LAUTERBACH
DEVELOPMENT TOOLS

Application Note C++ Debugging

MANUAL

Application Note C++ Debugging

Application Note C++ Debugging

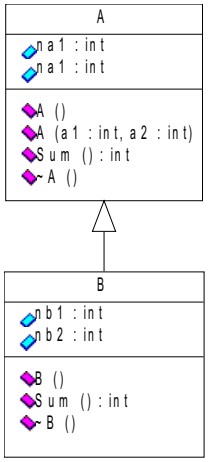
Version 06-Jun-2024

This application note describes dedicated TRACE32 PowerView commands and approaches that can be helpful when C++ applications are analyzed.

Sample Code used by This Application Note

Further chapters will contain pictures, commands and source code that are based to a small sample. This sample code implements two classes A and B, where B is derived from A.

UML Class Diagram



Class A

```
class A {
  public:
    int na1;
    int na2;
    A(){na1 = 1; na2 = 2;}
    A(int a1, int a2){na1 = a1; na2 = a2;}
    virtual int Sum() {
      return na1 + na2;
    }
    virtual ~A(){;}
};
```

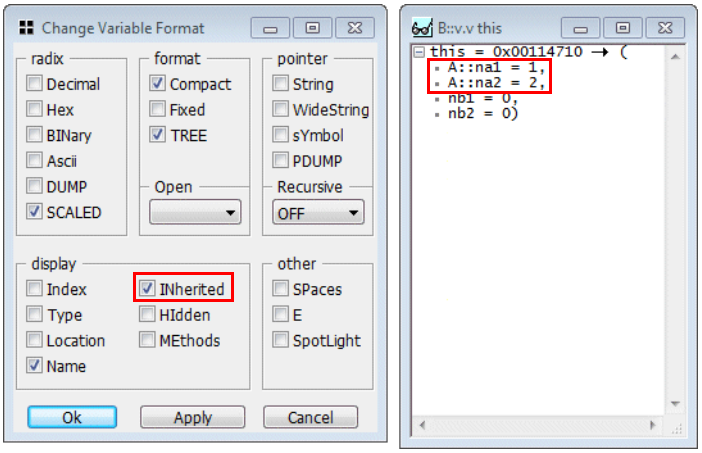
Class B

```
class B: public A {
  public:
    int nb1;
    int nb2;
    B(){nb1 = 3; nb2 = 4;}
    virtual int Sum() {
      return A::Sum()+nb1+nb2;
    }
    virtual ~B(){;}
};
```

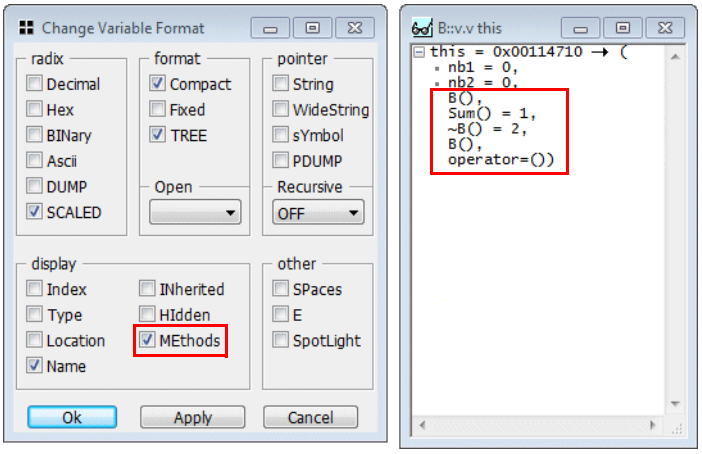
Display Options Dedicated to Objects

The **Var.Watch** or **Var.View** command can show the content of instances of structures or classes. The command **SETUP.Var** or the context menu entry **Format** will open a dialog with various display settings.

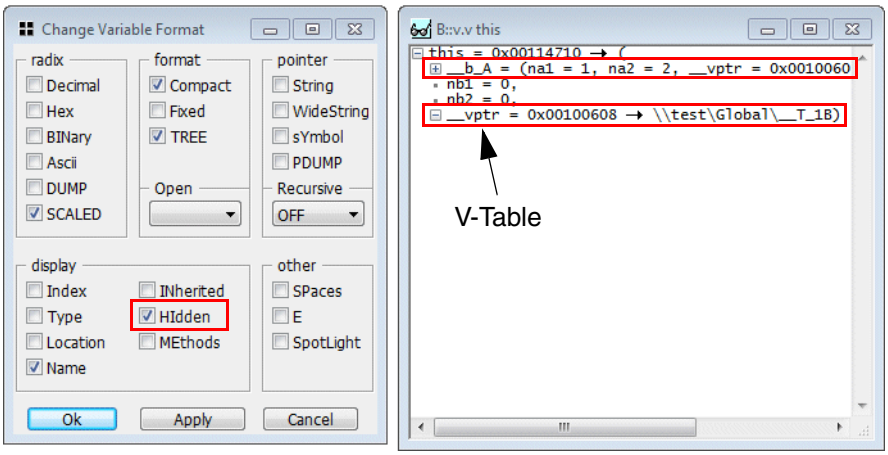
The **Inherited** check box controls whether anything is displayed from the base classes.



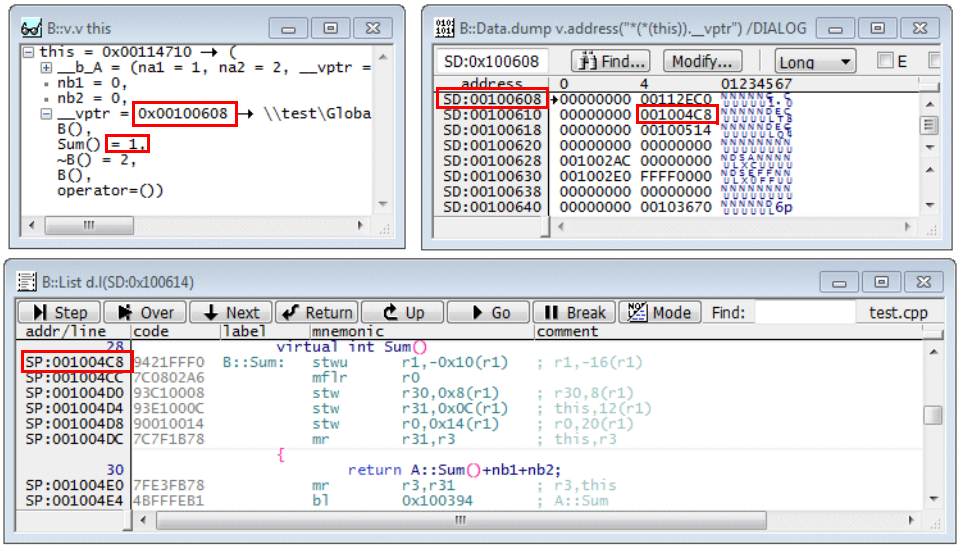
To show all known methods of the class, the **Methods** check box must be turned on. The integer number after a method indicates the index of that method within the Virtual Function Table.



The **Hidden** check box controls whether artificially created symbols are visible.



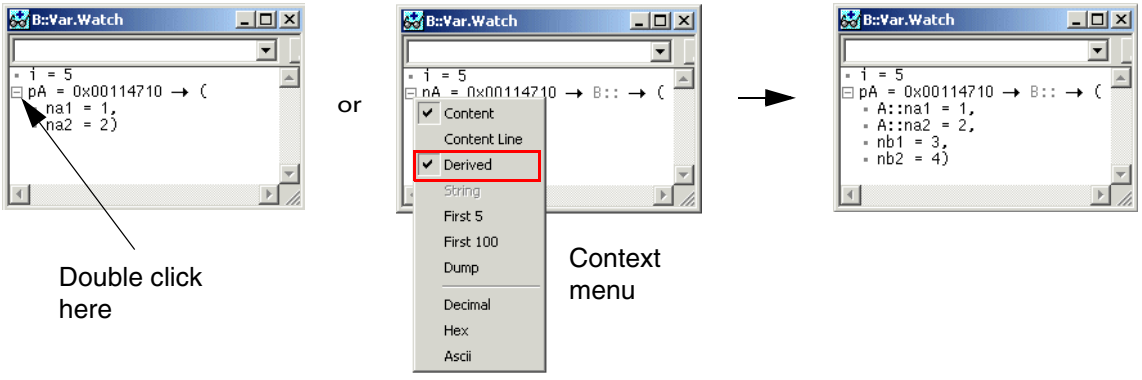
By the context menu commands “Indirect Dump” and “Indirect List” it is possible to navigate to the virtual function in order to check if the values match the debug information.



Most Derived Class

Generic algorithms use the polymorphic nature of classes they work with. These algorithms will handle pointers to objects that fulfill an interface or are derived from a certain parent class. The Most Derived Class is the class type that was used to create a particular instance. By default, the **Var.Watch** window will treat these pointers with the type that is used in the current context. The **Derived** option can be turned on for a certain entry of the **Var.Watch** window by double-clicking the tree node icon or using the context menu.

```
void TestMostDerivedClass() {
  B* pB = new B();
  A* pA = pB;
  //view content of pA here...
  pA = NULL;
  delete pB;
};
```



To automate the Most Derived Class feature, the **[]** operator can be used in PRACTICE:

```
; set nb1 to 1
Var.Set (*[pA]).nb1 = 1

; display most derived class of pA
Var.View *[pA]
```

Lifetime of the “this” Pointer

When a none-static method of a class is called, the object instance is transferred by the call, too. Within the method a symbol named **this** will be valid once the stack frame start code has finished. When a method is implemented into a single line of source code, it is necessary to do assembler steps until **this** is alive before any other class member can be observed correctly. The **this** symbol can be used to show the members of the instance:

```
Var.View this
```

Class Conversions

The **()** operator can be used to cast an object to another class.

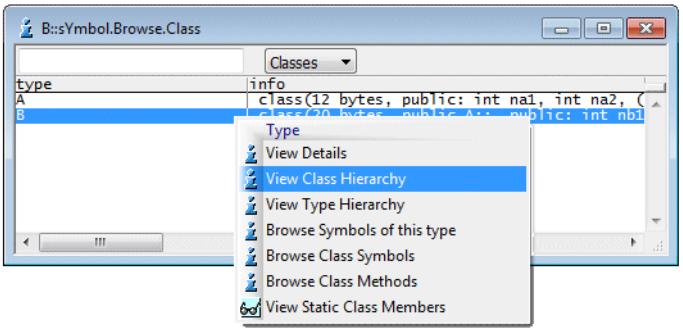
```
;cast this pointer to class B and display it
Var.View (B*)this

;cast object at address 0x00114710 to class B and display it
Var.View (B*)0x00114710
```

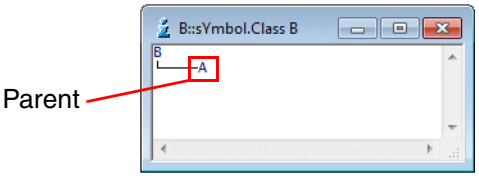
Gathering Information of Classes

There are two ways to gather information about classes: the class browser and the symbol browser.

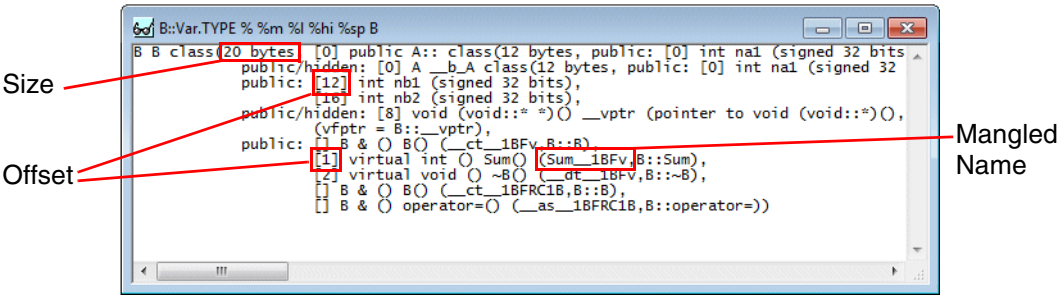
The class browser lists all known classes in the type column. Righ-clicking a class opens a context menu.



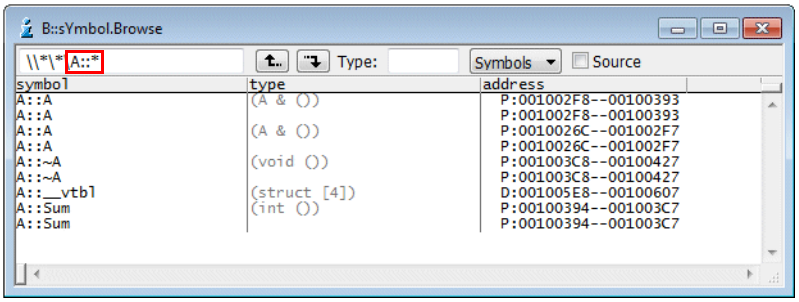
The context menu, for example, provides access to the class hierarchy diagram (Select the **View Class Hierarchy** option).



Via the context menu, it is also possible to view the details of a class (Select the **View Details** option).



The symbol browser can show things that are dedicated to one class by using its capabilities to filter names.



Demangling

In contrast to C, multiple functions can have the same name by using the overloading technique. The functions must then differ in their parameter signatures. In C++ object files, therefore an exported symbol is encoded using function name and parameter signature into one mangled name. The mangling can be different for different compilers. TRACE32 demangles the names controlled by **sYmbol.DEMangle**. The first parameter set whether the Demangler is turned on. The second parameter controls if the parameter signature is concatenated to the symbol name.

		Symbol of A::A(int,int)
sYmbol.DEMangle OFF	Demangler switched off	__ct__1AFiT1
sYmbol.DEMangle ON ON	Demangler switched on	A::A(int,int)
sYmbol.DEMangle ON OFF	Demangler switched on, but does not concatenate parameters	A::A

Ambiguous Symbols

Due to the overloading technique of C++, the debug information contains ambiguous short symbol names. Resolving these symbols will result in an error message:

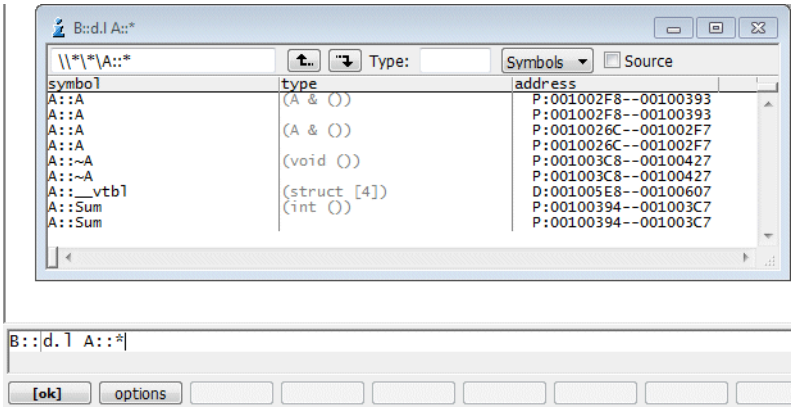
This behavior can be controlled by the **sYmbol.MATCH** setting:

sYmbol.MATCH Exact	Release error message when ambiguous symbol is resolved
sYmbol.MATCH Best	Take the first matching symbol
sYmbol.MATCH Choose	Open a symbol browser to select the right symbol

C++ symbols are often very long. Therefore use the **TAB** key to select and automatically complete the input of a symbol.

Symbols containing brackets must be enclosed in apostrophes to distinguish them from other parser functions that use brackets.

The symbol browser opens and the symbol matching the passed pattern can be chosen when an input symbol contains an asterisk character.



ELF Loader Options

The **Data.LOAD.Elf** command has additional parameters to tune the loader for formats or circumstances it cannot detect correctly.

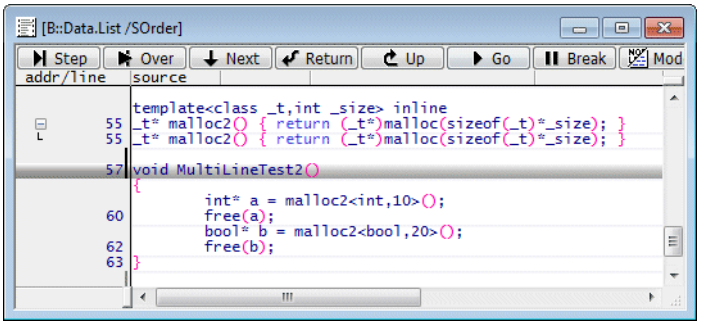
Option	Behavior
/CPP	Used for ELF/STABS format if symbols are loaded correctly.
/CFRONT	Load additional symbol information for CFront based compilers that translate C++ code to C code.
/NoInclude	Ignore debug information that is related to code in header files.
/SingleLine	One source line is assigned to multiple target code parts. Therefore the source lines are multiplied, too. In the Source Order view of the **Data.List** window, these lines are merged behind a “+” symbol. When this option is passed, the debugger doesn’t multiply the source lines. The source line is assigned to the first code part.
/SingleLineAdjacent	The option is similar to /SingleLine, but only takes effect if the code parts are concatenated.

Target Order / Source Order

Due to the usage of inline code, one source line can match multiple code sections. When a memory section is displayed, it can be source order or target order.

Source Order

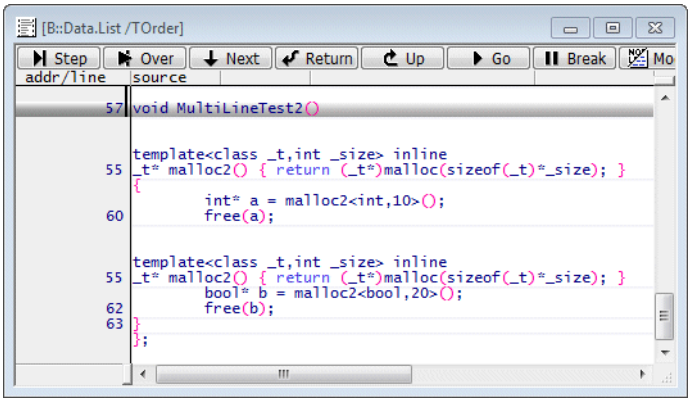
Source order mixes the code sections to the order inside the source file:



Target Order mode is active for HLL debugging by default. Both calls of the inline function “malloc2” will create one code piece each. Due to source order these code pieces are displayed above the “MultiLineTest2” function. When the function is executed by **Step.Over** commands, the windows selected source line will jump to the matching code lines. This can be confusing some times.

Target Order

Target Order displays the code in the order of ascending addresses of the code behind:



In this case it is more obvious that the compile has inlined the code from the “malloc2” template. The execution of the routine by **Step.Over** will work without any confusing jump to other source lines. Target Order display is the default in Mixed mode debugging.

sYmbol.ForEach	Sets breakpoints to every function and symbol.

```
sYmbol.ForEach "Break.Set *" A::*
```

Sometimes it is necessary to break at a method for a certain object. This can be achieved by a breakpoint containing a condition that checks the passed **this** pointer of the instance. For example, the **this** pointer is passed by register **R3** and the object has the address **0x0114710** the breakpoint for the function **A::Sum** would be set by:

```
Break.Set A::Sum /CONDition Register(R3)==0114710
```