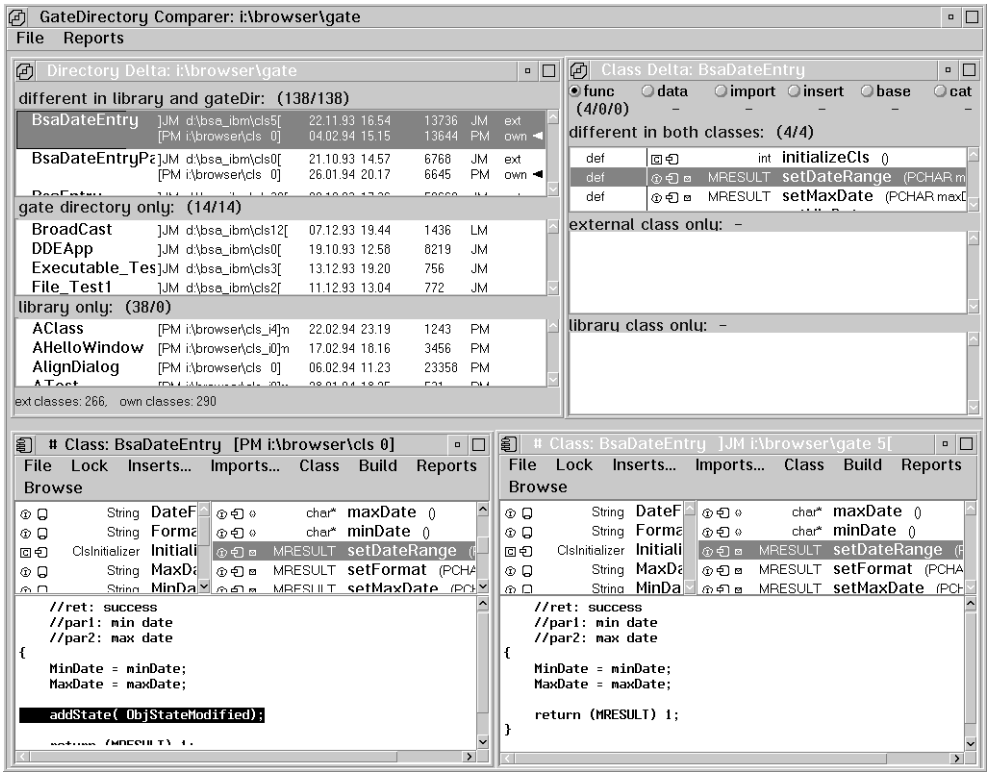
1. A *Class Archive* that is managed on a class and location basis. This means that every class in a specific location has its own class archive. The archive holds two types of files: class revisions and change-files. A class revision is a compressed copy of a complete class file. A change file is a compressed copy of changes containing source code that has been logged between two revisions.
2. *Group archives* contain compressed copies of different classes and are used mainly as a distribution mechanism (e.g. all classes for a particular application, in a particular location).

The **ArchiveBrowser** can be used to view the contents of all available class archives. It has two panes. The left pane shows symbolic names that have been attached to class revisions, e.g. **`<rel 1.1>`.** Selecting one of these names results in the display of only those class archives that contain classes with the selected attribute attached to them. These names are primarily used to identify class library releases, such as a snapshot of all class revisions at a particular date/time. The right pane shows all matching archive entries alphabetically sorted by class name. Several options are available via a set of controls in the upper portion of the window. Either all archived revisions of a class or only the most recent ones can be displayed. It is also possible to get the **`<nearest>`** archived revision of each class for a particular date. This is useful to get a ”best guess” state of the class library that was not a named release. Alternatively a **`<since>`** option allows to display all archives since a particular date.

A read-only **ClassEditor** on the selected archive entry can be opened to view the contents of the archived class by using the popup menu of the entry pane.



Compare and merge with a different class set

The **ArchiveViewer** can be used to take a closer look at the contents of a class archive. It consists of two panes. The upper pane shows all entries (files) contained in the archive.

Class revision entries are marked with a small symbol and a single revision number. Change-file entries are marked with a different bitmap and displayed with the range of changes (e.g. **<6-7>** meaning changes between and including revision number 6 and 7) that are covered by this file. The lower pane shows multiline comments either for the whole archive or for the selected archive entry. Existing entries can be loaded into a **ClassEditor** (for a revision entry) or to a **ChangeBrowser** (for a change-file entry) by double-clicking on them or by using the pane's popup menu. New entries can be added using the menubar (i.e. **<Archive current version>**, **<Archive current change-file>**) and a special archive dialog. This dialog is used to specify symbolic release names and also to enter multiline comments. After a change-file has been archived, it is emptied to prevent redundant archiving.

As for any library operation involving multiple classes, the **HierarchyViewer** can be used to update the class archives for a set of classes. Using this mechanism it is possible to specify the release name just once and perform all necessary archive updates in the background. If the current revision of a class is found to be already archived, only the release name is appended to the particular archive entry. No multiple archive entries are included for the same revision number of a class.

During application development it may be necessary to work on classes in parallel. The resulting developer efforts must then be merged into one consistent class library.

The picture shows the **DirectoryMerger** which is used to compare the current class library with the contents of a different directory holding class sources. It is the primary tool to compare and merge different class libraries. The **DirectoryMerger** has four child windows.

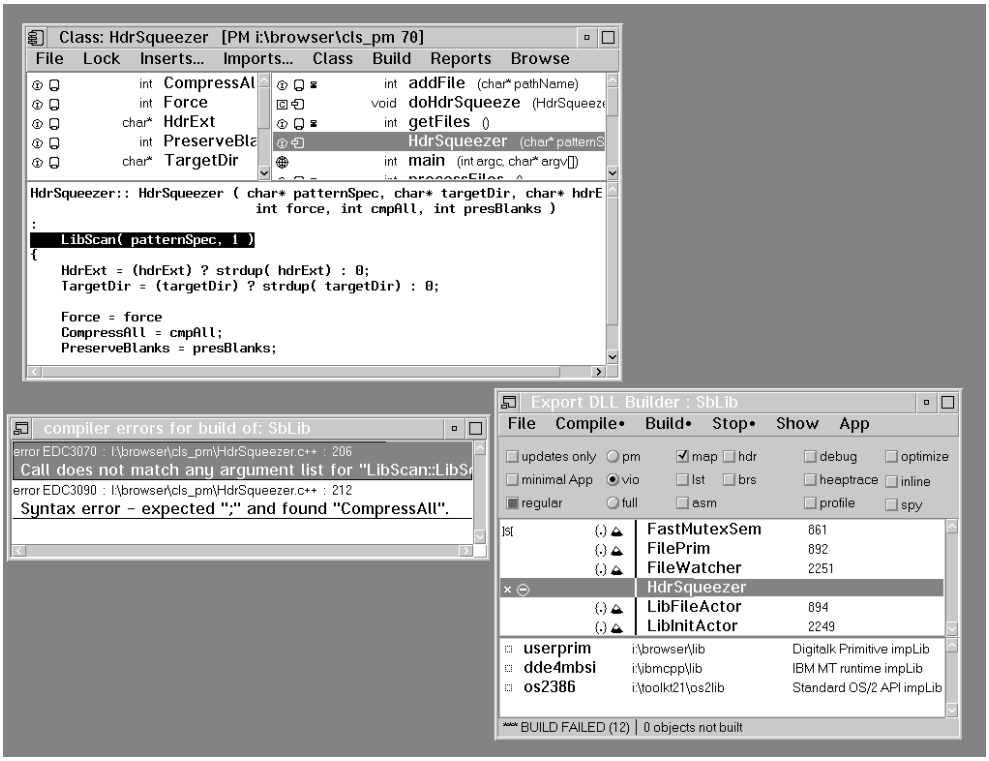
The **DirCompareView** is used to display similarities and/or differences between the two class libraries and consists of three panes. The upper pane lists all classes found in the library that differ from classes with same names found in the external directory. For each entry, the location, revision number and date is displayed for both the library and the external version of the class. The most recently changed version is marked with an asterisk. The middle pane lists all classes found only in the external directory. The lower pane lists classes that are present only in the current class library. Using popup menus, it is possible to copy a class or a group of classes from the external directory to a library location. If a class is updated from an external location, the own revision number is kept unchanged, and a history record, stating the source of the update, i.e. the external location and revision number, is added to the class file.

Selecting an entry in the upper pane (the pane containing differing classes) opens three additional views: a **ClassCompareView**, a **ClassEditor** on the library version of the class and a **ClassEditor** on the external version of the class. Selecting an entry in one of the lower panes opens only the particular **ClassEditor**.

The **ClassCompareView** has the same functionality as the **DirCompareView** with respect to two different versions of a particular class. It also has three panes used to display differing, external-only and library-only items of that class. The type of the item being displayed (e.g. function, data, import, etc.) is controlled with a set of buttons located in the upper portion of the **ClassCompareView** window. Like in the **LogBrowser**, the number of different/external/own entries is shown in braces underneath the corresponding button. Selecting an entry in one of the three panes positions both **ClassEditors** on the corresponding item (e.g. a function member); differences are marked by highlighting the corresponding source line (e.g. different function body lines). Again it is possible to update the own class with the selected item from the external class using the popup menu.

Build and Test *.OBJ, *.EXE and *.DLL

When changing classes you have to keep track of the changes and their fan-out. For example adding a virtual function to a base class requires all dependent classes to be recompiled; using template classes requires the instantiation of all used types exactly once in order to reduce memory consumption.



Automatically build applications or libraries

The tool used to manage all this, to compile classes into **`*.obj`** files and to link objects into executables (or dynamic link libraries) is the **ApplicationBuilder**. It represents an interactive substitute for the standard MAKE tool and automates all necessary compiler and linker calls that are transparent to the user.

The **ApplicationBuilder** window consists of two panes:

- ClassPane (upper half)
- LibPane (lower half)

The **ApplicationBuilder** can be launched from the **HierarchyViewer** or the **ClassEditor** and may be used with all classes that have appropriate attributes (**`<app>`** or **`<dll>`**). When opened, the **ApplicationBuilder** determines which classes are required for the application and performs several checks on them: each class is checked to see if it has an outdated object file, i.e. if the object file is older than the corresponding source file; compiler switches used to compile the existing object file are also compared against the current settings; you may even let ***CThrough*** determine the inheritance graph for a class and selectively recompile all depending classes. The classes are shown in the *ClassPane* with all classes that need recompilation being marked by a small checkmark (bitmap). Additional information may be shown: the attributes or switches of each object, e.g. compiled **`<opti-`**

mized>, **<inline>**, **<debug>,** are shown using small bitmaps. For classes whose object files have already been built, the size of the corresponding object file in bytes is shown behind the class name. The popup menu of the ClassPane can be used to launch several analysis and statistic tools. These tools can provide details about the users of a class, imports, layers, listings of corresponding ***.lst** and ***.asm** files, etc.

The *LibPane* shows a list of all ***.lib** files required for linking the application. This list is determined automatically from a library description entry in ***CThrough's*** configuration file that can be defined by the user. The user has complete control over the library list and can add other libraries and also remove existing ones using the popup menu of the pane.

The actual build operations, namely compile and link, are started using the menu bar options.

The **<Compile•>** option causes all entries in the ClassPane marked for either optional or forced update, to be recompiled according to the switches specified by the *ButtonPane* located at the top of the window. These switches affect compiler flags, such as **<debug>**, **<profile>**, **<optimize>** and **<inline>**. They also control the generation of additional information files, such as ***.map**, ***.lst**, ***.asm**, and the usage of compile and link options for several runtime analysis tools such as heaptrace, EXTRA and IPMD. The actual compiler switches, e.g. **</O+>** for **<optimize>**, can be set in a special configuration file.[1]

<Build•> starts the compiler just as the **<Compile•>** option does. However, after successful compilation it also starts the linker to bind the objects and the specified libraries. Linkage type depends on selected linker flags, such as **pm**, **vio**, **full**. Depending on the attributes of the project a ***.dll**, ***.lib** or ***.exe** file is created.

A build run started by **<Compile•>** or **<Build•>** may be aborted any time using the **<Stop•>** option of the bar menu.

When detecting compiler errors the **ApplicationBuilder** pops up a **MessageViewer** showing information about the sources of the error and the error itself. Each entry holds details about the file and the line number where the problem is located, and also the compiler generated error description. Error specific ***.INF** information can be shown using the popup menu.

To examine and correct an error, its corresponding entry can be dragged from the **MessageViewer** on a **ClassEditor** that is then automatically positioned at the class part that caused the error. In case no **ClassEditor** is opened you can launch one using the popup menu of the **MessageViewer**.

If an ***.exe** has been successfully built, it can be started via the **<App>** submenu. If special build options were specified, e.g. **<debug>** or **<profile>**, the appropriate tool, i.e. IPMD or IXTRA resp., is started automatically taking the application name as parameter. Optional parameters for the application may be specified using a Dialog.

The **<Show>** submenu is used to list and analyze additional information files, e.g. mapfile, heaptrace, etc. For a description of some of these, see the following section.

As mentioned above, several additional information files may be generated by the **ApplicationBuilder**. The picture above shows tools used to analyze two of them - the **AsmViewer** and the **MapfileViewer**.

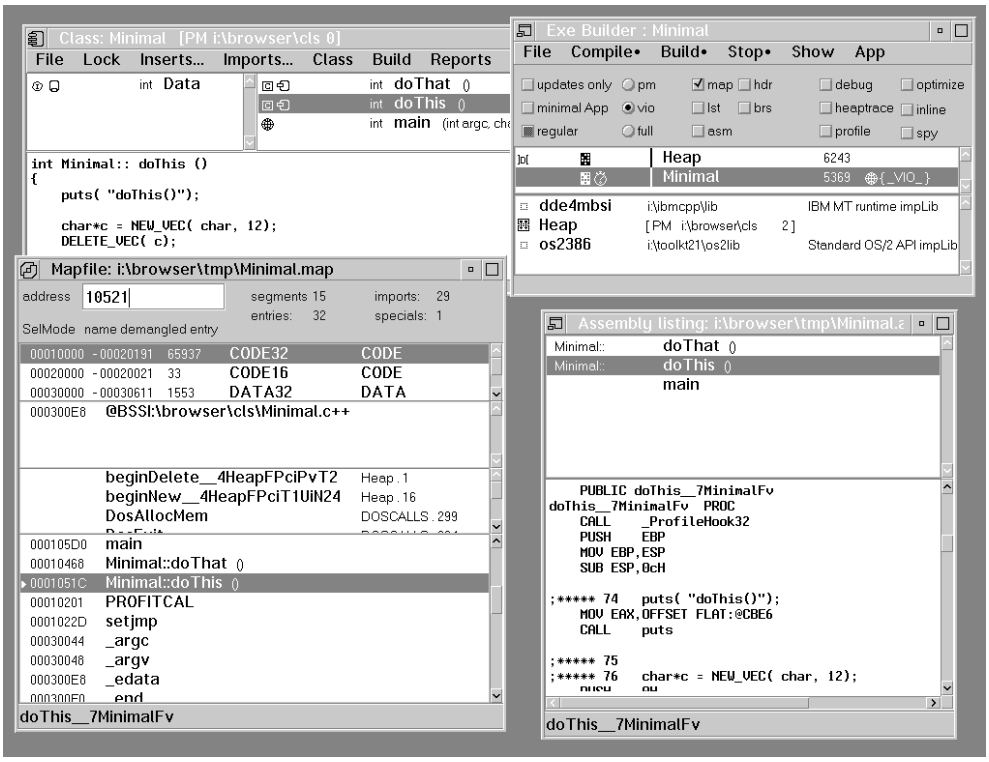
Compiling an ***.obj** with the **<asm>** attribute set creates an optional ***.asm** file. This compiler-generated assembly file can be viewed using the **AsmViewer** which consists of two panes. The upper pane lists all public entries found in the ***.asm** file. Selecting an

---

1 Available switches depend on the compiler used. Information shown is available only with the IBM C/Set++™ compiler and Toolkit.

entry causes the corresponding assembler code to be displayed in the lower pane. This tool is mainly used for optimization purposes.

The **MapfileViewer** analyzes a compressed version of the **`*.map`** information created for applications built with the **`<map>`** attribute set. From top to bottom, it shows a list of the application segments, the exported and the imported functions. It also lists all public functions of the application which can be sorted either by name or by address. Selecting a segment within the pane causes the function list to show all functions located in the selected segment. Selecting a function in turn marks the segment in which this function is located.



More views on compiled classes (mapfile and disassembly view)

This tool is useful to detect locations of runtime errors, such as general protection faults, in applications built without debug information. For large applications, it is often not feasible to use debug information for every incorporated class because of the implications on the compilation speed and size of the executable.

Entering the address of an exception into the Entryfield (top left) causes the **MapfileViewer** to select the function in which the error has occurred. This information is often sufficient to locate the error in the source code. Where the information is not sufficient, it provides at least a good starting point for deciding which classes have to be recompiled using the **`<debug>`** setting.

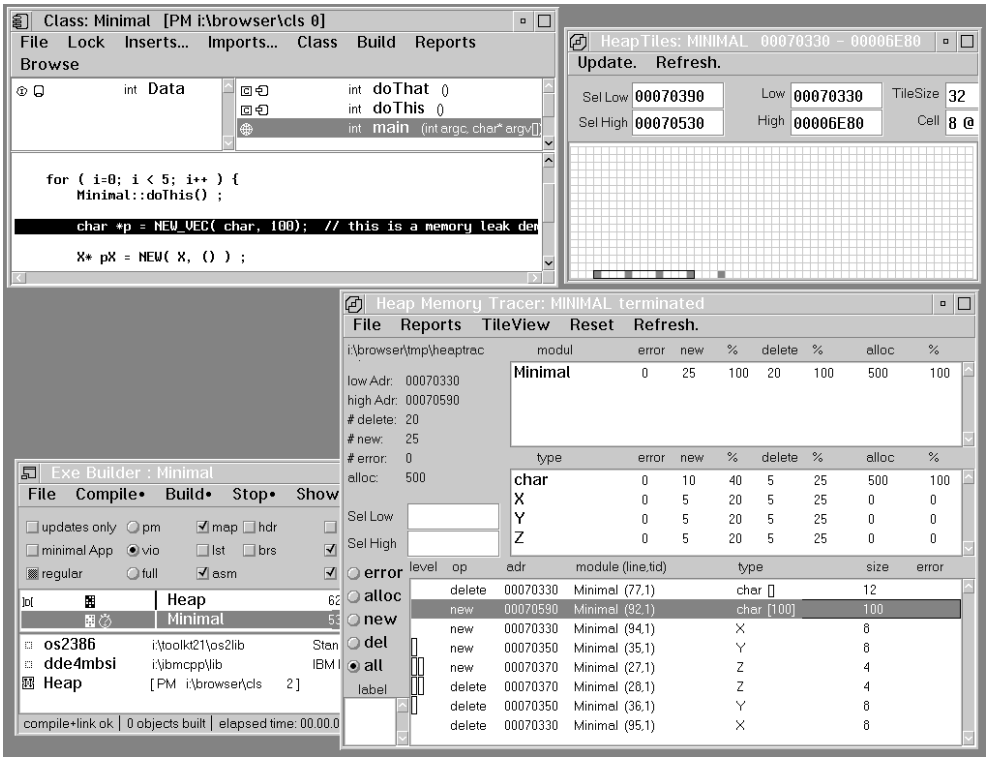
Heap problems are some of the most intricate ones C++ can present to the developer. A production quality application or class library should be "heap-checked" in order to prevent, for example memory leaks.

CThrough comes with support classes to ease this type of debugging. An application that uses the supplied heap classes and that was built using the **`<heaptrace>`** attribute logs all heap operations, i.e. **`new`** and **`delete`**, on a special file. The contents of this file may be

subsequently analyzed by the **HeapTracer**. This tool is primarily used to detect memory leaks. It consists of three panes:

- ModulePane (upper)
- TypePane (middle)
- OperationPane (lower)

The *ModulePane* shows statistic information about classes that have issued heap operations, e.g. number of **`new`** and **`delete`** operations, allocated bytes, etc.



Additional test tools (*HeapTracer* with *MemTileViewer*)

A type-based view is provided by the *TypePane*. This pane displays the number of allocated instances of a specific built-in or user-defined type, the number of instances freed, and the total number of bytes allocated. Each of these figures is also accompanied by its percentage representation.

Perhaps the most important pane is the *OperationPane*. All heap operations ever performed during the lifetime of the application are shown in the order of their execution. Each entry consists of the operation type, i.e. **`new`** or **`delete`**, its address space, the module or class which caused that operation, the storage type and the allocated memory size. The selected entry can be dragged onto an open **ClassEditor** that is automatically positioned on the code that performed the heap operation. The OperationPane contents may be filtered using the RadioButton selection to the left of it. For example, selecting **`<alloc>`** shows only active, i.e. not freed, allocation entries. Depending on the state of the application, such entries might be memory leaks. The **HeapTracer** may be used in parallel with the execution of an application (communicating with the application via pipes) or after the application has been terminated (analysing a log file).

Heapspace fragmentation can be viewed using the **MemTileViewer** which can be launched from the **HeapTracer**. This window displays the cluster based memory allocation for a specified memory range. It is useful for the analysis of the heap behavior.

Networking

CThrough can be used in a networked environment. The usual network configuration has a set of server directories that hold the classes available to all programmers. In addition, each programmer has one or more local directories for private classes or classes that are not yet ready for public use. The settings for the server directories and the local directories can be modified using one of ***CThrough's*** configuration files..

Classes can also be locked on a class file basis. All users, except the one that owns the lock, have read-only access to a locked class. Lock information, such as the name of the lock owner ("who locked it"), date/time of the lock process ("when was it locked"), and an arbitrary lock reason specified by the lock owner ("for what purpose was it locked"), may be displayed when accessing a locked class. The lock may be released only by the lock owner.

Requirements

In order to install ***CThrough*** you need

- OS/2 2.1 or higher
- IBM C/Set++ and Toolkit
- 6 MByte of disk space (for the binaries)
- 1024 x 768 display (or above) recommended, although VGA may also be used
- 486 processor recommended
- >= 8 MB recommended

Although this depends on personal preference, it is strongly felt that an appropriate display system is a top priority item on the requirement list. Much of ***CThrough's*** productivity momentum depends on being able to deal with a lot of information simultaneously. This is only possible using display formats larger than VGA.

Since many ***CThrough*** operations make heavy use of the file system, a fast harddisk is also highly recommended.

History

CThrough is part of a software development environment based on C++ called ***BSA*** (**B**ISS **S**oftware **A**rchitecture). ***BSA*** consists of an underlying technology, ***CThrough***, more than four hundred C++ classes, and a graphical tool used to manipulate and link objects. ***BSA*** classes cover almost all OS/2 programming tasks, providing support for GUI design, process/thread/file/pipe handling, database access, generic containers, help management, etc. The library has matured in more than four years being heavily used in commercial projects by several large European companies. All classes have been developed using ***CThrough***.

CThrough has also been used to build itself.

Availability

CThrough currently supports the IBM C/Set++™ compiler and Toolkit. Version for Borland, WATCOM and MetaWare compilers will be available soon.

For ordering information please contact:

BISS GmbH
Chaukenweg 12
D-26388 Wilhelmshaven
Germany

Phone: +49 4423 9289-0
Fax: +49 4423 9289-99
CIS: 100031,1733 (CompuServe ID)
Internet: 100031.1733@CompuServe.com

