Building Applications That Use PSE Pro, on page 25 uses the sample application described in Chapter 2 to illustrate the steps you must follow to build a PSE Pro application.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code, such as user input.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.
`[ ]`	Brackets enclose optional arguments.

Convention	*Meaning*
`{ }*` or `{ }+`	When braces are followed by an asterisk (*), the items enclosed by the braces can be repeated `0` or more times; if followed by a plus sign (+), one or more times.
`{ a | b | c }`	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
`...`	Three consecutive periods can indicate either that material not relevant to the example has been omitted or that the previous item can be repeated.

Progress Software Real Time Division on the World Wide Web

The Progress Software Real Time Division Web site (`www.progress.com/realtime`) provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

To obtain information about purchasing technical support, contact your local sales office listed at `www.progress.com/realtime/techsupport/contact`, or in North America call 1-781-280-4833. When you purchase technical support, the following services are available to you:

- You can send questions to `realtime-support@progress.com`. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at `www.progress.com/realtime/techsupport/contact`.
- You can file a report or question with Technical Support by going to `www.progress.com/realtime/techsupport/techsupport_direct`.
- You can access the Technical Support Web site, which includes
  - A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
  - Solution Knowledge Base that you can browse and query.
  - Online documentation for all products.
  - White papers and short articles about using Real Time Division products.
  - Sample code and examples.
  - The latest versions of products, service packs, and publicly available patches that you can download.
  - Access to a support matrix that lists platform configurations supported by this release.
  - Support policies.
  - Local phone numbers and hours when support personnel can be reached.

Education Services

To learn about standard course offerings and custom workshops, use the Real Time Division education services site (`www.progress.com/realtime/services`).

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. If you are outside North America, refer to the Technical Support Web site. For information on current course offerings or pricing, send e-mail to `classes@progress.com`.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the Technical Support Web site at `www.progress.com/realtime/techsupport/documentation`. The site provides documentation for the most recent release and the previous supported release. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

Real Time Division product development welcomes your comments about its documentation. Send any product feedback to `realtime-support@progress.com`. To expedite your documentation feedback, begin the subject with `Doc:`. For example:

```
Subject: Doc: Incorrect message on page 76 of reference manual
```

Chapter 1
Benefits of PSE Pro for C++

PSE Pro for C++ is a lightweight persistent storage engine that allows you to store and retrieve objects in their native C++ format. Without PSE Pro, you could store C++ data in flat files or relational databases, use MFC serialization, and write code to perform I/O and translation to and from object format. But such code is complex, error prone, and difficult to maintain. PSE Pro eliminates these problems in favor of

- A simple to use and easy to learn API
- A high-performance virtual memory mapping architecture

The memory mapping architecture guarantees optimal access speeds by using regular virtual memory pointers instead of *soft* pointers. With soft pointer schemes, every pointer dereference carries the overhead of a check to determine if the object pointed to has been transferred to program memory. With PSE Pro, no checking is required for objects that have already been transferred. Pointers to transferred objects are virtual memory pointers processed at hardware speeds.

You do not need to change the structure of your application to take advantage of these features. PSE Pro comes with an AppWizard that generates standard MFC windows applications. The only difference is that the application's document class is derived from a PSE Pro-specific document class that uses PSE Pro for persistence rather than object serialization.

Chapter 2
Description of the Data Model for the Tutorial Example

This chapter describes the data model for the example that is discussed in this tutorial.

This tutorial shows you how to create a simple PSE Pro application for storing and looking up telephone numbers. The application records names and numbers in a binary tree, which is stored in a database in its native format. Since the tree is allocated in database memory, all updates to the tree are automatically stored in the database (an operating system file), for use by a future run of the application.

The source code for the telephone numbers application is in *`PSE_Pro_install_dir`*`\examples\phone`.

This chapter presents the following information:

Class Template for Storing Objects in a Persistent Binary Tree 12
Modifications Required for Persistence 14
Description of Persistent New 15
Required PSE Pro Header File
Data Class for Storing Phone Numbers in a Persistent Binary Tree 16
`phone.cpp` 17

Class Template for Storing Objects in a Persistent Binary Tree

Here are the header file and source code file for a binary tree class template similar to one you might already use, although this class template is very basic. These files do not yet include any PSE Pro APIs.

Header file

```
/*** BinaryTree.h ***/

template <class T>
class BinaryTree
{
  BinaryTree<T> *left;
  BinaryTree<T> *right;
  T &data;

public:
  BinaryTree(T &d);
  ~BinaryTree();

  void add(T &d);
  T *lookup(T &d);
  void print() const;
  void dump() const;
};
```

Source code file

```
/*** BinaryTree.cpp  ***/

#include <stdlib.h>
#include <iostream>
#include <windows.h>
#include "BinaryTree.h"

// Create a binary tree whose root node contains the data item d
template <class T>
BinaryTree<T>::BinaryTree(T &d):
  data(d), left(0), right(0) {}

// destructor
template <class T>
BinaryTree<T>::~BinaryTree()
{
  delete this->left;
  delete this->right;
  delete &data;
}

// add an item. Class T must define operator <()
template <class T>
void BinaryTree<T>::add(T &d)
{
  if (d < data)
    if (left)
      left->add(d);
    else
      left = new BinaryTree<T>(d);
  else
    if (right)
      right->add(d);
    else
```

```
      right = new(   ) BinaryTree<T>(d);
}

// Return a pointer to an item in the tree that is ==
// to the specified item. Return 0 if not found.
// Class T must define operator ==() and operator <()
template <class T>
T *BinaryTree<T>::lookup(T &d)
{
  if (d == data) return &data;

  if (d < data)
    if (left)
      return left->lookup(d);
    else
      return 0;
  else
    if (right)
      return right->lookup(d);
    else
      return 0;
}

// Print a textual representation of the data item
// for the root node.
// Class T must have an associated operator <<()
template <class T>
void BinaryTree<T>::print() const
{
  cout << data << endl;
}

// Run print() on every node of the tree
template <class T>
void BinaryTree<T>::dump() const
{
  if (left)
    left->dump();

  print();

  if (right)
    right->dump();
}
```

Modifications Required for Persistence

To prepare a class so that its instances can be stored in a PSE Pro database, you might have to modify some member function implementations. If a member function allocates any new objects, you must decide whether these new objects should be allocated in a database. If so, you must change the allocation to use the `new` operator defined by the PSE Pro API.

In the phone application, the only member function that performs any allocation is `BinaryTree::add()`. It allocates a new node (that is, a subtree) to hold the data item to be added. Since the tree is going to be persistent, the new nodes must be persistent as well. To make the new nodes persistent, change the calls to `new` in `add()` to use the PSE Pro overloading of `operator new()`. None of the other member functions needs to be changed.

Here is the new implementation of `BinaryTree::add()`:

```
// add an item. Class T must define operator <()
template <class T>
void BinaryTree<T>::add(T &d)
{
  if (d < data)
    if (left)
      left->add(d);
    else
      left = new(
          os_database::of(this),
          os_ts< BinaryTree<T> >::get()
        ) BinaryTree<T>(d);
  else
    if (right)
      right->add(d);
    else
      right = new(
          os_database::of(this),
          os_ts< BinaryTree<T> >::get()
        ) BinaryTree<T>(d);
}
```

No other function implementations and no class definitions need modification.

Description of Persistent New

The heart of the PSE Pro API is a special overloading of `::operator new()`, known as *persistent* `new`. Allocating data with persistent `new` is a lot like allocating data with regular `new`. The difference is that data allocated with regular `new` is *transient*. That is, it disappears after the allocating application terminates. Data allocated with persistent `new` resides on stable storage, in a database, until explicitly deleted. You can use persistent `new` to allocate persistent instances of built in types like `int` or `char*`, as well as classes like `BinaryTree`.

Persistent `new` takes two arguments:

- An `os_database*`, which indicates the database in which to allocate the new object
- An `os_typespec*`, which specifies the type of the new object

In `BinaryTree::add()`, the database is specified by a call to `os_database::of()`, which returns the database that contains `this`. This call ensures that the new node is stored in the same database as the tree it is being added to. It also ensures that, if the tree is stored on the transient heap, so is the new node (in that case, `database::of()` returns the *transient database*).

The second argument is a call to `os_ts<BinaryTree>::get()`. It returns an `os_typespec*` for the class `BinaryTree`. You can retrieve `os_typespec*`s for built in types with static members of `os_typespec` such as `os_typespec::get_int()`. Use `os_typespec::get_pointer()` for pointer types.

Required PSE Pro Header File

Since `BinaryTree::add()` now uses the PSE Pro API, `BinaryTree.cpp` must include the header file `<os_pse/ostore.hh>`:

```
#include <os_pse/ostore.hh>
```

Data Class for Storing Phone Numbers in a Persistent Binary Tree

`BinaryTree` is a class template so that you can use instances of any class as the data items that you store in the tree. The telephone numbers application stores names and numbers. Consequently, it uses a class with a member for the name and a member for the number:

Header file

```
/*** Data.h ***/

class Data
{
public:
  char *name;
  char *number;

  Data(const char *na, char *nu);

  friend ostream &operator <<(
    ostream& os,
    Data const &d
  );

  // BinaryTree uses op < and op ==
  // to look up and add data items.
  // Lookups are keyed by name, so these operators
  // use strncmp() on Data::name
  int operator <(Data const &d) {
    return strncmp(name, d.name, strlen(name)) < 0; }

  int operator ==(Data const &d) {
    return strncmp(name, d.name, strlen(name)) == 0; }
};
```

Source code file

```
/*** Data.cpp ***/

#include <stdlib.h>
#include <iostream>
#include <os_pse/ostore.hh> // PSE Pro header file
#include <windows.h>
#include "Data.h"

Data::Data(const char *na, char *nu)
// Sets the name and number to persistent copies of
// na and nu
{
  int len;
  if (na) { // allocate persistent char array for name
    len = strlen(na) + 1;
    name = new(
        os_database::of(this),
        os_typespec::get_char(),
        len
      ) char[len];
    strcpy(name, na); // initialize the persistent array
  }

  if (nu) { // allocate persistent char array for number
    len = strlen(nu) + 1;
    number = new(
        os_database::of(this),
```

```
        os_typespec::get_char(),
        len
      ) char[len];
    strcpy(number, nu);   // initialize the persistent array
  }
}

// Insert the name and number strings into the stream
ostream &operator <<(ostream& os, Data const &d)
{
  return os << d.name << endl << d.number << endl;
}
```

The character arrays that hold the names and numbers should be allocated in persistent memory, so the `Data` constructor defined in `Data.cpp` uses persistent `new`. This overloading of persistent `new` is for allocating arrays. It takes a third argument, which is the length of the array.

phone.cpp

Here is the main routine of the phone application. It

- Establishes fault handlers
- Initializes PSE Pro
- Opens a database or creates one first if necessary
- Retrieves the binary tree or creates one first if necessary
- Executes a loop that allows you to do one of the following each time through the loop:
  - Look up a number
  - Add a name and number
  - Save the database
  - Revert to the last saved database
  - Dump the contents of the tree
  - Exit from the application

```
/*** phone.cpp ***/

#include <stdlib.h>
#include <iostream>
#include <os_pse/ostore.hh> // PSE Pro header file
#include <windows.h>
#include "BinaryTree.h"
#include "Data.h"

#define MAX_CHAR 500  // max input characters

int main()

{
  // The top of the stack of every thread of every program
  // must be wrapped in a fault handler.
  // This macro begins the handler's scope.
  OS_PSE_ESTABLISH_FAULT_HANDLER
```

```
  cout << "Starting..." << endl;

  // Initialize PSE Pro
  cout << "Initializing..." << endl;
  objectstore::initialize();

  cout << "Creating or Opening Database..." << endl;
  // Open the database named "phone.db". If one does
  // not exist, create and open one.
  os_database *db = os_database::open(
    "phone.db", 0, 0666);

  // Find the tree. If it does not exist, create it.

  // Find the database root that points to the tree
  os_database_root *db_root =
    db->find_root("phone_root");

  if (!db_root) {
  // If no such database root, create one

    cout << "Creating Tree..." << endl;

    db_root = db->create_root("phone_root");

    // Create a data item to be associated with
    // the root node of the new tree

    Data *new_data = new(db,
      os_ts<Data>::get())
        Data("Emergency", "911");

    // Create the tree
    BinaryTree<Data> *the_tree =
      new(
        db,
        os_ts< BinaryTree<Data> >::get()
      ) BinaryTree<Data>(*new_data);

    // Point the database root at the tree
    db_root->set_value(
      the_tree
    );

    // Save the updates to the database
    db->save();

  }

  // Retrieve the tree
  BinaryTree<Data> *the_tree = (BinaryTree<Data>*) (
    db_root->get_value()
  );
  char next_action = 0; // input from user
  char na[MAX_CHAR+1]; // used by action Add
  char nu[MAX_CHAR+1];  // used by action Add
  Data *data_item = 0; // used by actions Add and Lookup
  Data *answer_data; // used by action Lookup

while (next_action != 'q') {

  cout << "\nNext action?" << endl <<
    "(l for Look up a number," << endl <<
    "a for Add new data items to the tree, " << endl <<
    "s for Save database, " << endl <<
    "r for Revert to last save, " << endl <<
```

```
  "d for Dump tree, " << endl <<
  "q for Quit)" << endl << "==> ";

cin >> next_action;

switch(next_action)   {

case 'l':
  // Look up a number, given a name
  cout << "Name?" << endl << "==> ";
  cin.ignore(1);
  cin.getline(na, MAX_CHAR);
  data_item = new Data(na, 0);
  answer_data = the_tree->lookup(*data_item);
  delete data_item;
  if(answer_data)
    cout << endl << *answer_data;
  else
    cout << "Not found." << "\n\n";
  break;

case 'a':
  // Add a name and number
  cout << "Name?"  << endl << "==> ";
  cin.ignore(1);
  cin.getline(na, MAX_CHAR);
  cout << "Number?"  << endl << "==> ";
  cin.getline(nu, MAX_CHAR);
  cout << "Adding new persistent data..." << "\n\n";
  data_item = new(db, os_ts<Data>::get())
    Data(na, nu);
  the_tree->add(*data_item);
  break;

case 'd':
  // Dump tree
  cout << endl;
  the_tree->dump();
  cout << endl;
  break;

case 's':
  // Save database
  cout << "Saving database..." << "\n\n";
  db->save();
  break;

case 'r':
  // Close db, open it again, and retrieve tree
  cout << "Reverting to last save..." << "\n\n";
  db->close();
  db = os_database::open("test.db");
  the_tree = (BinaryTree<Data>*) (
      db->find_root("test_root")->get_value()
    );
  break;

default:
  break;
}

}
// Close database
```

```
  db->close();
  cout << "Done." << endl;
  // This macro ends the fault handler's scope
  OS_PSE_END_FAULT_HANDLER
  return 0;
}
```

Chapter 3
Writing an Application to Use PSE Pro

This chapter describes the additions to your program that are required by PSE Pro. These include

Establishing Fault Handlers 21
Initializing PSE Pro 21
Creating and Opening Databases 22
Using Database Roots as Entry Points 22
Saving Databases 23
Closing Databases 23

Establishing Fault Handlers

PSE Pro must handle all memory access violations, because some are actually references to persistent memory in a database. On Windows, there is no function for registering a signal handler for such access violations that also works when you are debugging a PSE Pro application. (The `SetUnhandledExceptionFilter` function does not work when you use the Visual C++ debugger.)

Therefore, every PSE Pro application must put a handler for access violations at the top of every stack in the program. This normally means putting a handler in a program's `main` or `WinMain` function and, if the program uses multiple threads, putting a handler in the first function of each new thread.

PSE Pro provides two macros for establishing a fault handler:

- `OS_PSE_ESTABLISH_FAULT_HANDLER`
- `OS_PSE_END_FAULT_HANDLER`

Initializing PSE Pro

You must call the static function `objectstore::initialize()` before making any calls to the PSE Pro API. The function takes no arguments.

Creating and Opening Databases

To gain access to a database, you must first *open* it. The `os_database::open()` function opens a database with a specified name. If a database with that name does not exist, the function can create one, if you want, and then open it. The function returns an `os_database*` that represents the opened database. Here is the call to `open()` in `phone.cpp`:

```
os_database *db = os_database::open(
  "test.db", 0, 0666);
```

The first argument is the name of the database. This can be an absolute or relative pathname.

The second argument indicates whether to open the database for read-only (1 for true and 0 for false). Multiple applications can have a database opened for read-only simultaneously. Only one application at a time can have a database opened for update.

The third argument indicates that the function should create a new database with the specified name if one does not already exist. The value indicates a protection mode. Pass `0` if you do not want to create a database in the event there is no database with the specified name. The function throws `os_err_database_not_found` if there is no such database and the third argument is `0`.

Using Database Roots as Entry Points

A *database root* provides a way to give an object a persistent name. This allows the object to serve as an *entry point* into a database. When an object has a persistent name, any process can look it up by that name to retrieve it. After you have retrieved one object, you can retrieve any object related to it by using navigation. That is, you can retrieve additional objects by following data member pointers. Each database needs only one or a small number of entry points. Looking up an entry point is significantly slower than navigating to an object.

A database root, an instance of `os_database_root`, associates an object with a string. You specify the name of the object when you create a root, with `os_database::create_root()`. You set and get a pointer to the entry-point object with `os_database_root::set_value()` and `os_database_root::get_value()`.

The phone application uses database roots as follows:

- It calls `os_database::find_root()` to look up the root named `"phone_root"`. The function returns `0` if the root does not exist.

  ```
  os_database_root *db_root = db->find_root("phone_root");
  ```

- If the root does not exist, the application creates one and sets the value of the root to point to a new `BinaryTree` whose root is a new instance of `Data`.

  ```
  if (!db_root) {
  // If no such database root exists, create one.
  ```

```
cout << "Creating Tree..." << endl;

db_root = db->create_root("phone_root");

// Create a data item to be associated with
// the root node of the new tree.

Data *new_data = new(db,
  os_ts<Data>::get())
    Data("Emergency", "911");

// Create the tree.
BinaryTree<Data> *the_tree =
  new(
    db,
    os_ts< BinaryTree<Data> >::get()
  ) BinaryTree<Data>(*new_data);

// Point the database root at the tree.
db_root->set_value(
  the_tree
);
```

- 'Finally, the application retrieves the entry-point object pointed to by the root's value.

```
BinaryTree<Data> *the_tree = (BinaryTree<Data>*) (
  db_root->get_value()
);
```

Saving Databases

To make your changes to a database permanent, use `os_database::save()`. After the save operation is complete, all the database's data resides safely on disk, including your latest changes. Subsequently. your data is recoverable as of the time of the last database save operation.

Closing Databases

You close a database by calling `os_database::close()`. This makes the database's data inaccessible; any pointers to objects in the database are rendered invalid. Closing a database also deletes the instance of `os_database` pointed to by `this`. Of course, the database itself is not deleted.

Chapter 4
Building Applications That Use PSE Pro

The steps for building an application that uses PSE Pro are

Creating the Schema Source File 25
Generating the Schema 26
Compiling the Code 27
Linking the Components 27

Creating the Schema Source File

You must link every PSE Pro application or library with an object file that contains information about the classes of persistent objects used by the application. You generate this object file from the schema source file, a simple source file you create as follows:

- Include the PSE Pro header file `<os_pse/ostore.hh>.`
- Include the definition of each class the application might allocate, as well as the definition of each class in each database the application might open.
- Call the `OS_MARK_SCHEMA_TYPE()` macro for each included class, supplying the class name as argument. Do not put quotation marks around the class name. Typedef names are not allowed. Enter the argument *without white space.*The macro cannot span more than *one* (1) line in the schema source file.
- By convention, the schema source files used in the PSE Pro example applications use the file extension `.scm`.

You must mark each instantiation of any class template you use. Since `BinaryTree` is a class template, you must mark `BinaryTree<Data>`. For the phone application, mark `BinaryTree<Data>` and `Data`, as follows:

```
/*** schema.scm ***/

#include <os_pse/ostore.hh> // PSE Pro header file
#include "BinaryTree.cpp"
#include "Data.h"
// Mark the classes to allow persistent allocation:
OS_MARK_SCHEMA_TYPE(BinaryTree<Data>);
OS_MARK_SCHEMA_TYPE(Data);
```

Generating the Schema

As mentioned before, you must link every PSE Pro application or library with an object file that contains information about the classes of persistent objects the application uses. *Schema generation* is the process of generating the *schema object file* from the schema source file. You run the PSE Pro schema generator (`pssg`) to generate a schema object file from the schema source file.

Running the pssg Utility

Here is the form of the `pssg` command line:

```
pssg.exe [ compiler-flags ][ -asof schema-object-file ] schema-source-file
```

`compiler-flags`: As part of schema generation, the schema source file is compiled by the C++ compiler. `pssg` passes *`compiler-flags`* to the C++ compiler.

`-asof` *`schema-object-file`*: Specifies the name of the schema object file. If you do not specify a name, the name defaults to `osschm.obj`.

`schema-source-file`: Specifies the name of the schema source file you previously created. This file's name must have the extension `.scm`.

The following `pssg` option is particularly useful for debugging purposes.

`-ashf` *`schema-header-file`*: Use this option to create a header file that contains relocation information for the marked classes in the *`schema-source-file`*. Note that the application never includes the schema header file.

Regenerating the Schema Object File

You should regenerate the schema object file whenever you add or remove class definitions or modify a class definition in your application or library.

You can use a make rule such as the following:

```
CC_OPTS = /DWIN32 /GX ...
schema-object-file: schema-source-file
  pssg $(CC_OPTS) -asof schema-object-file $@ schema-source-file
```

Default Compiler Switches

The `pssg` utility uses the following compiler switches. In case of conflict with user-supplied flags, these override the user-supplied flags:

- `/MD` instructs the compiler to use the `msvcrt.dll` run time.
- `/Od` Debug.
- `/Z7` MS C7 debug mode.
- `/Gx` Enable C++ exceptions.
- `/D_ODI_PSSG=1` Use this preprocessor macro to hide code from `pssg`. (ObjectStore's schema generator `ossg` uses `_ODI_OSSG_`).

Schema Generation in the VC++ IDE

In the VC++ IDE, you can create a custom build rule to compile the *schema-source-file*:

- Build command(s): `pssg /asof $(INTDIR)\osschm.obj` *schema-source-file*
- Output file(s): `$(INTDIR)\osschm.obj`
- Description: Generating ObjectStore PSE Pro typespecs

  You can use the Settings for `All configurations`.

Click on the Quickstart item in the ObjectStore/PSE program group for an example of how to set up the custom build rule:

- In the FileView, select `schema.scm`.
- Select the `Settings` item under the `Project` pull-down menu.
- Select the `Custom Build` tab.

Compiling the Code

Use the Microsoft Visual C++ compiler to compile PSE Pro applications and libraries. Use the following compiler switches:

- `/Gx` to enable C++ exceptions
- `/MD` to use the `msvcrt.dll` run time (multithreaded, DLL version), or
- `/MDd` to use the `msvcrtd.dll` `_DEBUG` run time

Linking the Components

Use the Microsoft Visual C++ compiler to link PSE Pro applications and libraries with the

- Schema object file
- PSE Pro library

Link against the library `osclt.lib` and run against the DLL `o5psepr.dll`.

For debug, link against the library `oscltd.lib` and run against the DLL `o5pseprd.dll`.

Upon inclusion, `os_pse/ostore.hh` sets the default library.

To use the MFC extension DLL, link with `osmfc.lib` or (for debug) `osmfcd.lib`, and run against `osmfc.dll` or (for debug) `osmfcd.dll`.

Typical Link Errors Related to Schema Generation

If you get a link error such as the following, your program uses the typespec (os_ts<Foo> in this case) for a class (Foo in this case), but your schema source file fails to mark the class.

```
osschm.obj : error LNK2001: unresolved external symbol
"private: static class os_pse::os_ts<class Foo>
os_pse::os_ts<class Foo>::the_instance"
```

Include the class's definition and mark it in the schema source file.

If the above error occurs for os_ts<class Foo*> (indicating a pointer) rather than os_ts<class Foo>, you marked a pointer type. The pssg utility cannot create typespecs for a typed pointer. A workaround is to add the following os_ts<T> specialization (Compare also to the stl42 example that defines STL list<Foo*>).

```
namespace os_pse {
  template <> class os_ts<Foo*> : public os_typespec {
  public:
    static inline os_typespec *get() { return os_typespec::get_
pointer(); }
  };
  // define more pointer types:
  // template <> class os_ts<Bar*> : ...
}
```

Yet another schema generation-related error that can occur during linking is an unresolved external for a virtual function table. An example of this error is

```
osschm.obj : error LNK2001: unresolved external symbol
"const Foo::`vftable'" (??_7Foo@@6B@)"
```

In this case, if class Foo has been marked in the schema source file, the pssg utility creates a reference for a virtual function table if class Foo contains one or more virtual functions. If an application never calls the constructor of class Foo, the VC++ compiler does not create a vftable for that class. A way to force the creation of that symbol into the .obj file is to add the following invocation of the constructor to your program to ensure that the function _force_vftable() is never called:

```
void _force_vftable(void*) {
  _force_vftable( new Foo() );
  _force_vftable( ..more classes..);
}
```

Static Builds

To link with the static version of the PSE Pro library, add the following compile switch:

- /D_OS_STATIC_LIB=1
- /MD, /MDd, /MT, or /MTd

Upon inclusion, os_pse/ostore.hh sets the default library based on these two compile switches. The PSE Pro installation contains the following static libraries built against different versions of the C-run time:

- libclts.lib

Use the multithreaded DLL version (`/MD`, `msvcrt.dll`).

- `libcltsd.lib`

  Use the debug, multithreaded, DLL version (`/MDd`, `msvcrtd.dll`).

- `libcltst.lib`

  Use the multithreaded static version (`/MT`, `libcmt.lib`).

- `libcltstd.lib`

  Use the debug, multithreaded, static version (`/MTd, libcmtd.lib`)

In the absence of `DllMain()` in the static PSE Pro libraries, the application has to explicitly notify PSE Pro on thread creation and termination. If a thread does not access persistent data, the functions do not have to be called.

```
static void objectstore::thread_attach(HANDLE h)
static void objectstore::thread_detach(HANDLE h)
```

`GetCurrentThread()` can be used to specify the handle `h`.

Schema Generation and Static builds

For static builds, make sure to add the `/D_OS_STATIC_LIB` compiler switch to `pssg`. If you do not set it, you see the following `LNK4049` warnings, which you can ignore, on `class os_basic_typespec` when linking statically with the generated `osschm.obj` file:

```
LINK : warning LNK4049: locally defined symbol protected:
void __thiscall os_pse::os_basic_typespec::cleanup(void)
 (?cleanup@os_basic_typespec@os_pse@@IAEXXZ) imported
void __thiscall os_pse::os_basic_typespec::initialize(void)
(?initialize@os_basic_typespec@os_pse@@IAEXXZ) imported
```

For all static builds, you see one other `LNK4049` warning, which can also be ignored, when linking statically with the generated `osschm.obj` file:

```
LINK : warning LNK4049: locally defined symbol
const os_pse::os_basic_typespec::vftable (
??_7os_basic_typespec@os_pse@@6B@) imported
```

Using MSDEV IDE

The PSE Pro installation adds the following directories to the IDE's directory search paths and to your system environment:

- Include files: `c:\odi\PSEPROC63\include`
- Executable files: `c:\odi\PSEPROC63\bin`
- Library files: `c:\odi\PSEPROC63\lib`

Make sure that these directory files are set correctly: in the Visual Studio IDE, select Tools | Options, and choose the Directories tab. Verify that the settings for Include files, Library files, and Executable files match the directories listed above.

Index

A

advantages 9
architecture 9
-ashf option to pssg 26
-asof option to pssg 26

B

benefits 9
building applications 25

C

closing databases
  description 23
  example 19
compiler switches
  schema generation 26
  source files 27
creating databases 22
creating schema source files 25

D

database roots
  creating, example 18
  defined 22
  description 22
  retrieving, example 18
  set value example 18
  using 22
databases
  closing 23
  closing example 19
  create example 18
  create root example 18
  creating description 22
  open example 18
  opening description 22
  retrieve root example 18
  saving 23
  saving example 18
  storing objects 15
debug compiler switch 26

E

entry points
  creating 22
  and data retrieval 22
examples
  class header file 12
  creating/opening database 18
  establishing fault handlers 17
  PSE Pro initialization 18
  retrieving root 18
  set value of root 18
  source file 12
exception handler, establishing 21

F

fault handlers
  establishing 21
  example 17

G

generating schema
  static builds 29
  typical 26
  VC++ IDE 27

H

header files
  Data class 16
  example 12
  PSE Pro 15

schema 26

I

initializing PSE Pro 21

L

linking
libraries 27
static builds for PSE Pro 28
typical errors 28

M

memory mapping architecture 9
MSDEV IDE 29

N

new operator 14

O

objectstore::initialize() 21
_ODI_PSSG preprocessor macro 26
opening databases 22
operator new
description 15
example 14
os_database::close() 23
os_database::create() 22
os_database::create_root() 22
os_database::open() 22
os_database::save() 23
os_database_root::find_root() 22
os_database_root::get_value() 23
os_err_database_not_found exception 22
OS_MARK_SCHEMA_TYPE() macro 25
os_pse/ostore.hh 15

P

persistence
database roots 22
modifying existing class 14
persistent new
description 15
example 14
persistent objects
creating 22
retrieving 22
saving 23
PSE Pro
benefits 9
building applications 25
header file 15
initialization example 18
initializing, description 21
modifications for persistence 14
sample data model 11
pssg utility 26

R

running pssg utility 26

S

sample data model 11
saving database example 18
saving databases 23
schema header file 26
schema object file 26
schema source files 25
schema.scm file 25
source file example 12
source files
Data class 16
main routine 17
static builds for PSE Pro 28
storing objects in database 15

U

unresolved external symbol 28

V

VC++ IDE 27