Table of Contents

Introduction

Welcome to ***CThrough***, a full-scale, integrated, graphical development environment for C and C++. This introduction is intended as a ”tour d'horizon” for programmers who are already familiar with structured programming in C or object-oriented programming in C++.

The first thing to learn about ***CThrough*** is that it does a lot more than just browsing and source code editing. It is a set of highly integrated tools that cover the following tasks:

- Source management (browse, edit and document C++ sources)
- Change management (keep track of class changes and versions, show and merge libraries and class deltas)
- Application building (build `*.obj`, `*.exe` and `*.dll` files)
- Test support (handle OS/2 exceptions within applications and analyze heap problems)

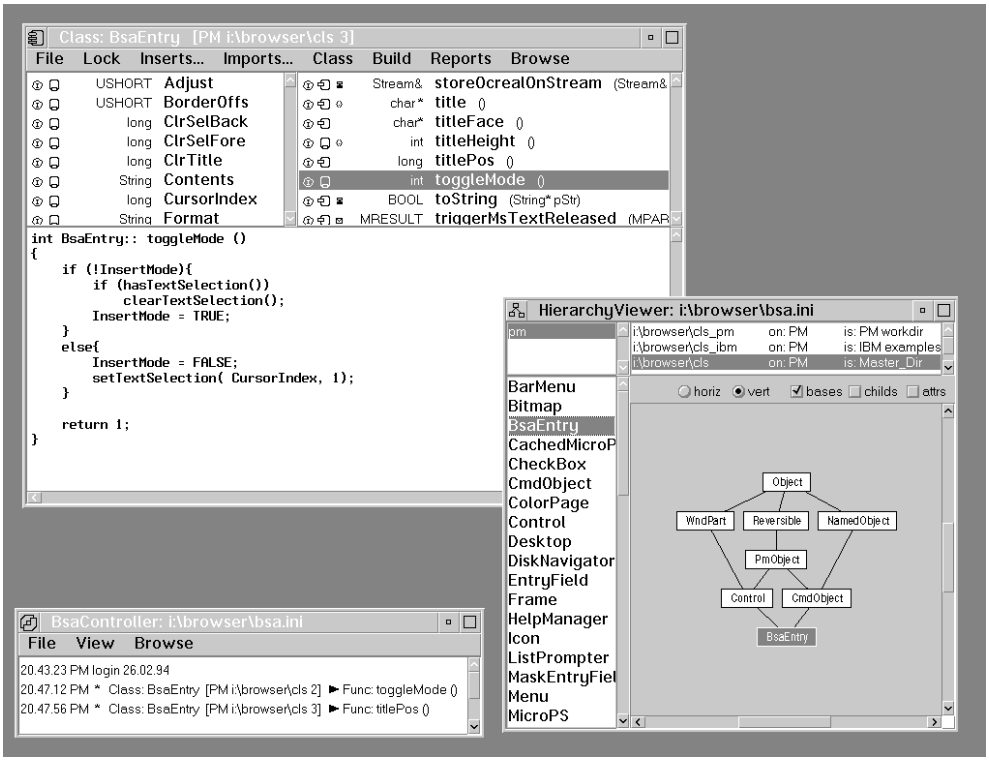
Support for each of these tasks is described in the following sections. The screen shots will give you an idea of ***CThrough's*** versatile graphical user interface. With respect to the individual windows it is useful to remember that

- most of the views can have multiple, simultaneously used instances
- *drag and drop* is the primary interaction scheme for various views
- almost all panes of the described windows have context-sensitive popup menus and extensive online help available.

Browse, Edit and Document C++ Classes

Working with classes instead of files is a main objective of ***CThrough***. All class sources should be processed as logical units, e.g. C++ function and/or data members, and should not be treated as raw text. Object-oriented programming in C++ provides some powerful abstraction mechanisms, for example inheritance. Without appropriate tools it is easy to get lost in this ”3rd dimension”. To be a successful OO programmer, you have to understand both, the programming language and your class library. Complex class libraries easily extend to more than two thousand methods. In order to make efficient (re)use possible, you must structure this information and provide different views.

The picture on the next page shows the most frequently used views of ***CThrough***. The first window that comes up after starting the system is named the **Controller**. It logs all developer activities such as login, function changes, relationship changes, etc. and controls session related topics (e.g. setting user preferences or toggling source code logging). It can also be used to launch additional tools. Closing the controller terminates the current ***CThrough*** session. The **Controller** is ***CThrough's*** main application window; it is the only window that cannot have multiple instances concurrently.



Navigate through your classes (*HierarchyViewer* and *ClassEditor*)

After a short initialization phase (typically less than 10 seconds for 300 classes on a 66 MHz 486 system) the **HierarchyViewer** comes up. This view shows the contents of the class library. It is divided into four panes:

- DirectoryPane
- AttributePane
- ClassPane
- GraphPane

The *DirectoryPane* shows all directories where ***CThrough*** is looking for C++ source files. Directories are searched in a predefined way (from top to bottom). Classes with identical names located in several directories are shown only once (i.e. if a class is found in more than one directory, the first occurrence will be used). You can view class locations and move or copy classes between directories. The list of directories initially scanned is one of many settings you can define in a user configuration file.

Selecting a class (e.g. by clicking the mouse in the *ClassPane*, see below), automatically indicates the corresponding directory within the DirectoryPane. Similarly, selecting a directory in the DirectoryPane shows only classes contained in the selected directory.

The *AttributePane* may be used to further narrow the display of classes in the ClassPane. You may define arbitrary attributes ("categories") to show just classes that match (or don't match) the specified selection criteria. Attributes are user defined names, such as **`<PM>`** (for **P**resentation **M**anager related class) or **`<Test>`**, that can be attached to classes.

The *ClassPane* shows an alphabetical list of all matching classes (with appropriate attributes or within a specified location/directory). Most functions used for day-to-day work can be launched via the popup menu of this pane. There are functions that work on the selected class only (e.g. open the **ClassEditor**) and also functions that operate on all selected/marked classes (e.g. build/update class archives). The ClassPane can be expanded to

show additional information such as corresponding file sizes, date and time of file creation/modification and all attached categories.

The *GraphPane* shows the inheritance relationships of the currently selected class. There are several display options to control how the inheritance graph is built. If the **<bases>** option is checked, then all the base classes of the selected class are displayed. If **<childs>** is checked, then all classes derived from the selected class are displayed. Most of the class manipulation functions available for the ClassPane can be also activated from the GraphPane with identical semantics. Within the GraphPane the group functions work on all classes displayed in the graph.

The **ClassEditor** is the most important tool of all. It is used to display and edit a single class and its associated source code. The **ClassEditor** consists of three panes:

- DataPane
- FunctionPane
- FunctionDefinitionPane

The *DataPane* shows all data members of the class. Each line starts with symbols for the standard attributes of the corresponding data member. The first symbol shows the association (**<instance>**, **<class>** or **<global>**). The second symbol is used to represent the access level (**<private>**, **<protected>** or **<public>**). There is an optional third symbol used for combinations of other attributes such as **<extern>**. The next column shows the type of the data member, followed by its boldfaced name. If a line is selected within the DataPane, then the FunctionPane shows only those function members that use the selected data member. Entering new data members and modifying existing ones is accomplished via a special dialog that can be launched from the DataMember's popup menu. A filter dialog enables the selective display of data members; attributes like association, access level or user defined categories similar to those used at class level may be specified. New data members are entered via another interactive dialog, which also takes care of C++ syntax specifics. This dialog is also used for modification of existing data members.

The *FunctionPane* is responsible for displaying all function members of the class managed by the ClassEditor. Like the DataPane, the FunctionPane uses one line per function member to show its association, access level and additional attributes together with the return type, boldfaced name and signature. Symbols are used to show C++ characteristics, e.g. **<inline>** or **<virtual>** function members. Selecting a function member within this pane loads its corresponding definition into the FunctionDefinitionPane. New function members are specified via a dialog, which also takes care of attribute dependencies. Function signatures may also be entered directly via the FunctionDefinitionPane.

The *FunctionDefinitionPane* is used to display and edit a particular function definition (its source code). It always holds just one function definition at a time. This pane is essentially an editor used to enter or change the function source code. In addition to traditional editing capabilities it allows you to record and play back keyboard macros, match all kinds of braces or find and replace text. There are even provisions for displaying online help for selected functions or querying the implementors of methods used within the system - just place the cursor on the keyword in question and choose *Info* or *Implementors* from the pane's popup menu. Context-sensitive help is also available.

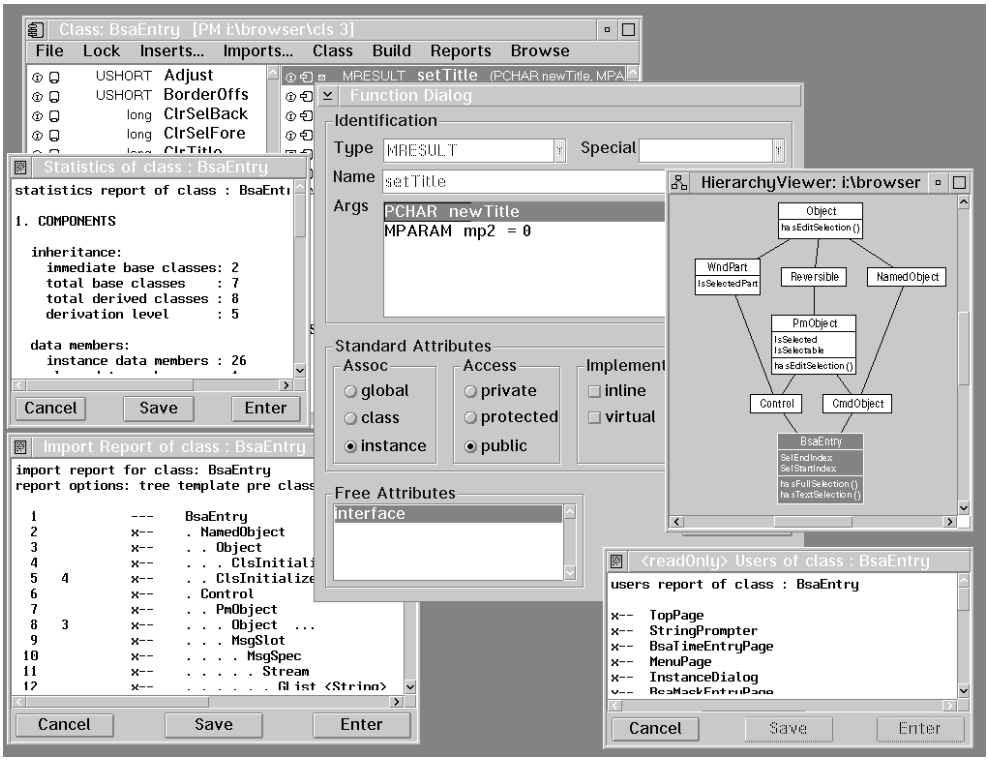
One important aspect of the **ClassEditor** deserves special attention: developers do not have to deal with the C++ class definition code because it is completely managed by the **ClassEditor**. Based on the member definitions (name, type and attributes) entered in the FunctionDefinitionPane or using the associated dialog, the class definition is generated automatically by ***CThrough***.

Two additional types of information used within a typical C++ class can be specified with the **ClassEditor**: *Imports* and *Inserts*.

Imports map to C++ **`#includes`**. They can be entered by opening the associated dialog and using drag/drop techniques (e.g. selecting a class within the HierarchyViewer and dragging it onto the import dialog). The same technique may be used for "traditional" header files (e.g. string.h) using ***CThrough's*** HeaderBrowser (alternatively you can simply enter the name of the desired import file within the import dialog). Class imports are shown as class names instead of filenames.

Inserts are simply pieces of unparsed text that can be included into the class. Usually they contain local type definitions, enums, typedefs and similar kind of information.

The user has complete control over the order and the position of imports and inserts. The possible options are (a) preceding the class definition, (b) following the class definition or (c) preceding the implementation section.



More tools to analyze and edit a class

The picture above shows additional windows that can be spawned from the **ClassEditor**. The *FunctionDialog*, as already mentioned, is used to enter new functions and to modify existing ones. Using the function dialog there is no need to remember the intricacies of the syntactic details of a C++ function definition. As an example, to inline a function member, just select the appropriate checkbox. ***CThrough*** not only inserts the required C++ keywords at the correct place but also automatically moves the function body into the header section of the class file. Unchecking the **`<inline>`** option reverses this action.

Various reports can be generated for a class. All of them are displayed using the same type of *TextWindow* that can be used to modify and/or print the report text. Following you will find some reports available for a class:

1. **Import Report**. This report shows all imports of the class either hierarchically (using the preprocessor **`#include`** order) or alphabetically sorted (with all users of the par-

ticular import appended). This is useful to detect problems such as infinite recursive mutual imports. The displayed screenshot example shows a hierarchical (tree) report.

2. **Users Report**. This report shows all users of a class, together with the particular import position (pre, post or implementation).
3. **Statistics Report**. This report lists various statistical information for the class (number of base classes, number of derived classes, data members, function members, lines of code, number of comment lines, etc.).
4. The **HierarchyViewer** in this screen shot shows two different display options compared with the **HierarchyViewer** displayed in the first picture. The GraphPane has been "zoomed" in order to provide more room and a detailed class display has been selected. This display shows selected data and function members of a class in alphabetical order within the inheritance graph. Although this is not an ideal way to display all members in a class hierarchy, it is a good feature when focusing on a specific functionality that works across multiple inheritance levels. It also proves invaluable for enhancing textual documents.

To document a class you first generate an outline of the class' description. This is done automatically using ***CThrough's*** document creation tool. The outline file is essentially a structured synopsis of the class and its components (data and functions members, bases, friends, etc.). You can view and edit the file using the **OutlineBrowser**, a tool for editing hierarchically structured texts. Usually in the next step you will add descriptions of the various class components and the class itself. Don't worry if the class changes after you have documented it; the new class and the existing description can be merged automatically. Also forget about complex formatting and layout. Just type in the text, concentrating on the contents.

Selecting the appropriate menu option in the **OutlineBrowser** you may now generate your favorite format using predefined descriptor files. These descriptor files use a simple language to facilitate the conversion of ***CThrough's*** internal outline format into almost any format you like. Provided with the current release of ***CThrough*** are descriptor files for the IPF format (OS/2 native help language) and TeX (so you can use one of the publicly available TeX tools to generate a ready-to-print document). Of course you may write your own descriptor files to support further conversions.

The **OutlineBrowser** may also be used to capture and administer error lists, user reports, todo-lists or any kind of hierarchically structured texts. It supports selective report facilities, queries and user defineable categories to group text into various sections.

Keep Track of Class Changes and Versions

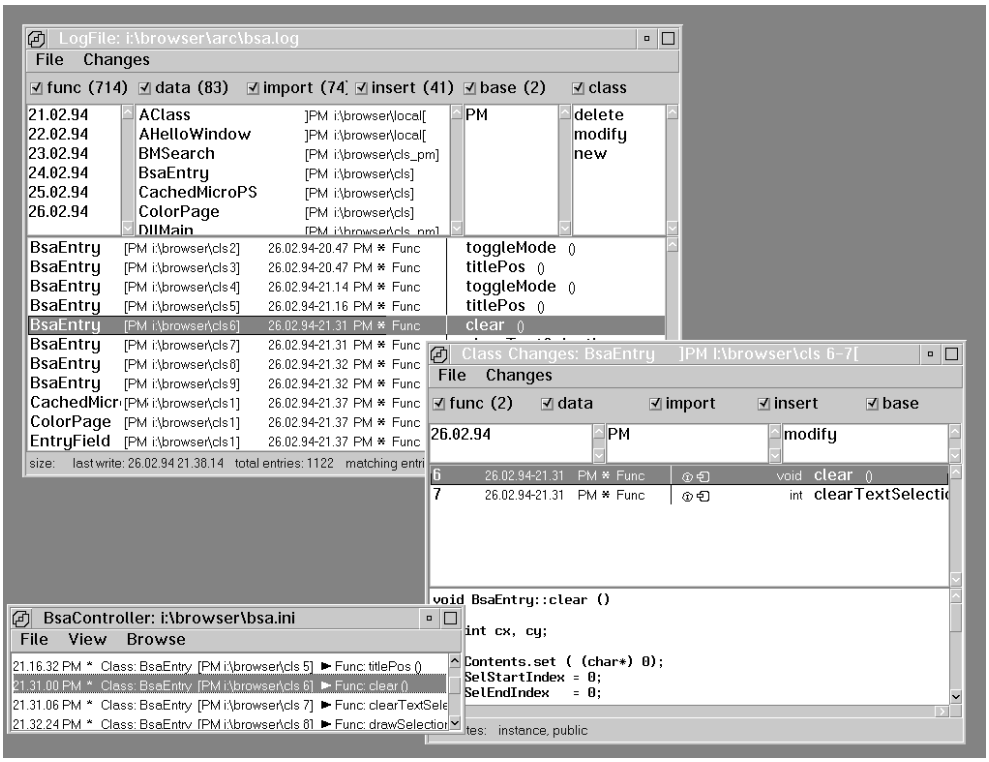
As the number of classes grows, it becomes increasingly important to keep track of changes. Since it is an OOP objective to increase code reuse, many classes are often used in different projects. Changes of a class caused by one project could adversely effect other users of the class. If the class is a potential base class, things may be even worse because changes of implementation details may have an effect as well. There is only one way to handle such problems: a close tracking of all changes. Since changes have to be understood afterwards, it is again important that logging them is done in logical units (class, member etc.) compared to a simple file basis (filename, lines).

The basis for ***CThrough's*** logging mechanism is a unique class identification scheme. Each class file contains information about the location (point in space, i.e. machine and path) and revision number (point in time) of the particular class. Each time a class is

modified (e.g. by saving a modified function definition), the revision number increases automatically. This means that every change can be identified by specifying a location and a corresponding revision number (e.g. **`<BsaEntry [MyMachine e:\mypath 123]>`**). This information is displayed in various views (e.g. **ClassEditor** title bar, Log-file entries, etc.).

As mentioned earlier, every programming action such as saving a function or adding an import is logged in the **Controller**. The corresponding log entry is not only displayed but also written to a log-file. The system can even be configured to use multiple log-files simultaneously (e.g. one log-file could be private to the user, another one could be shared by a developer group). Each log entry includes information concerning date, time, user id, operation (*add*, *modify*, *remove*), class name, class location, class revision number, the type (e.g. *function*) and the name of the changed item (e.g. function name). Also the symbolic machine name is logged to enable a precise subsequent log analysis.

The **LogFileBrowser** is used to analyze the contents of the log file. It is used not only to display the contents in a structured way but also to answer specific questions using predefined queries. Questions like "Show all function modifications for class *String* done by Fred M. after March 16th, 1994" may be answered by a few mouse clicks.



Basic logging mechanisms

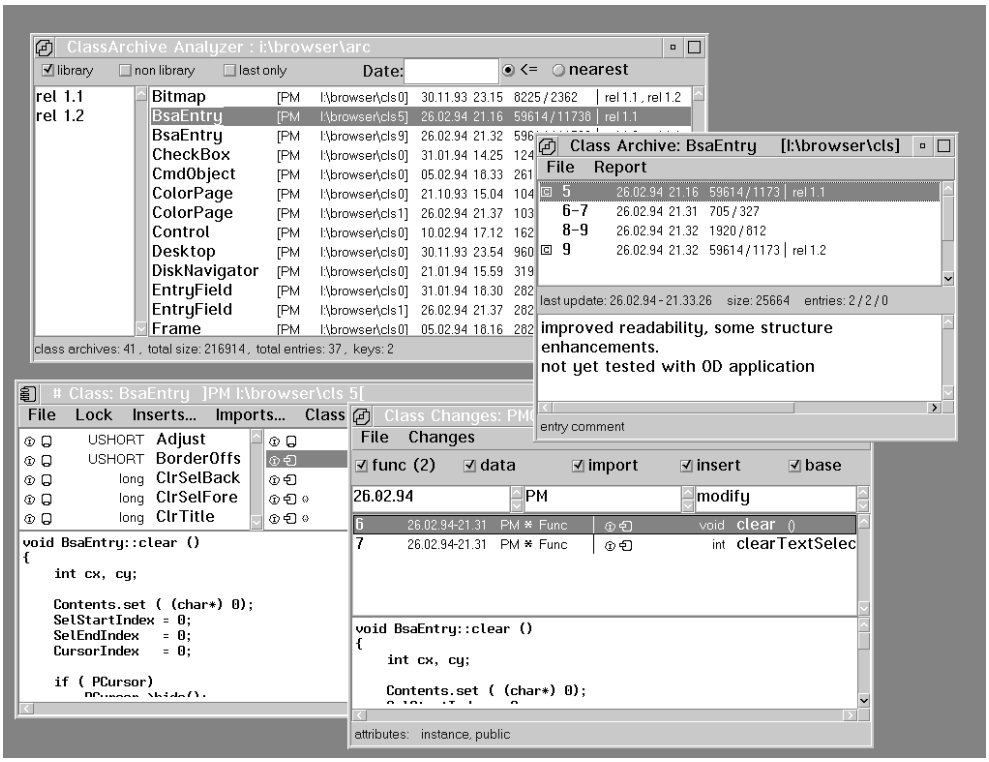
The pane in the upper half displays all change dates, all changed classes with locations and revision numbers, the names of all developers who have performed changes and all types of change operations. Selecting an entry in one of these panes filters information to the right of it. Essentially this connects all selected entries by a logical „and“. Selecting a date results in the display of all changes done since that date. The pane in the lower half lists all matching log events showing information similar to the one displayed by the **Controller**. Below the menubar a list of CheckBoxes enables filtering by item type. Checking

only **<func>** results in the display of events logged for function members only. The number of entries of this particular type is appended in braces.

The contents of the item changed, such as the function's source code, may also be included in the log file. This is particularly useful when performing sensitive modifications, e.g. on classes with a high fan-out. The logging mechanism for sources can be activated on a class or session basis. On a class basis, all changes for a particular class are logged in a class related file. Using session based logging, all changes done during a ***CThrough*** session are logged in a user selectable file. Class logging can be toggled using a **ClassEditor** menu option. Session logging can be activated or disactivated using a **Controller** menu option. Class logging may also be automated by attaching the special attribute **<monitor>** to a class. Each time a **ClassEditor** is opened on such a class source-based logging is done automatically.

All resulting change-files can be browsed using the **ChangeBrowser**. Its upper three panes show the dates, the user names and the types of change operations that appear in the change-file. Similar to the **LogBrowser**, these panes can be used to filter the entries displayed in the middle pane. The middle pane shows matching change log entries using the same format as the **LogBrowser**. Selecting an entry in the middle pane updates the lower pane, displaying the appropriate text or function definition. It is of course possible to restore changes back to the current class.

Monitoring alterations to change-files is a useful mechanism for sensitive operations and classes, but it produces some overhead that might not always be appropriate for all classes. This is where the archive mechanism comes in. A ***CThrough*** archive is essentially a ZIP (compressed) file that may even be handled outside of ***CThrough*** using publicly available tools. Archives are kept in user configurable directories, e.g. on a dedicated server. There are two types of archives:



Archive class revisions and change-files