Chapter 2. C++ Application Framework

Topic: Ignore

While every ACIS–based application is different, each application developed in C++ using ACIS will need to follow a similar basic framework. This includes initializing and terminating the modeler and components, checking the results of API calls, etc.

Initialization and Termination

Topic: *Modeler Control, *Building Applications

Applications must initialize the ACIS modeler and the individual component libraries before use, and terminate them after use.

APIs to Start and Stop the Modeler

Two APIs are provided for starting and stopping the ACIS modeler.

api_start_modeller	The API api_start_modeller starts the modeler, defines some global variables, and does a simple check on whether static initializers have been called (which can be a problem for non-C++ application developers). Its call must precede calls to any other API, DI function, or class method.
api_stop_modeller	The API api_stop_modeller attempts to release all memory allocated by ACIS. The application should not attempt to reference any data returned by earlier calls to APIs or DI functions after calling api_stop_modeller. Its call must not be followed by calls to any other API, DI function, or class method.

Note *These APIs use the British spelling of the word “modeler” in their names. This spelling uses two* ls*:* modeller.

Component Libraries

The libraries for each ACIS component must be initialized before use and terminated after use. When an ACIS component is initialized, it initializes any components upon which it depends, so an application only needs to initialize the highest level components in the flow of dependency that it uses. It is the developer’s responsibility to make sure any component is initialized by the application before use.

Any component library that was initialized must be terminated. Libraries should be terminated in the reverse order from which they were initialized, to avoid memory problems.

Refer to Chapter 3, *Object Libraries*, for information about the functions for component library initialization and termination. Refer to the *3D ACIS Getting Started Guide* for a component dependency diagram.

Other Functions

Some ACIS components may require calls to additional initialization or termination functions. For example, api_rh_initialise_image_utilities initializes the rendering base Image Format Utilities Library. Refer to the function lists in online help for information on initialization and termination functions.

Checking API Results

Topic: *Building Applications

Each API function returns a result as an outcome object, which indicates the success or failure of the API. The result of every API call should be checked by an application (using outcome methods), followed by appropriate error handling.

Refer to the outcome class reference template in online help for more information.

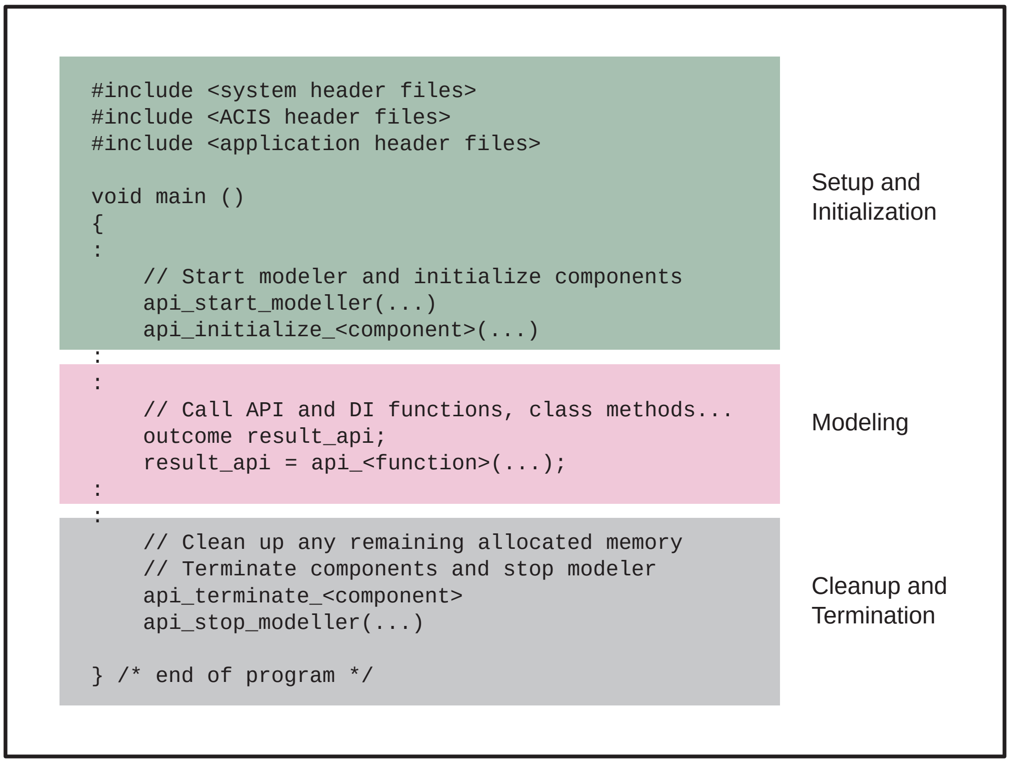
Application Layout

Topic: *Building Applications

The most simple ACIS–based C++ application can be broken down into three sections:

Setup and initialization	Include appropriate header files, start the modeler, and initialize components.
Modeling	Perform all modeling (and other application) functionality.
Cleanup and termination	Clean up any remaining memory allocations, terminate components that were initiated, stop the modeler, and terminate the program.

Refer to the pseudocode in Figure 2-1. This general application layout also applies to more complex applications, but the central “modeling” section would include such things as the provision of an appropriate user interface (e.g., windowing, graphical input/output, etc.), file management, model management, application data manipulation, memory management, etc.



```
#include <system header files>
#include <ACIS header files>
#include <application header files>

void main ()
{
:
    // Start modeler and initialize components
    api_start_modeller(...)
    api_initialize_<component>(...)
:
:
    // Call API and DI functions, class methods...
    outcome result_api;
    result_api = api_<function>(...);
:
:
    // Clean up any remaining allocated memory
    // Terminate components and stop modeler
    api_terminate_<component>
    api_stop_modeller(...)

} /* end of program */
```

Figure 2-1. C++ Application Layout

Simple C++ Application Example

Topic: *Examples, *Building Applications

The code snippet in Example 2-1 illustrates the basic framework of a very simple ACIS–based C++ application. It starts and stops the modeler, initializes and terminates libraries, checks the results of API calls, etc. This example uses a simple macro for API result checking and error reporting. Complex applications may need sophisticated error handling.

The header files (or paths) and/or the API arguments may not be those for the current release.

C++ Example

```
// Include header files

#include <stdio.h>
#include ”kernel/acis.hxx”
#include ”kernel/kernapi/api/api.hxx”
#include ”kernel/kernapi/api/kernapi.hxx”
#include ”constrct/kernapi/api/cstrapi.hxx”
#include ”kernel/kerndata/top/body.hxx”
#include ”baseutil/vector/position.hxx”
#include ”kernel/kerndata/data/entity.hxx”


// Create a simple macro for checking the results of an API call
// (using outcome class); this macro inserts the required semicolon
// following function call

#define CK_OK(API_FUNCTION)                              \
    {                                                    \
    outcome oc = API_FUNCTION;                           \
    if (!oc.ok())                                        \
        {                                                \
        err_mess_type errNum = oc.error_number();        \
        fprintf(stderr, ”%s (%d): %s (%d)\n”,            \
        __FILE__, __LINE__,                              \
        find_err_mess(errNum), errNum);                  \
    } /* end if */                                       \
        }                                                \
    }

void main ()
{
    // Start the modeler and initialize component libraries

    CK_OK(api_start_modeller(0))    // memory size as needed
    CK_OK(api_initialize_constructors())


    // Create a square block, called MyBlock; API api_solid_block
    // defines a block using two ACIS positions, and each position
    // is defined using 3 (xyz) coordinates

    BODY *MyBlock;
    CK_OK(api_solid_block(
        position (–20, –20, –20),
        position (20, 20, 20), MyBlock))


    printf(”Created the block.\n”);


    // Clean up memory allocation
    CK_OK(api_del_entity(MyBlock))
    // Terminate component libraries and stop modeler

    CK_OK(api_terminate_constructors())
    CK_OK(api_stop_modeller())

}
```

Example 2-1. Simple Application Example