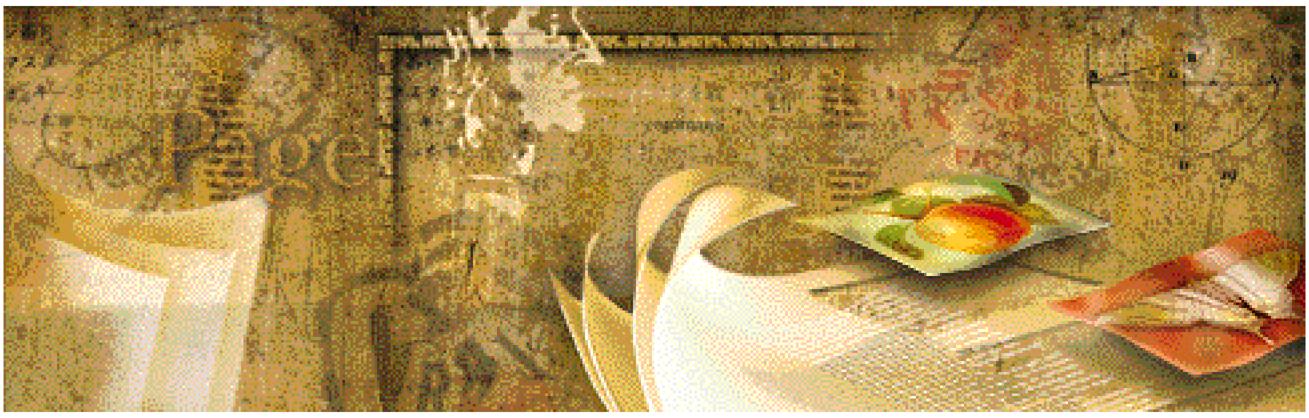
Adobe



PageMaker Class Library

Revised: 20 November 1995

Adobe PageMaker 6.0 Software Development Kit
PageMaker Class Library

Version History		
7 October 1995	Paul D. Ferguson	First draft

1 Introduction

The PageMaker Class Library, or **PCL**, is an exciting new way to create plug–in modules (formerly known as Additions) for Adobe PageMaker 6.0. This framework of C++ classes defines a powerful, elegant way to retrieve information from and issue commands to PageMaker.

Commands and queries can be expressed simply, in a syntax and style similar to the scripting language built into PageMaker. The PCL objects handle all aspects of communication with PageMaker. In addition, several low level classes permit customization for special situations.

Requirements

You should be familiar with C++ programming and have a basic understanding of how to create PageMaker plug–in modules. For more information about creating plug–ins, refer to the *Adobe PageMaker Programmer's Reference Guide* included with this SDK. Some familiarity with PageMaker is also helpful.

You should have the appropriate C++ compiler to build PageMaker plug–ins. On the Macintosh, the PageMaker Class Library is designed to work with Metrowerks CodeWarrior CW7 or later.

This release of PCL has not been used on Windows95; the next release of the SDK will support cross platform development using PCL.

Complete C++ source code for PCL is included. You may wish to browse the source code to better understand the concepts presented in this chapter.

Dependencies

The PageMaker SDK is designed with minimal dependencies on other software. The SDK source code does use standard ANSI C types like `size_t`, and library calls like `strlen()` and `strcpy()`. The examples may use other library routines (e.g. `sprintf()`) as well.

The PageMaker Class Library requires C++ compiler support for templates and exceptions. It does not require any run–time type identification (RTTI) support.

PCL does not require any other classes or other compiler or vender specific features, although PCL should be compatible with popular application frameworks like PowerPlant and MFC. Refer to your compiler's documentation for more details.

Design Goals

The PageMaker Class Library has four primary design goals:

- Define a simple, natural syntax to express PageMaker commands and queries in C++.
- Insulate programmers from the mechanics of issuing commands and queries, and manage the memory associated with them.
- Balance performance and memory considerations.
- Provide robust type checking and error handling.

The first two goals will be discussed in the next section, “Theory of Operation” on page 3. Performance and memory issues are discussed in “Performance Issues” on page 16.

Achieving the last design goal of robust type checking and error handling is a direct result of applying C++ features properly in the class library and in your plug–in code. PCL error handling is discussed further in “Error Handling” on page 14.

One of the plug–in examples (Colorer.add), which uses a variety of commands and queries, has been converted to use PCL. You can compare the original C code to that of the C++ version to see how much simpler and easier it is using the PageMaker Class Library.

Theory of Operation

The concept behind PCL is very simple. For each PageMaker command or query (there are over 200 of them), there is a corresponding PCL class. When you create an instance of one of these classes, the constructor for the class issues a command or query to PageMaker.

Here’s an example:

```
PGetPageNumber curPage;       // Get the current page number
if ((short) curPage == 27) {  // Check the page number value
    PPage newPage(33);        // Change to page 33
    ...
}
PPage oldPage(curPage);       // Change back to current page
```

In this example, we have created a query object (`curPage`), and two command objects (`newPage` and `oldPage`). These objects are created by declaring them with any appropriate parameters.

This simple syntax closely mirrors the PageMaker scripting language:

```
pagenumbers 1, -2, true, arabic, "I-"   // PageMaker scripting
PPageNumbers pn(1, -2, true, kNumArabic, "I-"); // C++ & PCL
```

The C++ code is more efficient than the scripting code, however, because no interpreter step is needed; all PCL commands are directly processed by Page-Maker.

Overloaded Operators

Notice that the PGetPageNumber class overloads `operator short()`, permit-ting the natural syntax in the “`if (curPage == 27)`” expression above. Many PCL classes use operator overloading in this fashion to achieve the first design goal.

Here’s a slightly more complicated example:

```
PDeselect deselect;  // deselect anything that is selected

// loop through each chosen page
for (short i = firstPage; i <= lastPage; i++)
{
  PPage thePage(i);               // change to page i
  PGetObjectList objectList;      // get list of objects on page i
  short n = objectList.Count();   // n is number of objects
  for (short j = 0; j < n; j++)   // examine each object
  {
    if ( objectList.cTypeOfObject == 4  // if circle or rect
      || objectList.cTypeOfObject == 5 )
    {
       PSelect select(objectList.nDrawingNumber); // select obj
       PColor color("mauve");  //set color of selected object
       PDeselect deselect;
    }
    objectList++;  // increment list
  }
}
```

In this example, we have created a number of PCL objects. The `objectList` object contains a variable amount of information about the objects on the current page (in this case, we're referring to PageMaker objects like text blocks, placed graphics, etc. rather than C++ objects). We can iterate over this list using postfix increment notation (`operator++`) to examine each object in the list.

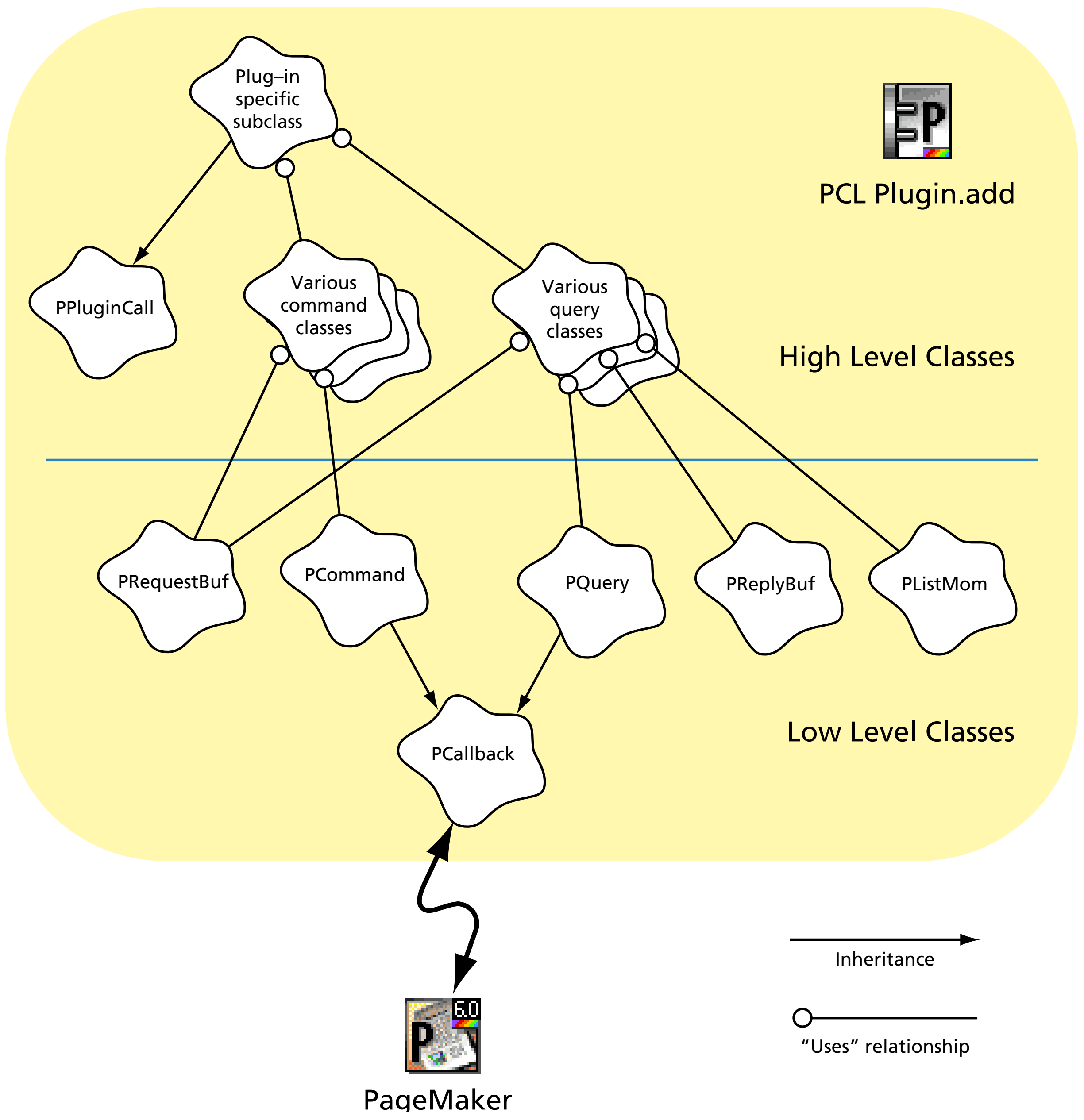
Memory Management

All the PCL objects created in this code example will be automatically destroyed when they go out of scope; this is guaranteed by the C++ language.

The (C++) object, `objectList`, may contain information about one, a few, or many dozens of (PageMaker) objects. The PGetObjectList destructor frees the memory block, which was allocated by PageMaker, associated with this information.

This illustrates the second primary design goal: insulate the programmer from the mechanics of issuing commands and queries, and manage the memory associated with them.

The PCL Class Hierarchy



Communication with PageMaker occurs through **PCallback** using its two subclasses **PCommand** and **PQuery**. (The diagram above shows this relationship using the Booch notation.) These base classes handle the mechanics of setting up and executing PageMaker commands and queries.

The high level classes create PCommand and PQuery objects in their constructors, as well as **PRequestBuf**, **PReplyBuf**, and (for list oriented queries) **PListMom** objects. The high level command and query objects properly format any parameters that must sent; query objects also save and manage the returned query results.

The **PPluginCall** class is discussed in "The PPluginCall Class" on page 6.

"Use once, throw away..."

Command and query objects are the software equivalent of paper towels. They are designed to do one thing: issue a single callback to PageMaker. They are lightweight, simple objects intended to perform one task and then be thrown away.

The command classes are simpler than query classes. The only public interface to command objects are their constructors; the only thing you can do to a command object is create it. Once its been successfully created, a command object's job is done.

Query objects also execute a single callback into PageMaker. Once it is created, you can extract the results from the query in your code using either specific functions (`PGetObjectList::Count()`), instance variables (`objectList.nDrawingNumber`) or overloaded operators (`PGetPageNumber pgNum; short j = pgNum;`). Once you no longer need a query object's information, its job is done.

Because there are many commands and queries that have a similar structure, some of the high level classes are implemented as template classes. For example, a number of queries, like PGetPage, return a single 16-bit short value and don't pass any parameters to PageMaker. Thus all these classes can be implemented through a template class. Refer to the file PShortQuery.h to see how PGetPage and similar classes are implemented.

The PPluginCall Class

This class represents the call from PageMaker that was initiated when the user chose your plug–in or your plug–in was executed by another plug–in. It is responsible for dispatching the incoming call based on the opcode in the `sPMParamBlock` structure.

This class also provides member functions to implement common actions that you may want to perform on the original parameter block, such as storing your plug–in's private data, or setting custom error messages. Refer to "Error Handling" on page 14 for more details.

Utility Classes

The utility classes **PRequestBuf** and **PReplyBuf** manage buffers for passing data between your plug–in and PageMaker. These classes have overloaded `operator<<()` and `operator>>()` functions to simplify packing and unpacking strings, shorts, longs, etc.

The **PListMom** class is used by PListQuery and its subclasses to manage queries that return lists.

If you use the high level classes you shouldn't need to create PRequestBuf, PReplyBuf or PListMom objects directly, but you will see them throughout the PCL source code.

Here's a snippet from the constructor from PGetColorInfo that uses both utility classes:

```
PGetColorInfo::PGetColorInfo(short cModel,
                             const char * sColorName)
: PCountQuery()
{
  PRequestBuf request(strlen(sColorName) + 4);

  request << cModel        // build request buffer
          << sColorName;

  PQuery query(pm_getcolorinfo, request);  // do query

  PReplyBuf reply(replyPtr);

  reply >> nPercent1      // extract reply information
        >> nPercent2
        >> nPercent3
        >> nPercent4
```

```
          >> cType
          >> nEPS
          >> bOverprint
          >> &sBaseColor
          >> cDefinedModel
          >> nInks;
}
```

2 Using PCL

This chapter describes how to create a PageMaker plug–in module using the PageMaker Class Library.

The sample plug–in Colorer.add illustrates typical programming with the PCL. You should take some time to familiarize yourself with the Colorer source code.

If you're developing on the Macintosh you may also want to examine the code that implements the overall interface to PageMaker; it is responsible for loading and executing your 680x0 or PowerPC code. This can be found in the "RAG1 Main" folder.

Class Names

As shown in the code snippets in chapter 1, PCL classes are distinguished by the letter "P" at the front of their class names. For the high level command and query classes, the rest of their names correspond to the specific command or query as documented in the *PageMaker SDK Guide*. Thus, the classes named "PSet..." represent PageMaker queries, while similar classes without "Set" in the class name represent PageMaker commands.

Although the *PageMaker SDK Guide* was not written specifically for PCL, the parameters and results from command and query objects correspond closely to what is shown in the Guide. You should use it as your primary reference to PCL classes, as it contains important information about each specific command and query.

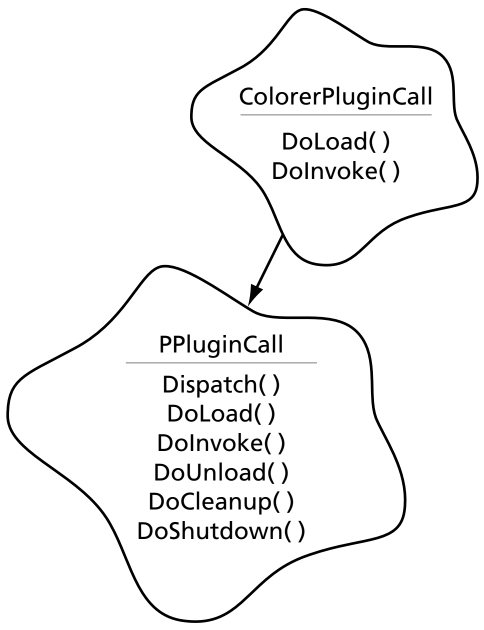
Getting Started

In the PageMaker Class Library folder, you should copy the files “PCL Plug-in Main.cp” and “PCL Plug-in Main.rsrc” to begin your PCL plug–in project. These are in the “(Project Stationaries)” folder. You will need to add or modify two lines to the main.cp source code to create an instance of your subclass of PPluginCall. The project stationaries folder also contains Mac template files for the 68K and PPC projects.

The main.cp source code declares and manages a global pointer **gPB** to point to the current `sPMParamBlock` that is passed from PageMaker. For maximum efficiency, this parameter block is used by all command and query classes to communicate with PageMaker. The global variable `gPB` is used to avoid having to explicitly pass a reference to this parameter block through every command and query constructor. Each command and query object maintains a local reference to the parameter block in the PCallback root class.

The `main()` function stores the old value of `gPB` on the stack, and restores it upon exit. This addresses any potential reentrancy issues that might arise through the use of this global.

To build a plug–in, you should create a subclass of PPluginCall. Upon entry to `main()`, an instance of your subclass of is created, and its function `PPlugin-Call::Dispatch()` dispatches the call to one of five other functions: `DoLoad()`, `DoInvoke()`, `DoUnload()`, `DoCleanup()`, and `DoShutdown()`.



The default behavior of each of these functions is to do nothing. To add behavior to your plug–in, override one or more of these functions; you must override `DoInvoke()`, the other four functions are optional. You can then add additional classes and functions to complete your plug–in code. Refer to the Colorer (C++) source code for an example of how to create a PPluginCall subclass.

High Level Command and Query Classes

The high level command and query classes provide wrappers around the PageMaker callback functions, which are documented in chapters 9 and 10 of the SDK Guide. Wrapping these callbacks provide an easy and efficient way for you to interact with PageMaker in your plug–in code. Refer to “Theory of Operation” on page 3 for basic information about the high level C++ syntax defined by these classes.

The main functions of the high level classes are to:

- provide a natural syntax for making PageMaker commands and queries, and
- perform proper parameter packing and unpacking using PRequestBuf and PReplyBuf objects.

Creating Command and Query Objects

The simplest way to use PCL classes is to create them directly on the stack as shown in chapter 1:

```
PSetPage pg(33);
```

While you can use the C++ `new` operator to create these objects:

```
PSetPage * pg = new PSetPage(33);
```

there is little reason to do so. By creating direct objects, you avoid the potential problem of forgetting to delete objects created with the new operator, a common source of memory leaks in C++.

The high level objects are lightweight; they impose minimal speed or space overhead on the process of issuing commands and queries, so it is practical to embed many of them within a single code module. Refer to “Performance Issues” on page 16 for more information about getting maximum performance with PCL.

The High Level Command Classes

Plug–ins make changes to PageMaker and its open publications through commands. Whatever the user can change through menus and dialogs can be performed efficiently and automatically with command objects.

PageMaker callbacks vary widely in the parameters required. The command classes enforce proper callback parameters by their constructor(s) parameters. Constructor parameter lists also contain default values as indicated in the command descriptions (chapter 9 of the SDK Guide).

Some command classes have a constructor with reference to the equivalent query object as as its parameter. This simplifies changing one or more of the object's parameters. The following code snippet illustrates its usage:

```
PGetWordSpace curWordSpace;              // get current settings
curWordSpace.dWordMin += 5;              // change word min
PWordSpace newWordSpace(curWordSpace);   // update publication
```

Variable Length Commands

Several commands require a variable number of parameters:

- the Book command,
- the Tabs command,
- the Import command.

You must build a proper PRequestBuf object containing the complete list of items for these commands, and use it as a constructor parameter:

```
PRequestBuf request(numItems * sizeof(long) + 2);
request << numItems;  // size of list
for (short i = 0; i++ < numItems; )
    request << tabstop[i];
PTabs tab(request);       // set tab stops in current story
```

The High Level Query Classes

Plug–ins obtain information from PageMaker and its open publications through queries.

Like the high level commmand classes, the high level query classes wrap the process of making queries into PageMaker. These classes are responsible for enforcing proper callback parameters where needed, and manage the information returned by PageMaker. This section discusses how to use query objects.

Working with Query Results

Query obects are more complex than command objects. A command object's only public interface is its constructor; query objects also have various functions to access the information returned by PageMaker.

The public interface to get information from a query object depends on the type of query. If you examine chapter 10 of the *SDK Programmer's Guide*, you will see that the information returned from queries can be:

- a single value, which may be fixed (e.g., a 16-bit short) or variable length (a string),
- a block, or structure of values, which may be fixed (no strings) or variable length (one or more strings),
- a list of values, which may be single values or blocks, or
- an initial block of values followed by a list.

In two queries, the blocks in a list themselves each contain another list. These are handled as special cases by those high level classes.

Because of this complexity, the interface to query objects has some unique features. To show how the interface works, let's begin with two simple examples and work our way up.

```
PGetLeading lead;      // Get current leading value (a short)
if (lead == 12) ...    // compare value to another short

PGetPubName pubName;   // Get current publication name
strcat(x, pubName);    // copy name to x
```

In these two cases, the query is a single value, and thus we treat the object itself as if it were the return value. Operator overloading is used in these classes.

The next example shows how to access a query object that contains a block of information.

```
PGetSpaceOptions spaceOptions;
if (spaceOptions.bAutoKerning == true &&
    spaceOptions.cLeading > lead) ...
```

The PGetSpaceOptions class contains public variables with names corresponding to the names used in the *PageMaker SDK Guide.* For single value queries, or string values, the query object acts as a read–only structure. You cannot assign a new value to a query object field:

```
strcpy(spaceOptions.sInkName, "wacky green");  // ERROR!
PGetLeading lead;       // Get current leading value (a short)
```

```
lead = 12;              // ERROR!
PLeading setLead(12);   // Use command object instead.
```

The next example illustrates a query that contains a list of information.

The GetColorInfo query returns a complex data structore with an initial block of information, followed by a list. Both the initial block and the list items are variable length because they each contain strings.

```
PGetColorInfo colorInfo(-2, "mauve");
short theType = colorInfo.cType;    // check type
strcpy(x, colorInfo.sInkName);      // first ink name in list
short ink = colorInfo.dInkLevel;    // first ink level
```

Use the `Count()` member function to find out how many items are on the list. You can then iterate over the list using the postfix `operator++` on the object. `Operator++` wraps around when it gets to the end of the list, or you can use the `Reset()` function to reset the list to its first item.

```
short i, j, numColors = colorInfo.Count();
for (i = 0 ; i < numColors; i++)
{
  strcpy(x, colorInfo.sInkName);         // get i-th ink name
  if (colorInfo.dInkLevel > threshold)   // and ink level
    break;
  colorInfo++;                           // iterate the object
}
colorInfo.Reset();  // so next iteration starts at beginning
```

You should always check `Count()` before referencing object variables that are part of the list, such as `colorInfo.sInkName`. If `Count() == 0`, the values of these variables are undefined, but the object cannot detect the error. `Operator++()` will throw an exception (CQ_FAILURE), however, if there are no list members.

Error Handling

In a perfect world, a PageMaker plug–in would never cause PageMaker to crash. The PCL exception mechanism is designed to prevent this from happening. However, you must still exercise some care in your code to avoid an unpleasant surprise to your plug-in user.

You should always program with the assumption that ***any command or query object may throw an exception when it is created***. If you don't want control to immediately return to PageMaker, you must handle the error in your code.

PCL uses the C++ exception mechanism to report errors. The function `PCall-back::CallPageMaker()`, which issues the actual callback to PageMaker, will throw an exception if the return value from PageMaker is non–zero. Other functions—primarily constructors—may throw exceptions for problems that occur within them, for example if an invalid (null) pointer is passed in. All errors thrown by PCL are error code numbers, of type `PMErr`.

The PCL main.cp source file contains a custom `unexpected()` error handler, which helps prevent incorrect exception situations from terminating Page-Maker.

In general, PageMaker will return an error code—and thus PCL will throw an exception—for invalid parameter data (for example, verifying that the parameter you pass to PPage to set the current page is within the range of actual pages). PCL classes do not validate parameters sent to or received back from PageMaker. If you wish to handle such errors gracefully, you should either perform explicit data validation in your code and/or implement try/catch handlers in the appropriate points in your code.

You can use the `PPluginCall::BuildErrorMessages()` function to create custom error messages. For errors that are returned from PageMaker, the standard error messages will often be adequate, but you may want to provide more information to the user so they can take corrective action, or assist you in troubleshooting their problem.

```
try
{
   PGetPrivateData privData(...);
}
catch (PMErr err)
{
   if (err == CQ_NOPDATA)
     this->BuildErrorMessages("You don't have any private data",
                              "Doh!");
    // now rethrow error, or continue elsewhere
}
```

A plug–in should never cause PageMaker to quit unexpectedly, which is the default behavior of an uncaught exception. Therefore, your plug–in code must catch all exceptions thrown, and report an error back to PageMaker if appropriate.

The `main()` function defined in main.cp contains `catch{ }` clauses that trap all exceptions and returns an appropriate `PMErr` to PageMaker, so you shouldn't have to do anything special to avoid this problem.

Potential Exception Handling Problems

One common situation that requires careful programming is creating command or query objects within a destructor. The C++ language explicitly defines the case of nested exceptions as a fatal error, which can occur when an exception is thrown within a destructor.

Here's an example:

```
class StSavePage  // a stack class to save/restore current page #
{
public:
  StSave() { }
  ~StSavePage() { PgPage restore(curPage); }

private:
  PgGetPage ourPage;   // on creation, saves current page #
};

...

void MyPluginCall::SomeFunc()
{
  StSavePage save;      // save current page # on stack
  PgPage gotoPg(-999);  // oops! an exception will be thrown!
}
```

When the `PgPage gotoPg(-999);` statement executes, an exception will be thrown. If the StSavePage destructor were to also cause an exception—which could happen if `MyPluginCall::SomeFunc()` had deleted a bunch of pages including the saved page—PageMaker would terminate due to nested exceptions.

To prevent this, you should explicitly catch all exceptions that might occur within the destructor:

```
StSavePage::~StSavePage()   // this is better...
{
  try { PgPage restore(curPage); }
  catch (...) { }
}
```

Performance Issues

There are performance tradeoffs with any class library, and PCL is no exception. While great care has been taken to maximize performance, there are several situations where performance considerations—both size and speed—should be noted.

As with any performance tuning, you should perform careful testing and profiling to ensure that changes will provide measureable, meaningful improvements in your software.

Speed Issues

The “use once, throw away” philosophy of the command and query classes introduces a slight performance overhead for the construction and destruction of objects. Because of the lightweight design of the classes, it is unlikely this will be a significant cause of a performance bottleneck, especially when you consider the processing that PageMaker does as the result of a command or query callback.

The overhead for these objects is mostly in some additional copying of parameters into a PRequestBuf object, or query results out of a PReplyBuf object. String values returned by queries are normally not copied by the high level objects.

Size Issues

The high level wrapper classes can lead to an explosion of classes; there is one class for every command and query. Great care has been taken to optimize each class with respect to its code and data size, but if you are very tight for space, you may want to look at ways to economize (of course, the first choice for economizing may be to write the program in C, since just using C++ introduces an increase in code size due to the complexity of the language and its libraries).

The high level classes are fine grained; only classes that you use in your code will contribute to the size of your plug–in executable. The high level wrapper classes do contribute to code size, so in some situations you may want to consider using low level classes instead of high level ones. In most cases, however, the savings will only be a few bytes per high level class replaced.

The template classes are defined in header files, and will only be instantiated when used in a source code file. If the same classes are used across multiple modules, you may want to compile a single instance of those classes in one place to avoid the potential for redundant code increasing your code size. Again, given the lightweight design of the template classes, this is unlikely to be a concern in most situations. Refer to your compiler documentation on how to explicitly instantiate template classes.

You can use the low level classes to build custom high level classes that replace high level PCL classes (although with some loss of simplicity and increase in potential programming errors). Simple commands are the most likely candidates for this:

```
PgSetPage pg(33);            // high level object to set the page
PgCommand(pm_setpage, 33);   // equivalent low level object
```

Query classes, and more complex command classes, are not as easy to replace. You may have to create your own PRequestBuf, PReplyBuf, and PListMom objects to accomplish the same results as the high level class.

The wrapper classes do provide some benefits for the small increase in size. Most importantly, they make sure that proper parameters for a command or query are provided. At a minimum, they provide a consistent interface for all commands and queries regardless of how complex or simple they are.