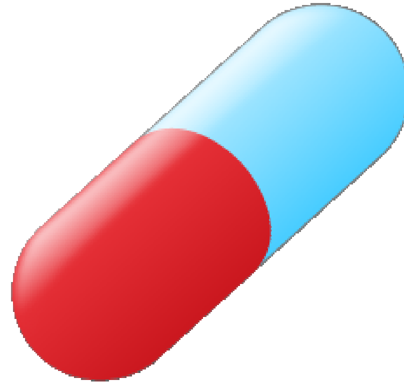
Vitamin-C

For Metastock®

For Traders using
Metastock® Ver 7,8,9 and 10.



Advancing the Art of Trading with Science.

User Guide for Vitamin-C for Metastock®

Version 1.0.1 Build 2 - Final Release
Last Update 13 September 2009

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Contents

CONTENTS 2

DISCLAIMER 6

COPYRIGHT AND LICENSING AGREEMENT 7

Trademarks 8

ACKNOWLEDGEMENTS 9

VITAMIN-C FEATURE SUMMARY 10

INTRODUCTION 12

The problem 13
Why the MetaStock Formula Language has limitations. 13
Trying to code simple systems using the MSFL 15
How MetaStock Communicates with Vitamin-C 16
Calling script files from your MetaStock code 16
Anatomy of a Vitamin-C function call 16
- Example of using CallScript with No User Array Arguments 18
- Examples of using CallScript with 1 User Array Arguments 18
- Example of using CallScript with 3 User Array Arguments 18

Your First Step 19
Coding simple systems using the 'Vitamin-C' language 19
Exercise 23
A Closer look 23
What happens if you make a mistake? 25
Detecting coding errors before you run the code-script. 25
Installing the Free Compiler on your system 25
Running A Syntax Check 25
Correcting coding errors at runtime 27
Auto Save feature. 28
A Brief Introduction to C++ 29
Predefined Variables and Functions 30
Predefined Variables 30
Vitamin-C specific functions 30
Predefined Array Variables 31
Your Second Step 32
Exercise 34
Your Third Step. 35
Corollary 40
Exercise 42
Your Fourth Step 43
Getting a little bit more serious ! 43
Coding a Time Stop 43
Creating a MetaStock Time Stop Expert 47
Building a simple Profit Stop Indicator Using Vitamin-C 50
Spicing up the Profit Stop. 54
Handling the Short side of the Market 55
Building a Trailing Stop function with Vitamin-C 57

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The TradeSim Trailing Stop Function Description **57**
Trailing Stop Algorithm on the Long Side 58
Trailing Stop Algorithm on the Short Side 59
Running the Vitamin-C Trailing Stop code **61**
Exercise 62
Implementing Moving Averages with Vitamin-C **64**
The Simple Moving Average (SMA) **65**
Improving the SMA 66
The Exponential Moving Average **68**
Porting Code from other Charting Platforms to Vitamin-C **72**
Example 1: Adaptive Moving Average (AMA) **72**
Example 2: The Guppy Count back Line Trailing Stop Indicator **75**
Multi Dimensional basic arrays in Vitamin-C **78**
Basic Array Declarations **78**
Single Dimensional Arrays 78
❑ Examples 78
Two Dimensional Arrays 78
❑ Examples 78
Three Dimensional Arrays 78
❑ Examples 79
N-Dimension Arrays 79
❑ Examples 79
Using basic arrays in Vitamin-C **79**
2-D basic array example 79
3-D basic array example 80
Using the Standard 'C' library from Vitamin-C **83**
Transcendental Example **83**
File IO example **84**
Using Vitamin-C with TradeSim **88**
Method 1: Generating Entry and Exit Triggers using Vitamin-C **88**
Method 2: Creating a Text Trade Database. **92**
Using Vitamin-C with BullCharts **95**

ADVANCED TOPICS **99**

Using the Standard Template Library **99**
❑ Example 1 99

APPENDIX A **101**

A Brief Introduction to C/C++ **101**
Adding Comments to your C-script **101**
What Is a Variable? **101**
Storing Data in Memory 102
Allocating Storage for Variables 102
Size of Variables 102
Signed and Unsigned Integers 102
Volatile and Non-Volatile storage 103
Keywords **103**
Fundamental Variable Types **103**
Storage Classes **104**
Declaring Variables **104**
Case Sensitivity 104
Typical Program Forms **105**
Statements **105**
Anatomy of a function. **106**
Program Flow Control **106**
Relational Expressions 106
❑ A common pitfall 107
The if Statement 108

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The if…else Statement 108
The if…else…if…else Statement 109
- Example 1 110
- Example 2 111
The ?: Operator 111
The switch Statement 111
- Example 1 112
- Example 2 112
Logical Expressions 112
The Logical OR Operator || 112
The Logical AND Operator && 113
The Logical NOT Operator ! 113
Loops 113
The for-loop 113
- Example 115
The while loop 115
- Example 116
The do…while loop 116
- Example 117
The break statement 117
- Example 118
The continue statement 118

APPENDIX B 119

General Forms of CallScript 119
CallScript (0 user array arguments) 119
CallScript1 (1 user array arguments) 119
CallScript2 (2 user array arguments) 119
CallScript3 (3 user array arguments) 119
CallScript4 (4 user array arguments) 120
CallScript5 (5 user array arguments) 120
CallScript6 (6 user array arguments) 120
CallScript7 (7 user array arguments) 120

APPENDIX C 122

The Array Class Type 122
Available Operators 122
Array Member Functions 123
Array Friend Functions 124

APPENDIX D 125

Installing the Free Borland C++ Compiler on your system 125

APPENDIX E 128

MetaStock Sample Indicator code 128
Adaptive Moving Average 128
Exponential Moving Average 128
Simple Moving Average 128
Guppy CBL 129
2D-Array 129
3D-Array 129
STL string 129
Sinewave 129
If() 129
Simple Moving Average 130
Trailing Stop 130
Time Stop 130
Profit Stop 131

REFERENCE LITERATURE 132

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

General Reading and References 132
- **C/C++ Programming and Language Reference** 132
- **Online C/C++ References** 132
- **Cited TradeSim Documents** 132
- **General References on Trading** 132

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Disclaimer

In no event shall Compuvision Australia or its suppliers be liable for any damage either direct or indirect, including, without limitation, damages for loss of business profits, business interruption, loss or business information or other losses arising out of the use of or inability to use the software.

The results obtained from using this software are not indicative of, and have no bearing on, any results, which may be attained in actual trading. Results of past performance are no guarantee of future performance. It should not be assumed that you would experience results comparable to that reflected by the results from this software. No assurance is given that you will not incur substantial losses, nor shall Compuvision Australia Pty Ltd be held liable if losses are incurred.

Compuvision Australia Pty Ltd is not a licensed investment advisor and so the information and results obtained by using this software is for educational purposes and of the nature of a general comment and neither purports nor intends to be, specific trading advice. The information obtained from using this software should not be considered as an offer or enticement to buy, sell or trade and is given without regard to any particular person's investment objectives, financial situation and particular needs. This software is not designed to replace your Licensed Financial Consultant or your Stockbroker. You should seek appropriate advice from your broker, or licensed investment advisor, before taking any action.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Copyright and Licensing Agreement

Vitamin-C for Metastock is Copyright© 2000-2009 by Compuvision Australia Pty Ltd.

IMPORTANT - READ CAREFULLY

This license statement and limited warranty constitutes a legal agreement ("License Agreement") between you ("Licensee", either as an individual or a single entity) and Compuvision Australia Pty Ltd ("Vendor"), for the software product Vitamin-C for Metastock® ("Software") of which Compuvision Australia Pty Ltd is the copyright holder.

BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE, YOU AGREE TO BE BOUND BY ALL OF THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT.

Upon your acceptance of the terms and conditions of the License Agreement, Compuvision Australia Pty Ltd grants you the right to use the Software in the manner provided below.

If you do not accept the terms and conditions of the License Agreement, you are to promptly delete each and any copy of the Software from your computer(s).

This license agreement only applies to the software product "Vitamin-C for Metastock" and not to any other product even if that product is similar to Vitamin-C for Metastock and has a similar name.

The Vendor reserves the right to license the same Software to other individuals or entities under a different license agreement.

After accepting this license agreement, the Licensee is permitted to use the Software under the terms of this agreement.

Under this license agreement, the Software can only be used by those persons or entities that have purchased a license key. Payment information is available at http://www.compuvision.com.au/PurchaseOnline.htm.

The Software is provided "as is". In no event shall Compuvision Australia Pty Ltd be liable for any consequential, special, incidental or indirect damages of any kind arising out of the delivery, performance or use of this Software, to the maximum extent permitted by applicable law. While the Software has been developed with great care, it is not possible to warrant that the Software is error free. The Software is not designed nor intended to be used in any activity that may cause personal injury, death or any other severe damage or loss.

When errors are found in the Software, the Vendor will release a new version of the Software that may no longer contains those errors a reasonable amount of time after the Vendor is given an accurate description of those errors. Which amount of time is reasonable will depend on the complexity and severity of the errors. The Vendor will mention the release at http://www.compuvision.com.au, at the Vendor's option, directly contact the Licensee to announce the new release. The Licensee can then, at their option, upgrade to the latest version or to continue to use the older version the Licensee already has. In the latter case, the Licensee will no longer be entitled to technical support until the Licensee has upgraded to the latest version.

The Vendor reserves the right to charge an upgrade fee in the case of major new enhancements or additions to the Software. This major new version will then start a new version line, which will use version numbers clearly distinguishable from the old version line. The Licensee has no obligation to upgrade to the new version line.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

You must not attempt to reverse compile, modify, translate or disassemble the Software in whole or in part. You must not run the Software under a debugger or similar tool allowing you to inspect the inner workings of the Software.

The Software remains the exclusive property of the Vendor. Any Licensee, which fully complies with the terms in this license agreement, may use it according to the terms of this license agreement. You must not give copies of the Software or your license key to other persons or entities. If you have received a download password or an URL with an embedded password for downloading the Software, you must keep this password secret. You must also take reasonable steps to prevent any third party from copying the software from one of your machines without your permission.

The Vendor reserves the right to revoke your license if you violate any or all of the terms of this license agreement, without prior notice.

Trademarks

- TradeSim® is a registered trademark of Compuvision Australia Pty Ltd.
- Metastock® is a registered trademark of Equis International.
- Microsoft Windows® is a registered trademark of Microsoft Corporation.
- Microsoft Excel® is a registered trademark of Microsoft Corporation.
- Word for Windows ® is a registered trademark of Microsoft Corporation.
- TradeStation® is a registered trademark of Omega Research Corporation.
- Cint and associated tools are owned by Agilent Technologies Japan Company.
- CINT is developed by Masaharu Goto, who works for Agilent Technologies, Philippe Canal and Paul Russo at Fermilab, and Leandro Franco, Diego Marcos, and Axel Naumann from CERN.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Acknowledgements

Vitamin-C for Metastock® is adapted from CINT which is a C/C++ interpreter aimed at processing C/C++ scripts. Scripts are programs performing specific tasks. Generally execution time is not critical, but rapid development is. Using an interpreter the compile and link cycle is dramatically reduced facilitating rapid development. CINT makes C/C++ programming enjoyable even for part-time programmers.

CINT is written in C++ itself (slightly less than 400,000 lines of code). It is used in production by several companies in the banking, integrated devices, and even gaming environment, and of course by ROOT, making it the default interpreter for a large number of high energy physicists all over the world.

CINT covers most of ANSI C and ISO C++. Support for K&R-C, ANSI-C, ANSI-C++
CINT has 80-90% coverage on K&R-C, ANSI-C and C++ language constructs. (Multiple inheritance, virtual function, function overloading, operator overloading, default parameter, template, etc..) Cint is solid enough to interpret its own source code. CINT is not aimed to be a 100% ANSI/ISO compliant C++ language processor. It rather is a portable script language environment which is close enough to the standard C++.

CINT is developed by Masaharu Goto, who works for Agilent Technologies, Philippe Canal and Paul Russo at Fermilab, and Leandro Franco, Diego Marcos, and Axel Naumann from CERN.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Vitamin-C Feature Summary

- ✓ Implement functions not provided with MetaStock and/or which require the MetaStock Developers Kit.
- ✓ Perform complex calculations on price and volume data that can't be performed using the MetaStock Formula Language (MSFL).
- ✓ Provide multiple functions in a single script.
- ✓ Create functions that can be used by Custom Indicators, Systems Tests, Explorations and Experts.
- ✓ Distribute your Vitamin-C scripts to other users who also have Vitamin-C. Vitamin-C scripts are just standard text files that can be edited using any text editor or the fully featured context sensitive editor built into the Vitamin-C Integrated Development Environment (IDE).
- ✓ Enhance and extend the MetaStock Formula Language using structured programming constructs such as looping, for, while, do-while, break, continue, switch-case, pointers, structure and class objects, function calls etc.
- ✓ Simple Interface to MetaStock Formula Language (MSFL) and MetaStock Interface.
- ✓ No need to migrate to a new charting package. Re use your existing MetaStock code and enhance it using industry standard C/C++ language constructs.
- ✓ No need for cryptic MSFL code such as latches, to code simple functions such as protective stops, profit stops, time stops, trailing stops, pattern matching and recognition, wave count etc. Code it the way it makes sense !!
- ✓ Full programming access to most ANSI-C and C++ features although in most cases the simple C language constructs will suffice.
- ✓ C/C++ code allows the ability to easily code and reference a particular value(s) in a MetaStock data array for the purpose of writing stop functions in a straightforward way.
- ✓ Advanced Array class allows encapsulation of existing MetaStock price and data arrays for an intuitive and streamlined program interface. Both array indexing and array manipulation are available to the programmer and can be used independently or together to get the most flexibility.
- ✓ Hand optimized machine code, which takes advantage of the on-chip data caches as well as the CPU and Numeric Processor instruction set for super fast array processing ! In fact much faster than MetaStock !!
- ✓ Full access to ANSI compatible C library. If you need file IO operations then no problems; Vitamin-C has it all and more and there is no need to worry about which include files to use ! Access to the libraries are transparent to the user !
- ✓ Streamlined and transparent interface to MetaStock with full-featured Integrated Development Environment (IDE) and unlimited context sensitive editor, which can be resized and is not limited in any way.
- ✓ Write your own plug-in code without the hassle and added expense of using the MetaStock Developers Kit (MDK). There is no need for the MDK and no need for a qualification process in order to purchase or use Vitamin-C !!
- ✓ Because Vitamin-C script is fully interpreted there is no need to learn the idiosyncrasies of C compilers and linkers along with the complex interface structure used in the MDK.
- ✓ Run unlimited amounts of code.
- ✓ Full portability guaranteed. Unlike pseudo 'C' like script code used in other charting packages, the industry standard C code used in Vitamin-C allows code to be re-compiled later on for further performance and speed enhancements if need be.
- ✓ Modular programming interface allows libraries of functions to be easily compartmentalized and catalogued.
- ✓ Structured programming in C and C++ forces sensible programming approach.
- ✓ No need to deal with complex external formula library interface in the MDK, which is only really usable by experienced and seasoned programmers.
- ✓ Save time and money ! There is no need to change to another charting package and spend time and money learning another formula language. Use your existing investment in time building existing MSFL code and enhance it using Vitamin-C. Vitamin-C brings MetaStock into the 21st Century !

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

- ✓ Vitamin-C will form the de-facto standard for programming amongst other MetaStock users so it will be possible to easily share Vitamin-C scripts amongst your colleagues. Even if your forte is not programming you can still use scripts written by others just as you currently do with other MetaStock code.
- ✓ 12 months free updates from date of purchase !

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Introduction

I would love a dollar for every time someone asked me how to write some simple MetaStock code to reference a particular price at a particular bar. This information is usually needed for the purposes of building a simple protective or profit stop trading system. In the end this functionality would usually be hard coded into the TradeSim formula library that we provide with TradeSim. This had to be done at the lower level using the MetaStock Developers Kit (MDK) using the C language. Having this functionality built into the TradeSim formula library plug-in, allowed people to easily back test their systems using TradeSim but made it difficult to use this functionality outside of the TradeSim environment.

What was needed was the ability to easily code these things in a user-friendly environment without requiring a University computer science degree and years of experience. This formed the basis of the need for developing a new product, which would sit beside MetaStock and would allow people to easily code external functions that traditionally would require the MDK. It would also transcend the limitations imposed by the MetaStock Formula Language(MSFL) without resorting to the obfuscation and complexities of the MetaStock Developer's Kit ! The result is Vitamin-C for MetaStock ! I hope you have as much fun with it as I have in developing it. It’s been a long time coming but it is finally here.

Please carefully read through this User Guide and tryout the many examples that we have provided. Hopefully by the end of it you will be able to code your own systems without even touching the MDK. At the end of the User Guide you will find a brief introduction to C/C++ as well as some good references on the C/C++ language, which you will find useful if you have never had any experience programming in this great language before.

Throughout the manual you will notice that I refer to C and C++ as the one indivisible language. Historically the C language standard was established first and then later C++ built on C with lots of brand new concepts including object orientated programming. In essence C should be looked on as a subset of C++. It is often stated that it is more beneficial for first time programmers wanting to learn C++ not to have any prior experience with C but if you have then don’t despair as I had to and I ended up writing TradeSim with it ☺

Regards,
David Samborsky
BE Comms Electronics, RMIT

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The problem

Anyone who has used Metastock for a long period of time and have hand coded indicators, experts and explorations using the Metastock Formula Language (MSFL) will have come across some of the following problems:

How do I code a protective stop using the MSFL ?
How do I code a profit stop using the MSFL ?
How do I code a time stop using the MSFL ?
How do I reference an entry point ?
How do I reference the price at entry ?
How can I iterate through a series of price values ?
How can I process elements in an Nth dimension array ?
How can I code a pattern matching routine ?

While conceptually simple in most structured languages, unfortunately it is almost impossible to do simply using the MSFL without resorting to some very contorted coding practises, which they are prone to error and are also computationally inefficient.

To understand why this is the case you need to understand how Metastock processes its language constructs.

Why the MetaStock Formula Language has limitations.

On first glance the MSFL appears to be a comprehensive language, rich in arithmetic and logical operators as well as formulas which operate on an array of values, but on deeper inspection and usually after one has used it for long enough its limitations become apparent.

In the MSFL each variable and formula function is treated as an array or series of values rather than a single entity as in most procedural languages such as Pascal and C++. Because of this MetaStock always interprets a whole array of data rather than a single entity so that from a programmer's perspective, access to a particular element at a certain date or bar is not possible. However deep in the internals, which are invisible or transparent to the user, and at the machine level only one value of the array can be processed at any one time so the MSFL interface gives the impression or illusion that all processing of an array of data is carried out simultaneously. For all intents and purposes the user should see it like this.

As an example the closing price for two consecutive weeks of data is shown below

0	1	2	3	4	**5**	6	7	8	9
$10.21	$10.31	$10.45	$10.15	$9.87	**$9.91**	$9.89	$10.01	$10.05	$10.10
Mon	Tue	Wed	Thu	Fri	**Mon**	Tue	Wed	Thu	Fri

When referencing the closing price in the MSFL which value does CLOSE or C refer to? The answer is, all of them and not one particular value !!

To access the closing price for the second Monday how would you do this using the MSFL? Actually it's not very easy ! The MSFL does not provide any native language constructs to do this easily.

In the C++ language you would do this by indexing an array variable with an index that corresponds to the appropriate position in the array. For example if our closing price array has been previously declared we can then access any individual value by indexing it;

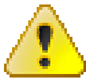
```
DesiredClosingPrice = Close[index];
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

In Vitamin-C, a special Array class with overloaded functions is used to encapsulate the raw price data stored within the MetaStock environment. This provides an intuitive and yet very powerful interface, which can be used to retrieve individual values from an array. This allows out of bounds checking of the array index but for all intents and purposes the user has direct access to each element of the array.

Because C++ uses zero-based subscripting for array variables an index of 0 refers to the first value of the array and an index of 1 refers to the second value in the array and so on. In the following example the DesiredClosingPrice variable is set to the 6th element of the closing price array variable.

```
DesiredClosingPrice = Close[5];
```



In Vitamin-C, automatic checking is done to filter out illegal index values, which would otherwise cause illegal memory access problems and potential program instability issues.

For example the following direct array access would be illegal on an array with 1000 elements:

DesiredClosingPrice = Close[-1];
DesiredClosingPrice = Close[1011];

However in Vitamin-C if the index is out of bounds no problems will occur. In fact when the index value is out of bounds the array will just return return a zero value and an error will be reported to the Report Log.

The MSFL does provide mechanisms to manipulate the values contained in an array similar to other languages. It provides both arithmetic and logical operators to do this but unlike other languages and from a users perspective it always operates on all the values in the array at once and not on any element in isolation. For example if we were to add the value 2 to the closing price we could do this in the MSFL using the following code:

```
NewClose := Close + 2;
```

What exactly happens behind close doors and deep in the internals of the MetaStock environment can be better seen by taking the example above and disseminating it as in the following diagram:

Index →	0	1	2	3	4	5	6	7	8	9
Close	10.21	10.31	10.45	10.15	9.87	9.91	9.89	10.01	10.05	10.10
+	+	+	+	+	+	+	+	+	+	+
2	2	2	2	2	2	2	2	2	2	2
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
NewClose = Close + 2	12.21	12.31	12.45	12.15	11.87	11.91	11.89	12.01	12.05	12.10

Each index corresponds to one value of the closing price data. A value of 2 is added to this closing price and the new value is stored in the 'NewClose' array for that particular index. This process is repeated until all of the closing price data has been processed and stored in the new array. There is no chance for the programmer to intervene in this process.

To do this in the C++ language you would have to iterate through all of the values in the 'Close' array and add 2 to each value. It would look something like this:

```
for(int index = 0; index < BarCount; index++)
  NewClose[index] = Close[index] + 2;
```

This is what is happening at a lower level in the Metastock environment, however the user never gets to see this. The MSFL coding appears to provide a simpler solution since it automatically encapsulates the lower level iterative loop needed to add to each value in the Close price array. However this simplicity comes at

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

the cost of flexibility, which limits the scope of the language since in a lot of cases, access to the index variable or a particular element of the array during the iterative process, is imperative in order to do some really useful things.

Vitamin-C also allows operation on arrays just like the MSFL but it also allows access to each element in the array.

Trying to code simple systems using the MSFL

So what if you can't access an individual element in the price array ! Why does this stop you from creating a simple profit or time stop etc ? To understand this dilemma let us try to construct a profit stop using the MSFL.

For the sake of the argument lets artificially force entry on the first day of every month. i.e.,

```
EntryTrigger := dayofmonth()=1; { force entry on the first day of every month }
```

We will enter at the opening price. i.e.,

```
EntryPrice := Open;
```

We will also exit at the closing price. i.e.,

```
ExitPrice  := Close;
```

Now all we need now is a mechanism that will provide the ExitTrigger when the profit gain exceeds a certain amount. As part of our example, lets force the trade to exit when the profit gain exceeds 50% on the long side. Most people will try and write something like this:

```
ExitTrigger:=(ExitPrice – EntryPrice)/EntryPrice * 100 >= 50;
```

However this won't work because what actually does the EntryPrice refer to ? We know it should be the EntryPrice at the point of entry, or in this case the first day of the month but in this case the EntryPrice is just giving us the opening price on the same bar as ExitPrice, so we are just comparing the opening and closing price on the same bar to see if there has been a 50% increase and ExitTrigger is essentially an array of values where each value in the array represents whether or not the profit gain exceeds 50%.

To calculate the Profit gain we need to know the actual entry price at the point of entry i.e., EntryPrice when EntryTrigger = 1. To do this we use the ValueWhen() function that allows us to reference a specific value when a certain condition is met i.e.,

```
ActualEntryPrice := valuewhen(1,EntryTrigger,EntryPrice);
```

Now we can code the ExitTrigger using the following code;

```
ExitTrigger := (ExitPrice – ActualEntryPrice)/ActualEntryPrice > 0.5;
```

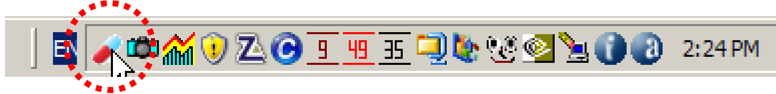
Although not obvious this will not work correctly. The reason why is that if the profit gain is not detected by the time a new EntryTrigger comes along (every Monday in our case) then the ActualEntryPrice point will then change to the next EntryTrigger price point (or next month) and thus the profit gain calculation will become invalid. This is why the MSFL is not designed to handle such coding techniques in a simple way since it does not allow the user access to the internal mechanism that is used by MetaStock to iterate through the array of values.

In the next sections we discuss how we can overcome these limitations using Vitamin-C for MetaStock.

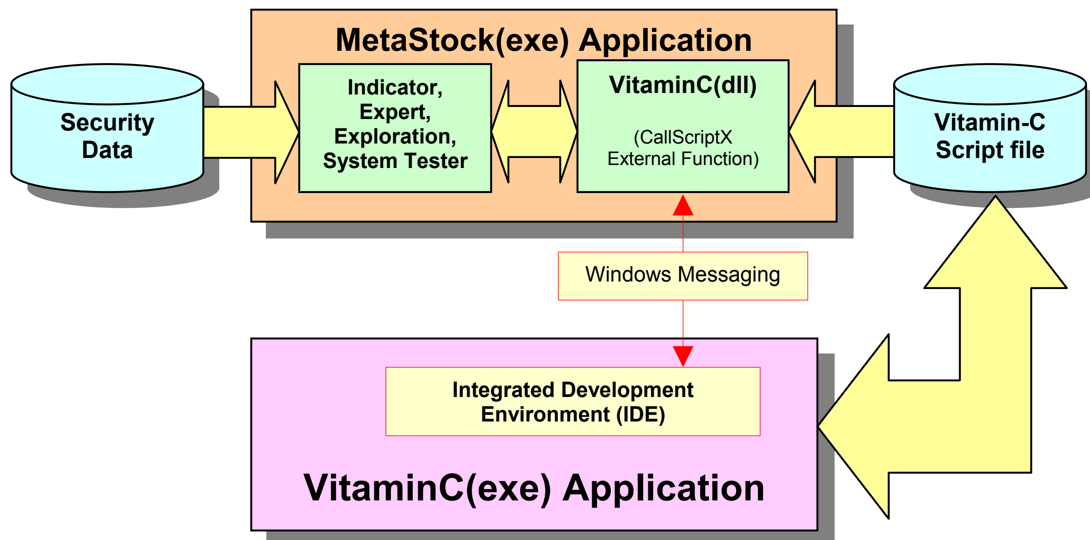
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

How MetaStock Communicates with Vitamin-C

Before we start coding stuff using Vitamin-C it is important to familiarize yourself with how MetaStock communicates with Vitamin-C. The Vitamin-C engine is essentially an external MeyaStock plug-in DLL (called VitaminC.dll) that sits in the MetaStock External Formula directory along with any other MetaStock plug-ins. When MetaStock is started, the initialization routines in the DLL load up the Vitamin-C Integrated Development Environment (IDE), which is automatically minimized to the system tray. Provided that you have registered Vitamin-C and activated it with a license key then you should be able see the Vitamin-C icon in the system tray similar to the following:

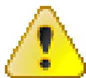


When the VitaminC.dll plug-in is loaded and communicating with MetaStock the DLL also communicates with the Vitamin-C IDE through a Windows messaging process. This allows seamless integration of the IDE with MetaStock as though Vitamin-C was built into MetaStock.



Calling script files from your MetaStock code

There is essentially one type of function (and its variants) used to call the Vitamin-C script from your MetaStock code. The number of user array arguments supplied will determine which variant is used. Implicit with each call to the CallScriptX function is access to predefined data arrays such as Open, Close, High, Low, Volume, Date, Time etc. The additional *UserArray* parameters are available for any user-defined arrays.



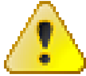
The Vitamin-C engine does not have access to the whole library of MetaStock functions so if you need access to any function or indicator you should first create this using the MetaStock formula language and pass it as an argument to one of the functions. We purposely created Vitamin-C to be like this because we did not see any point in duplicating the entire MetaStock formula library again.

Anatomy of a Vitamin-C function call.

The MetaStock code used to call the external Vitamin-C script takes the following form:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
ExtFml( "VitaminC.CallScriptX",
        "SCRIPT_FILENAME",
        "FUNC_NAME_ARGUMENTS",
        UserArrayArg1,…,UserArrayArg7 { Optional User Supplied Array(s) }
);
```



Please see Appendix B for the complete syntax for all variations of the CallScriptX function.

	Parameter	Required	Argument Type	Description
1	"VitaminC.CallScriptX"	Yes	String	The first parameter in the call specifies the DLL and name of DLL in the usual MetaStock convention for calling External DLL function libraries. The X refers to the number of user array arguments passed. If X is not specified then no arguments are passed to the function otherwise X can be anything from 1 to 7, which specifies the number of user arguments that the function accepts.

2	"SCRIPT_FILENAME"	Yes	String	Name of script file enclosed by quotes. If no path is given then the default path '\VitaminCScript' is used.
				Examples:
				`"MovingAverage.c"`
				`"simple.c"`

3	"FUNC_NAME_AND_ARGUMENTS"	Yes	String	This string defines the function name and arguments. If the function accepts constant valued arguments than these will be included in the string separated by commas and enclosed by left and right parentheses pair. If the function returns a value then this value will be ignored by MetaStock. All values returned to MetaStock should be written to the Result Array. Contrary to the limits on the number of arguments that can be passed to an external DLL there is no limit to the number of constant arguments passed to the function C-Script since it just equates to one string argument. If a char string is passed as an argument to a char pointer parameter in a Script function then to avoid conflicts with the double quotation marks used to define the string argument then a character string should be enclosed with single quotation marks (the one next to the left of the '1' key on your key board)
				
				Examples of Vitamin-C script function name and argument:
				`"EMA(10)"`
				`"CBLTrailingStop(3)"`
				`"TrailingStop(BAND,LONG)"`
				``"SetStop(`AMP`,10.67)"``

4..7	UserArrayArg1… UserArrayArg7	Optional	MetaStock Array	Standard MetaStock formula language data array. Anything that equates to an array can be used as a UserArray argument.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved

				Examples of User Array Arguments:
				`Mov(C,10,E)`
				`(Open+Close)/2`
				`Ref(Cross(MACD(),Mov(MACD(),9,E)),-1)`

Example of using CallScript with No User Array Arguments

```
ExtFml( "VitaminC.CallScript",
"GuppyCBL.c",
"CBLTrailingStop(3)");
```

Examples of using CallScript with 1 User Array Arguments

```
EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"ProfitStop.c",
"ProfitStop(5)",
EntryTrigger);
```

```
EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",
"TimeStop(30,'AMP')",   { string character array wrapped in single quotes }
EntryTrigger);
```

Example of using CallScript with 3 User Array Arguments

```
ExtFml("VitaminC.CallScript3",
"TrailingStop.c",
"TrailingStop(BAND,SHORT)",
3*ATR(10),
CLOSE,
HIGH);
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Your First Step

Coding simple systems using the ‘Vitamin-C’ language

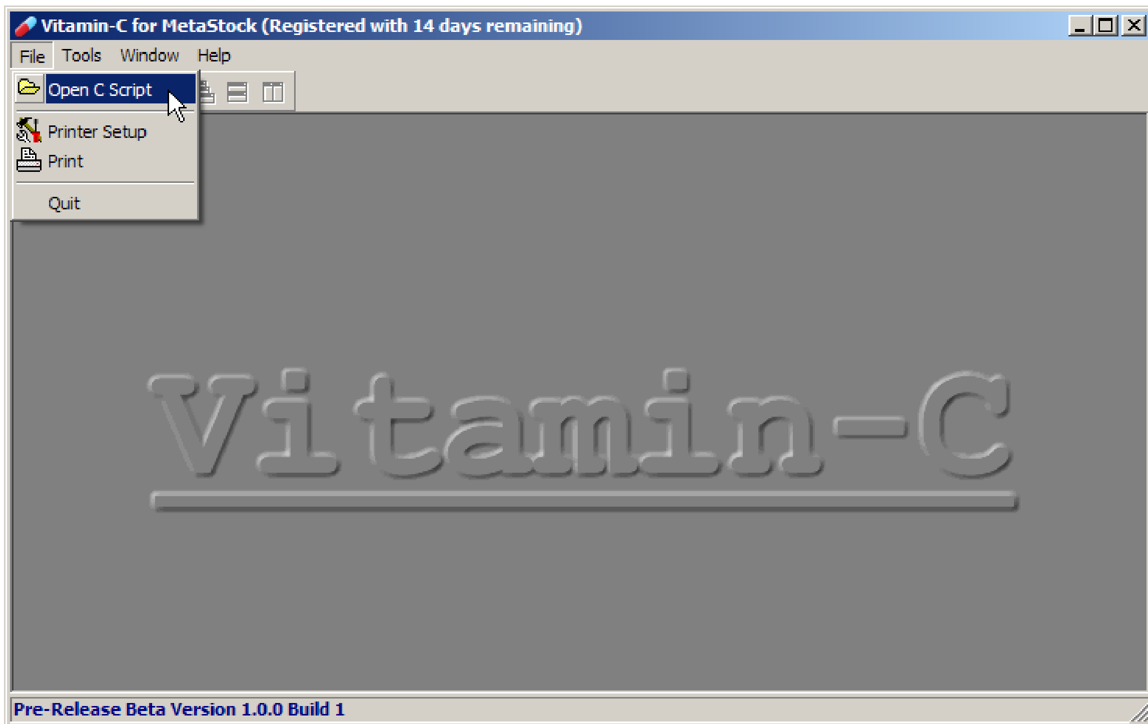
How would we code a simple time stop using the ‘Vitamin-C’ language ? Before we get to addressing this issue lets start with a simple example of the process of using Vitamin-C to write some C++ code that can be accessed from MetaStock. We will start with something that is pretty useless and superfluous just to illustrate the process of creating a Vitamin-C script and calling it from within the MetaStock environment as well as demonstrating the use of subscripting an array variable. You will see that the process of creating and running C++ compatible script from MetaStock is quite seamless as though MetaStock was running the code itself.

We will now create a function called ‘Demo()’ that returns the values of the Closing price array using array subscripting in a for-loop. If you don’t know what a for-loop is or have not programmed in the C++ language before then don’t worry, as we will discuss these things in depth later on. For now just follow the instructions.

If you have installed Vitamin-C Integrated Development Environment (IDE) then it should automatically start when MetaStock is started or it can be started from its desktop icon or from the start menu.

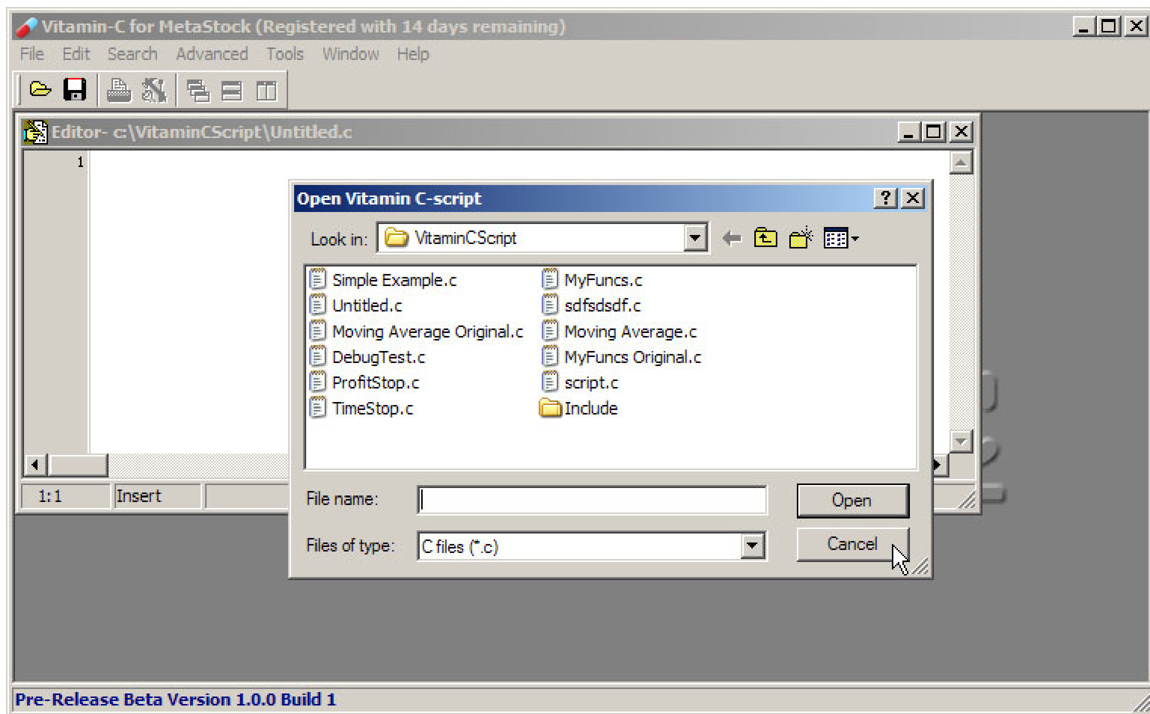
Once the Vitamin-C IDE is running, to create and edit a new Vitamin-C script click on:

File → Open C Script



When the ‘Open C-Script’ Dialog box appears then click on Cancel:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Now you can start editing!

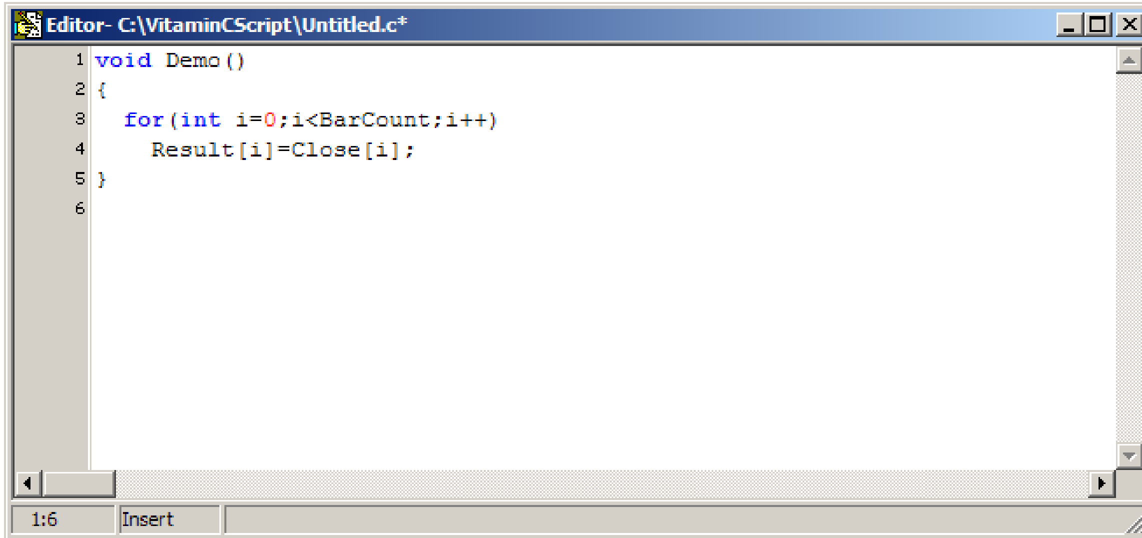
Type the following code below into the editor, as you would using NotePad or Word etc. You can also copy and paste the code directly into the editor if you want to be sure that you have copied it correctly ! To do this, highlight the code below in yellow background and copy it to the clipboard using the 'Ctrl'+'C' keys or use the 'Copy' command from the edit menu. Once you have done this paste it back into the Vitamin-C edit window using 'Ctrl'+'P' or the Paste command from the edit menu. Don't worry if you are not familiar with the C++ language as we will cover this more in-depth later on.

```
void Demo()
{
  for(int i=0;i<BarCount;i++)
    Result[i]=Close[i];
}
```

Because the editor in Vitamin-C IDE is context sensitive it will highlight different code contexts in different colours. Unlike the bland old MetaStock editor, which shows everything in black and white in a small window that can't be resized, the Vitamin-C editor highlights different contexts in different colors and does not restrict editing in a small window.

If you have copied the code segment above into the Vitamin-C editor it should something look like the following.

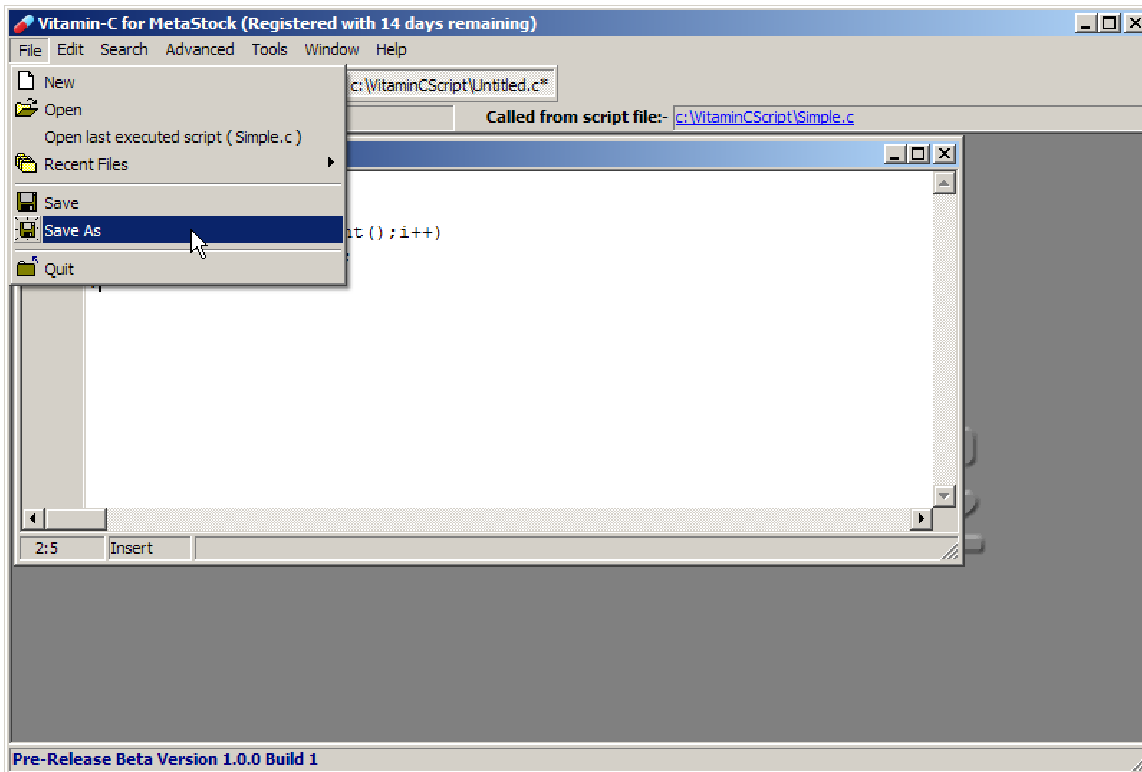
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Now you should save this code into a file, which we will call ‘simple.c’ rather than use the default ‘Untitled.c’ filename.

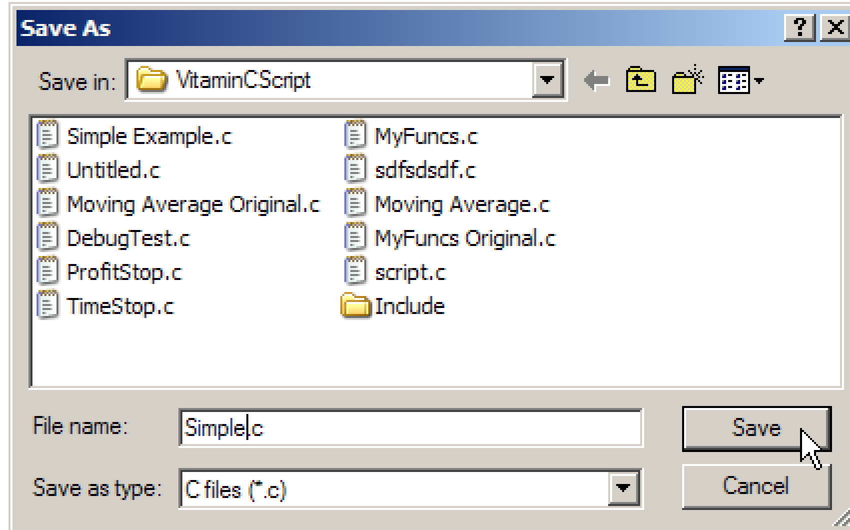
To do this click on the following menu sequence:

File → Save As



Type ‘Simple.c’ in the Filename edit box and then click ‘Save’

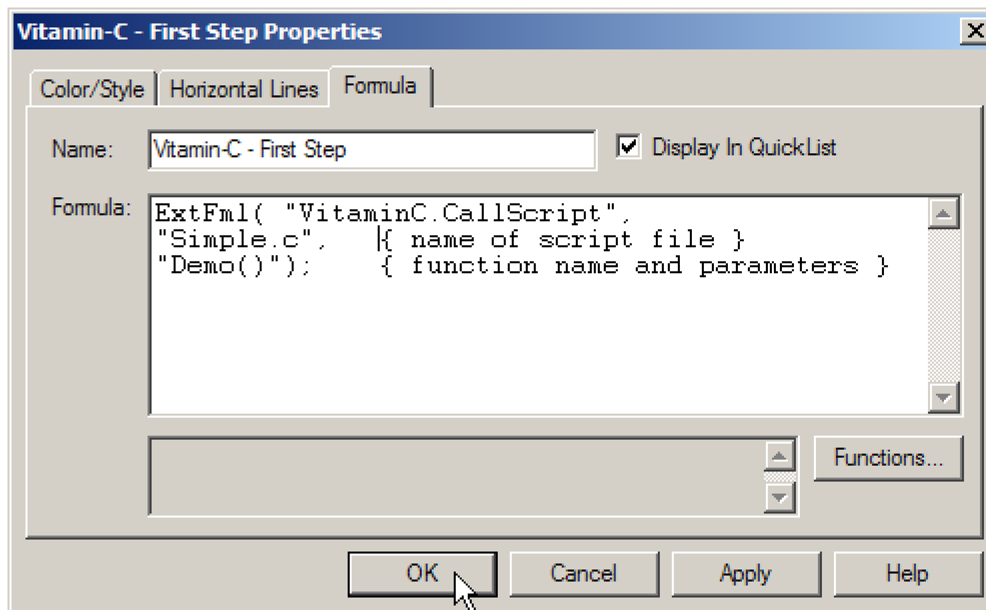
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Now we will create an indicator in MetaStock, which calls this function and overlay this indicator on a chart.

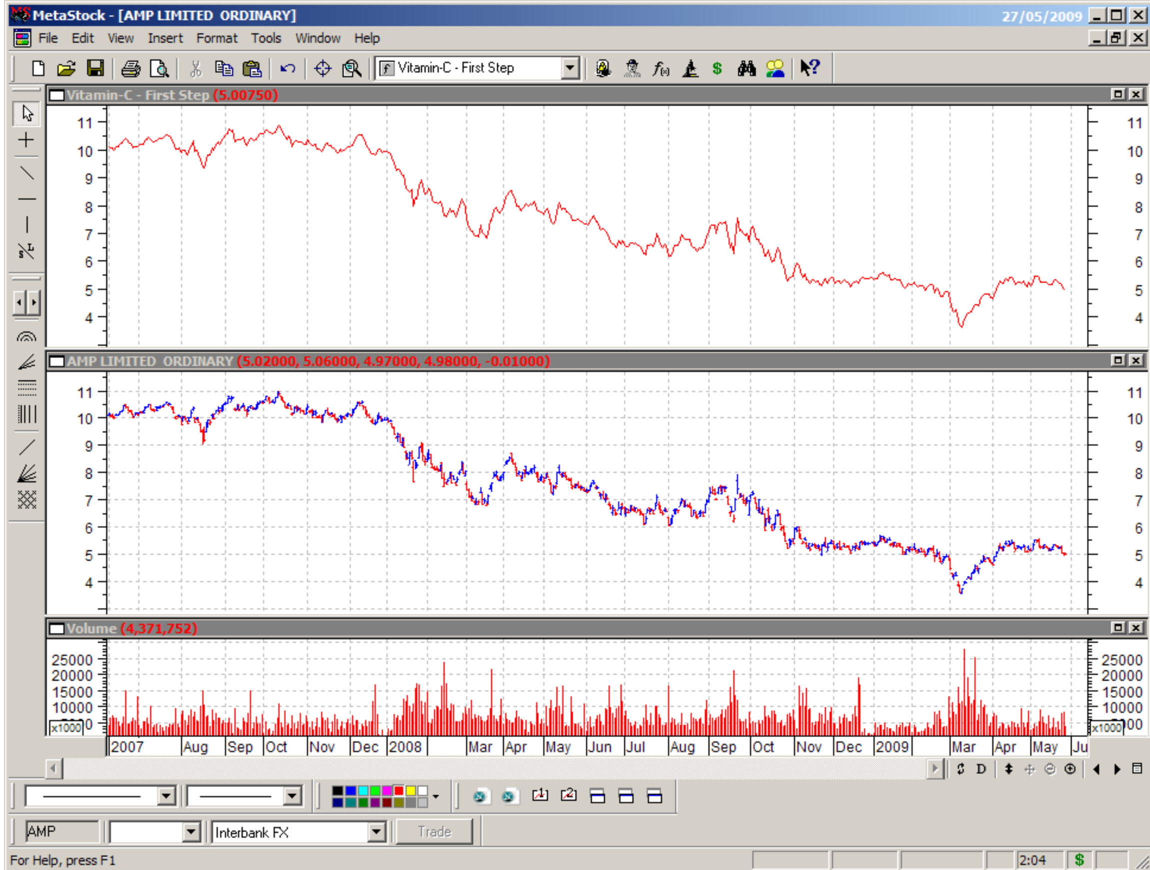
Run MetaStock, and create an indicator with the following code and call it 'First Step'. You can also copy and paste the formula if you like.

```
ExtFml( "VitaminC.CallScript",
"Simple.c",       { name of script file }
"Demo()");        { function name and parameters }
```



Load up a chart and now add the indicator to the chart. What do you see ?

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



The indicator should be displaying the closing price of the underlying security.

Exercise

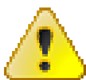
As an exercise expand the display and check to see whether the indicator is actually showing the closing price.

A Closer look

Now how exactly does this bit of C++ code work ??

```
void Demo()
{
  for(int i=0;i<BarCount;i++)
    Result[i]=Close[i];
}
```

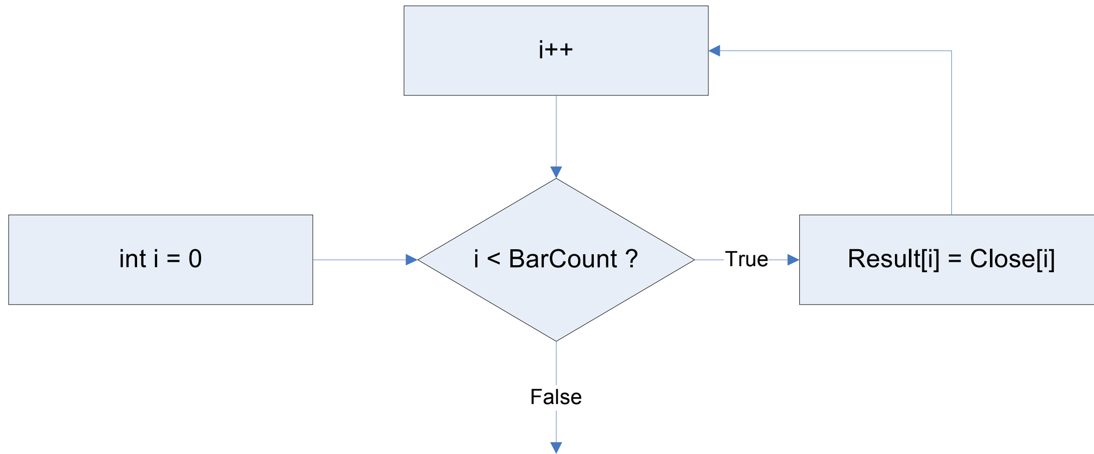
The first line consists of the function declaration and start of the function block. The '**void**' word tells us that when this function is called it does not return a value of any kind to calling program. Next to the type of return value is the name of the function (in our case it is called '*Demo*') followed by opening and closing parenthesis which tells us about any parameters that the function needs which in this case is none. The body of code that defines what the function does is always enclosed within braces.



In MetaStock braces are used to denote comments that don't do anything. In 'C/C++' braces are used to delineate blocks of code. An opening brace should always be paired up with a closing brace.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The next two lines of the code form the body of the function. This basically defines what the function will do. The first line of the body is a for-loop which is a construct used to control how many times a particular body of code gets executed. To understand the role of the for-loop in the context of this particular function we will take a closer look by means of a flowchart:



As you can see from the flowchart above a temporary index or iterator variable is created called ‘i’ and is initialized to zero, which is the first value of the array for both the closing price and result arrays. The index variable ‘i’ is temporary because it is local to the for-loop and does not exist outside of it. This variable will be created temporarily for the loop only and will be discarded when the loop is finished.

Next a comparison is made to the total bar count variable BarCount. The BarCount variable is a special reserved variable that contains the number of bars of information available. This variable can only be read from but not written to. Attempting to write to this variable will halt program operation and report an error. If the index is less than BarCount then the value of the closing price for the bar denoted by the index ‘i’ is assigned to the element in the Result array denoted by the same index.

In Vitamin-C the Result array is predefined and has a special significance over all of the other pre-defined price arrays such as, Open, Close, High, Low. The Result array is used to return values back to the CallScript function in MetaStock that was used to call the script. It can also be read from or written to whereas the other predefined arrays can only be read from.

The value of the index is then incremented and another comparison between the index and the bar count is made again. If it is true then the closing price is assigned to the Result for the next bar and so on. If the index has reached BarCount which is one element passed the last element of the array then the comparison becomes false and the loop is terminated and the temporary index variable disappears. The Result array now contains all of the closing price values and this is available for MetaStock to read.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

What happens if you make a mistake?

It is most likely that the first time you write your own code and run it that it won't run because it has some syntax errors. Fortunately the Vitamin-C IDE has built in tools, which can help you to minimize errors before and after the code is run.

In Vitamin-C there are two methods of detecting errors. If you have used a 'C/C++' compiler before then you will know that when you compile your C/C++ code the compiler may issue warnings or errors if your code has been written incorrectly. You then fix these errors and rerun the compiler until there are no more errors. When there are no remaining errors then the code is ready to be run. The Vitamin-C environment is a little bit different because it has an interpreter environment where the code is interpreted rather than compiled to machine code instructions and the run by the CPU. For this reason it is possible to run some script even if it contains errors, which can trip it up. In this case the C-interpreter engine will halt program operation at the first error and display a description of the error in the error-log below the editor window. Unfortunately this will also cause MetaStock to halt as well and display an exception dialog box. For this reason it is best to eliminate errors before the script is run by the interpreter. We shall discuss this in the next section.

Detecting coding errors before you run the code-script.

Detecting code errors before you run the code-script requires some sort of C/C++ syntax checking software. This type of software is a major application in itself because the C/C++ standard is such a vast and comprehensive language environment. Fortunately nearly every decent C/C++ compiler contains it's own built-in syntax checker which can help you fix errors before the code can be compiled. Vitamin-C makes use of a freely available C/C++ compiler to do the syntax checking on the Vitamin-C script and report the errors back to Vitamin-C. It is no coincidence that it is this same compiler that was used to create Vitamin-C in the first place ☺

Installing the Free Compiler on your system

To use an external C++ compiler to parse the Vitamin-C script file and report back the errors to the IDE requires that you have installed the freely available Borland C++ compiler Version 5.5 compiler. Please see Appendix D for more details how to obtain and install this compiler on your system.

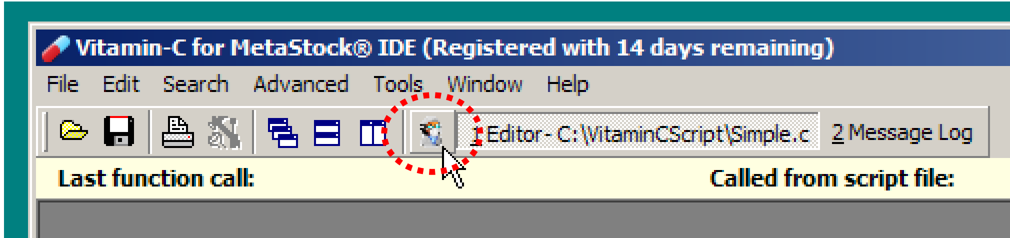
Running A Syntax Check

As an example let's purposely add some mistakes to the previous Vitamin-C code example. We will misspell the name of closing price array as outlined in grey background in the following:

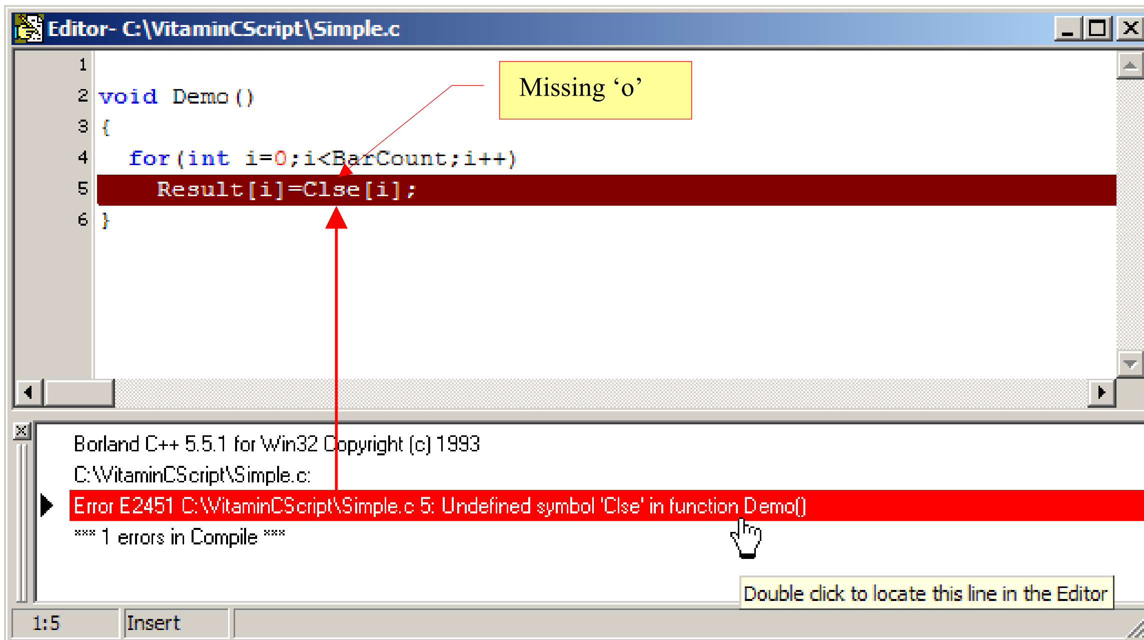
```
void Demo()
{
  for(int i=0;i<BarCount;i++)
    Result[i]=Clse[i];
}
```

Now run the Syntax Checker by clicking on the main Toolbar button as shown below:

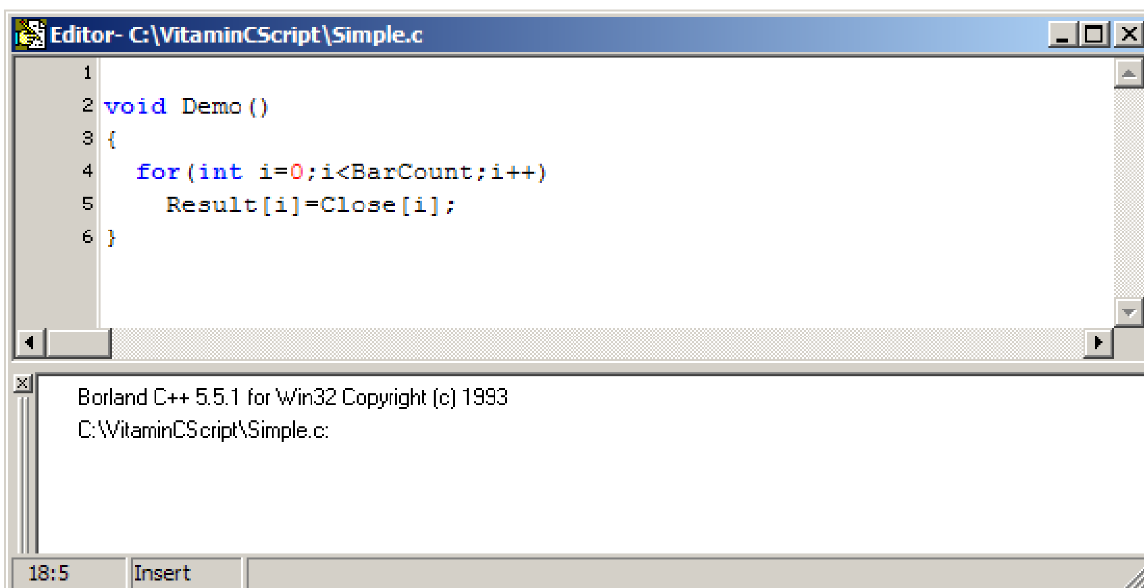
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



After clicking on the Syntax Check Tool button an Error Log window will open up below the Editor window and report any messages from the compiler as in the following screen grab below. Any error messages are highlighted in red and any warning messages are highlighted in yellow. You can double click on these messages to highlight them in the editor as shown in the example below:



Correct the errors by inserting the missing 'o' and re-run the syntax checker.



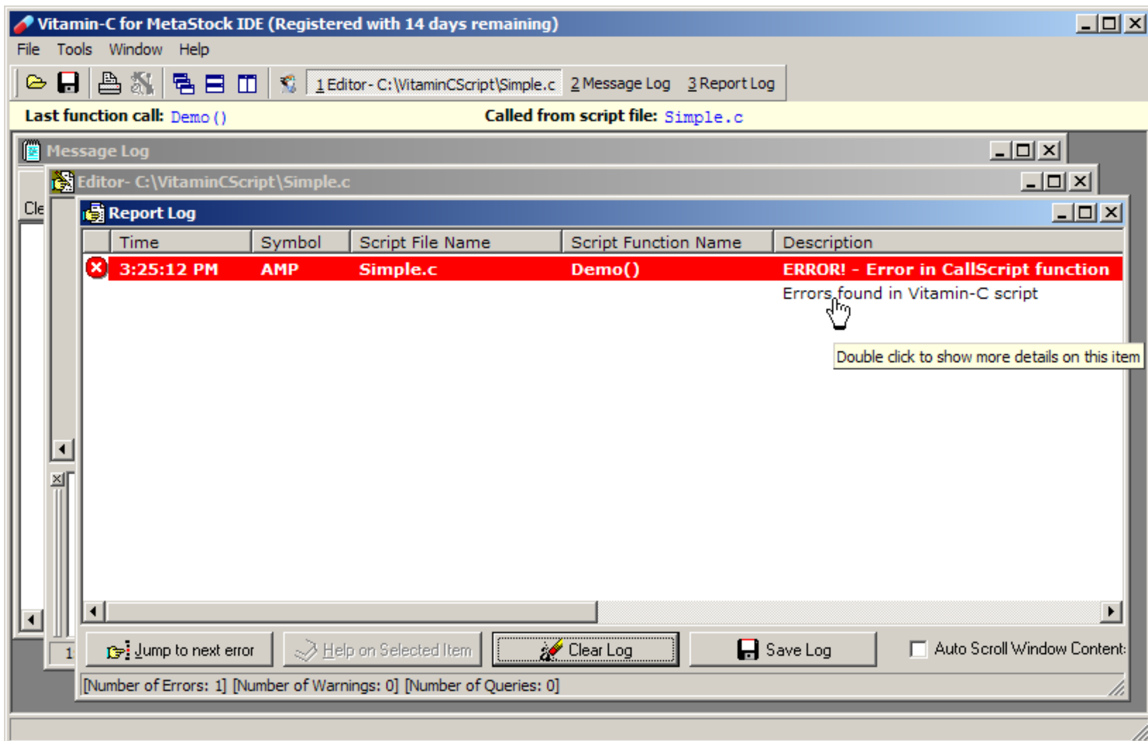
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Correcting coding errors at runtime

If your code has errors and you have not corrected them before you run the script by calling the script from MetaStock, then Vitamin-C will report any errors in your code and give an explanation as to the cause of the problem. Sometimes the error description may be a bit cryptic for the person just starting out in C/C++ but with experience you should be able to identify and correct these errors quite quickly. Even so it is better to correct errors before you run the code but for the sake of demonstration lets purposely add some mistakes to the previous Vitamin-C code example. We will misspell the name of closing price array as outlined in grey background in the following:

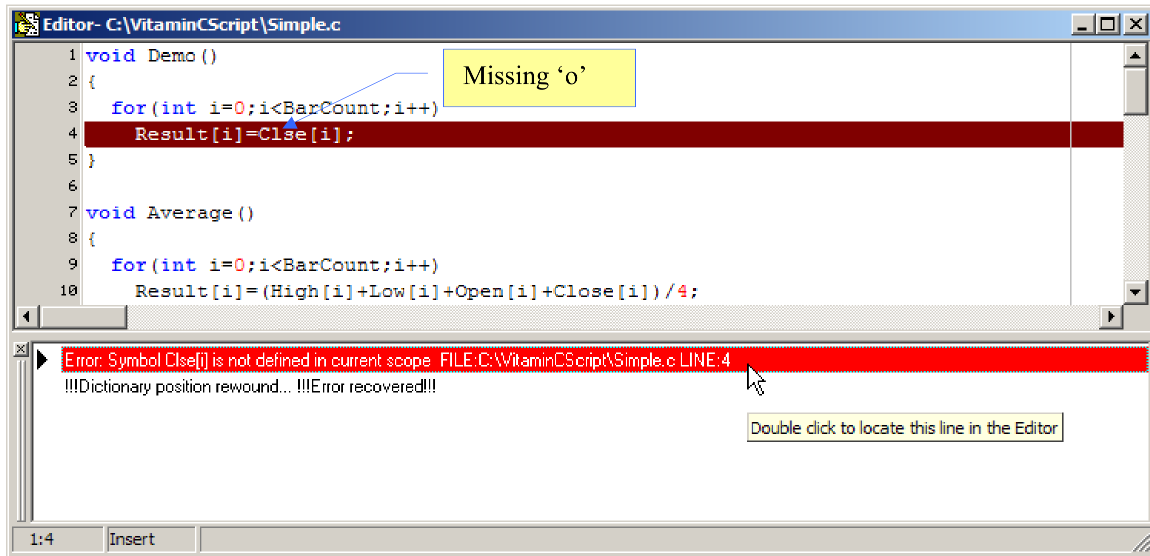
```
void Demo()
{
  for(int i=0;i<BarCount;i++)
    Result[i]=Clse[i];
}
```

When we run the code again by double clicking on the indicator that references it, then the IDE will report an error in the Report log.



Double click on the error message in the Report Log or click on the Editor Window and double click on the error message in the Error Log window at the bottom of the Editor Window.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



You will see that the line with the error is highlighted and the mistake can be corrected by adding 'o' to Clse to make it Close.

Auto Save feature.

The Vitamin-C IDE has an auto save feature, which detects when you switch to another window and automatically reminds you to save the C-script only if it has been modified. Once you have saved the C-script file the results in MetaStock are not automatically updated. You have to double click on your indicators or experts in MetaStock to see the changes or re-run your explorations. There is also a reminder for this as well.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

A Brief Introduction to C++

Before we can move on we need to discuss some of the fundamental aspects of the C/C++ language. In Appendix A at the back of this guide we have included an introductory section on the C/C++ language. If you are a seasoned C++ programmer then you can skip this section. For newcomers we suggest that you read it because it will form the basis of understanding the material presented in the following chapters. This discussion of the C/C++ language is by no means exhaustive and only really scrapes the surface of its true capabilities.

Whilst the C/C++ language gives you plenty of rope to swing on, the same bit of rope can also be used to hang yourself with, so bear this in mind before trying use every bit of the C++ language to do something that could otherwise be done simply. At the end of this section if you would like to know more then please refer to the end of this User Guide for more references on this subject.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Predefined Variables and Functions

Predefine variables and functions are available to be used in your Vitamin-C code and should not be redefined or modified in your code.

Predefined Variables

The following variables are accessible from your Vitamin-C script and should not be re-declared.

Variable	Variable Type	Description	Read Only (RO) Read and Write (R/W)	Analogous Variable in MetaStock.
BarCount	integer	Total number of accessible bars typically used in loop constructs.	RO	-

Vitamin-C specific functions

The following predefined functions are available to be used in your Vitamin-C script and should not be re-declared.

Function prototype	Description
`void dprintf(const char *fmt, ...)`	Similar to the standard library printf() except that it writes to a dedicated debug log window in the Vitamin-C IDE.
`void ReportError(const char *fmt, ...)`	Used to report errors from your Vitamin-C script but also allows printf() style reporting back to the Vitamin-C IDE.
`bool IsSymbol(const char *_Symbol)`	Used to compare the current symbol with a given symbol. If the current symbol is identical to the symbol being passed as the parameter then the function will return true. For example IsSymbol("AMP") will return true if the current symbol data is "AMP" otherwise it will return false.
`bool IsSymbolIn(const char *_Symbol)`	Similar to IsSymbol() but looks for a partial match. For example IsSymbolIn("AMP") will return true if the current symbol is either "AMPIZZ" or "AMPISQ"
`long GetDate(int _index)`	Returns the current date as a long integer value in the form of YYYYMMDD, where YYYY=years, MM=months (12-1), DD=days (31-1)
`const char *GetDateString(int _index)`	Returns the date as a formatted null terminated string in the form of "YYYY-MM-DD"
`long GetTime(int _index)`	Returns the current time as a long integer value in the form of HHMMTTT, where HH=hours (23-0), MM=minutes (59-0), TTT=ticks (999-0)
`const char *GetSymbol()`	Returns the current symbol as a pointer to a null terminated string.
`const char *GetSecurityName()`	Returns the current security name as a pointer to a null terminated string.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

<table>
<tr><td><code>const char *GetPeriodicity()</code></td><td>Returns the periodicity of the current security as null terminated string.
<table>
<tr><th>Periodicity</th><th>GetPeriodicity() returns</th></tr>
<tr><td>Intraday</td><td>“Intraday”</td></tr>
<tr><td>Daily</td><td>“Daily”</td></tr>
<tr><td>Monthly</td><td>“Monthly”</td></tr>
<tr><td>Quarterly</td><td>“Quarterly”</td></tr>
<tr><td>Yearly</td><td>“Yearly”</td></tr>
</table></td></tr>
</table>

Predefined Array Variables

These Array variables are accessible from with your Vitamin-C script and should not be re-declared. The Array type is a special Class type used in Vitamin-C that encapsulates the price arrays or arrays of single precision floating point numbers, used in MetaStock. You’ll learn a bit more about using Array variables later on however there are full details on the Array class in Appendix C at the end of this User Guide. With the **`Array`** class you can have individual element access using a subscript or you can treat the whole array as an object in much the same was that it is used in MetaStock. The *Array* variables marked read only should only be read from and not written to, or only used in the left side of an expression. However if a value is assigned to them then this value will be ignored.

Variable	Description	Read Only (RO) Read and Write (R/W)	Analogous Variable in MetaStock.
Open	Opening Price	RO	Open or O
Close	Closing Price	RO	Close or C
High	High Price	RO	High or H
Low	Low Price	RO	Low or L
Volume	Volume	RO	Vol
Result	Stores the values returned back to the MetaStock calling function	R/W	-
User1	1st User Array parameter	RO	Any MSFL Expression
User2	2nd User Array parameter	RO	Any MSFL Expression
User3	3rd User Array parameter	RO	Any MSFL Expression
User4	4th User Array parameter	RO	Any MSFL Expression
User5	5th User Array parameter	RO	Any MSFL Expression
User6	6th User Array parameter	RO	Any MSFL Expression
User7	7th User Array parameter	RO	Any MSFL Expression

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Your Second Step

The last example was pretty basic but should give you an idea of how the Vitamin-C environment fits in with the MetaStock environment. Note that there was no use of a compiler anywhere (except for syntax checking), which is one of the really nice features of the Vitamin-C environment! Now lets step up the pace a bit and add a bit more complexity. We will create a function, which computes the average of the High, Low, Open and Close of the bar for each bar in the array.

We'll discuss functions later on but just for now a function is a section of code, separate from the main program that, perform a single, well defined task.

A function is a section of code, separate from the main program that, perform a single, well-defined task.

In the MetaStock Formula Language (MSFL) we can compute this average using the following code but again we are just trying to demonstrate how Vitamin-C allows as individual access to each element of the array. This is how you would do it in MetaStock MSFL.

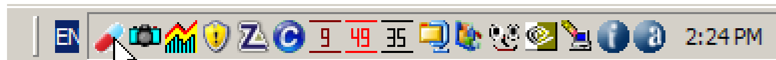
```
Average:=(High + Low + Open + Close)/4;
```

Now we will code the same thing in Vitamin-C using array indexing. We can also code it using array processing in a similar fashion to the way it is coded in MetaStock but we will discuss this later on. The Vitamin-C code used to do this is shown below in yellow background.

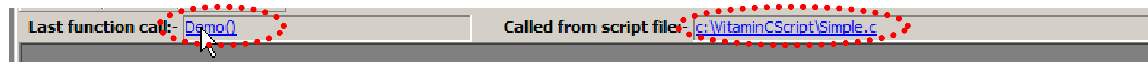
```
void Average()
{
  for(int i=0;i<BarCount;i++)
    Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
}
```

Instead of creating a new script file we will just add the function below the Demo function that we created earlier on. One of the advantages of Vitamin-C is that it allows us to create a library of functions in the one script file and call each one individually. You may not want to do this though and keep each function in a separate script file.

If it's not already open, open the file Simple.c file in the Vitamin-C editor. You can do this in a number of ways. If the Vitamin-C window is not being displayed you can bring it up by clicking on its icon in the system tray.

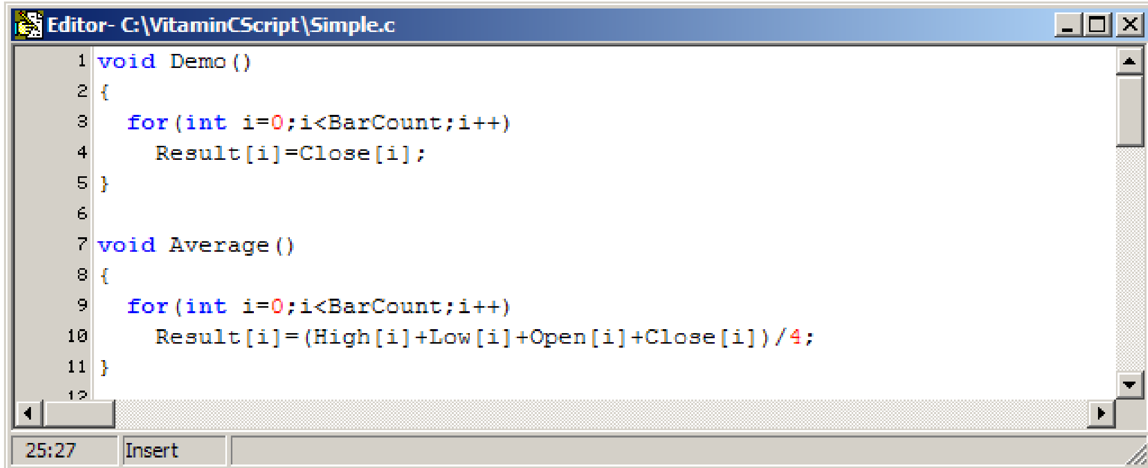


If the script is not open you can quickly open it from the File → Recent Files, menu or click on the function or script file name in the top status bar if it has previously been run in MetaStock.

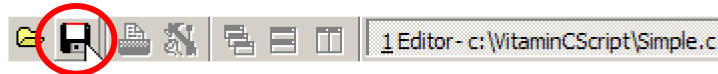


Now add the code to the 'Simple.c' script below the original 'Demo' function.

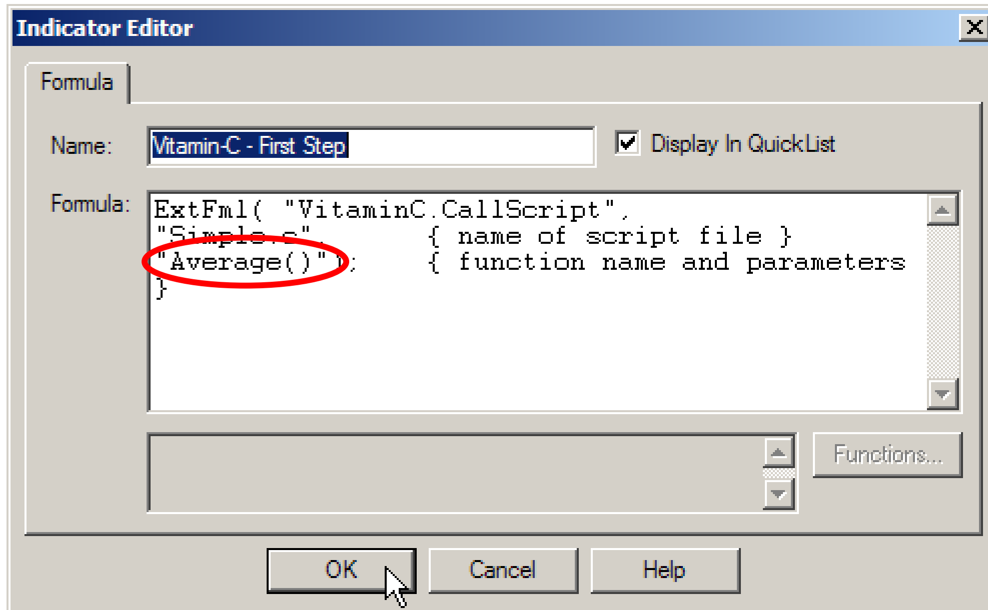
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Save the script by clicking on the 'Save' tool button in the main toolbar.

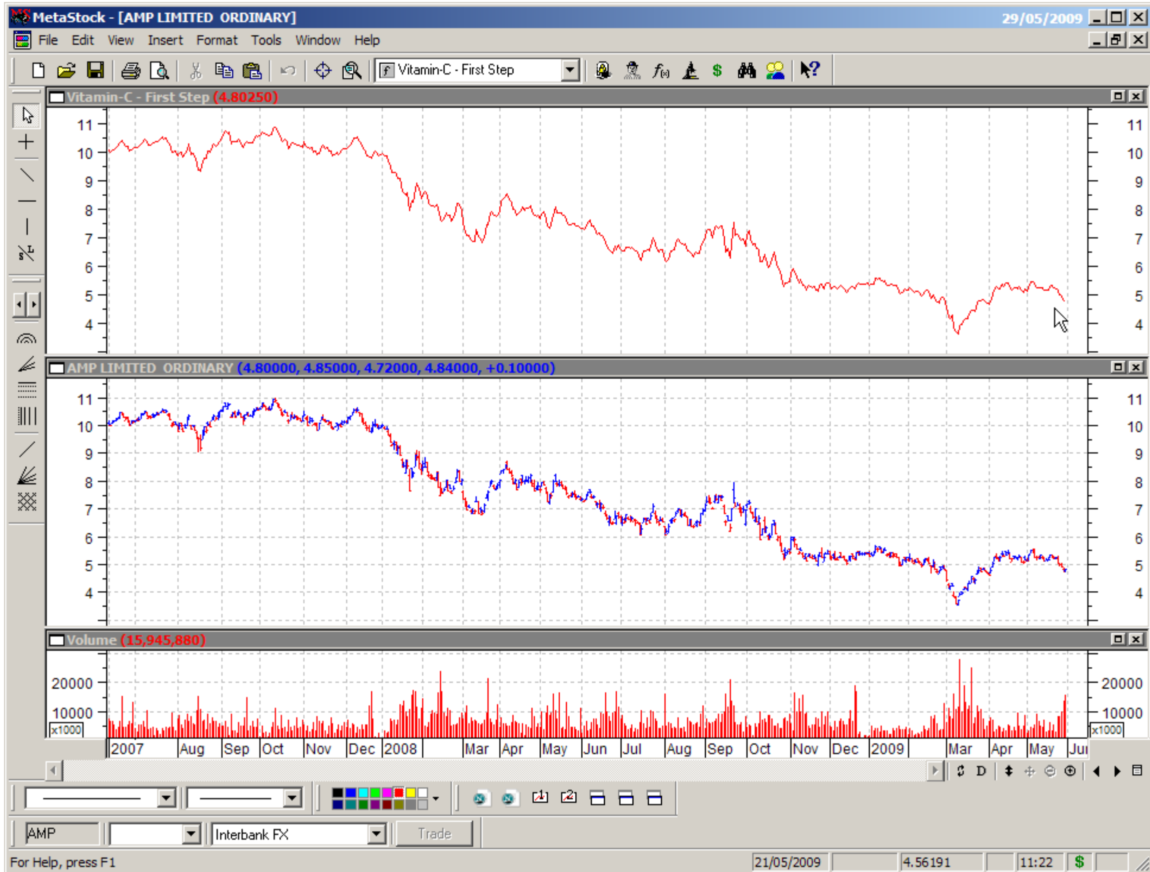


Switch back to the MetaStock window and edit the indicator by double clicking on the indicator chart and change the function call from 'Demo()' to 'Average()'.



Click 'OK' and the indicator should automatically invoke the new function and the indicator will change slightly.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Exercise

As an exercise expand the display and check to see whether the indicator is actually showing the average of the four prices

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Your Third Step.

Now we will change the code so that after a certain date we will calculate the average slightly different so that is the average of only H,LC instead of H,L,C,O. Lets chose an arbitrary date such as 3rd November 2008.

> In Vitamin-C we use long integer variables to represent dates and time, which provides a convenient means to express dates irrespective of country. For example the date 3rd November-2008 would be represented by the long number 20081103, which is sort of like writing the date backwards. In general we represent dates using long integer variables in the following format:
>
> YYYYMMDD
>
> Where ,
>
> - YYYY = year,
> - MM = month [12-1]
> - DD = day [31-1]
>
> Time is also represented by a long integer number of the following format:
>
> HHMMTTT
>
> Where,
> - HH = hours [23-0]
> - MM = minutes [59-0]
> - TTT = ticks [999-0]

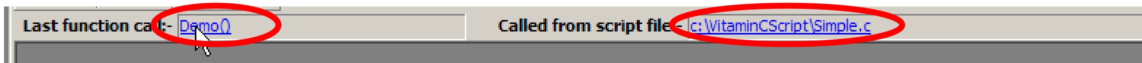
Ok so now lets modify the code and call the function 'NewAverage'. The additional code is highlighted in grey background:

```
void NewAverage()
{
  for(int i=0;i<BarCount;i++)
  {
    if(GetDate(i) > 20081103)
      Result[i]=(High[i]+Low[i]+Close[i])/3;
    else
      Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
  }
}
```

Instead of creating a new file we will add the new function to the existing file 'Simple.c'. If it's not already open, open the file Simple.c in the Vitamin-C editor. You can do this in a number of ways. If the Vitamin-C window is not being displayed you can bring it up by clicking on its icon in the system tray.

If the script is not open you can quickly open it from the File → Recent Files, menu or click on the function or script file name in the top status bar.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Now add the code to the 'Simple.c' script below the original 'Average' function.

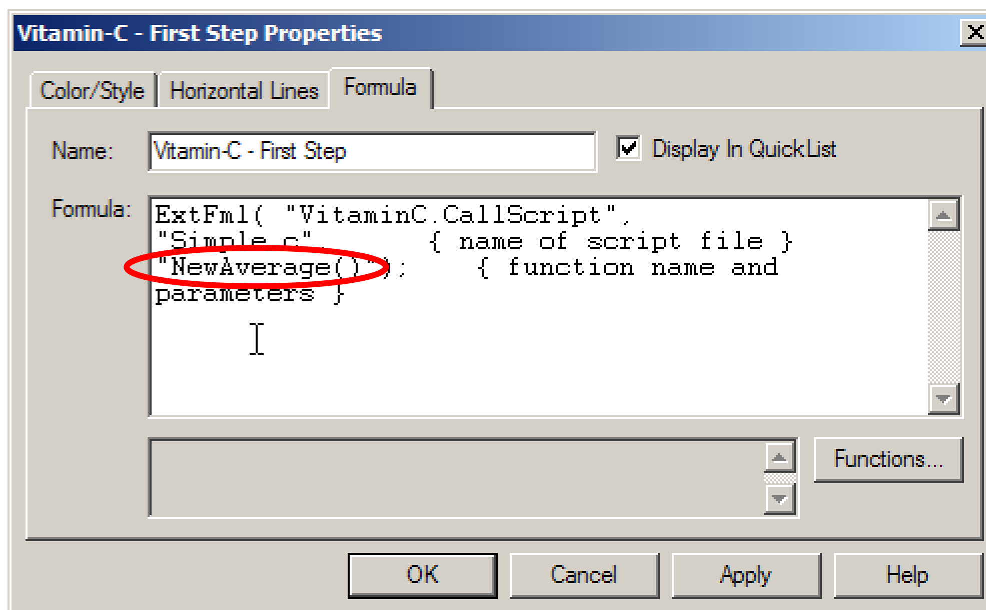
Editor- C:\VitaminCScript\Simple.c

```
 7 void Average()
 8 {
 9   for(int i=0;i<BarCount;i++)
10     Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
11 }
12
13 void NewAverage()
14 {
15   for(int i=0;i<BarCount;i++)
16   {
17     if(GetDate(i) > 20081103)
18       Result[i]=(High[i]+Low[i]+Close[i])/3;
19     else
20       Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
21   }
22 }
```

25:27 Insert

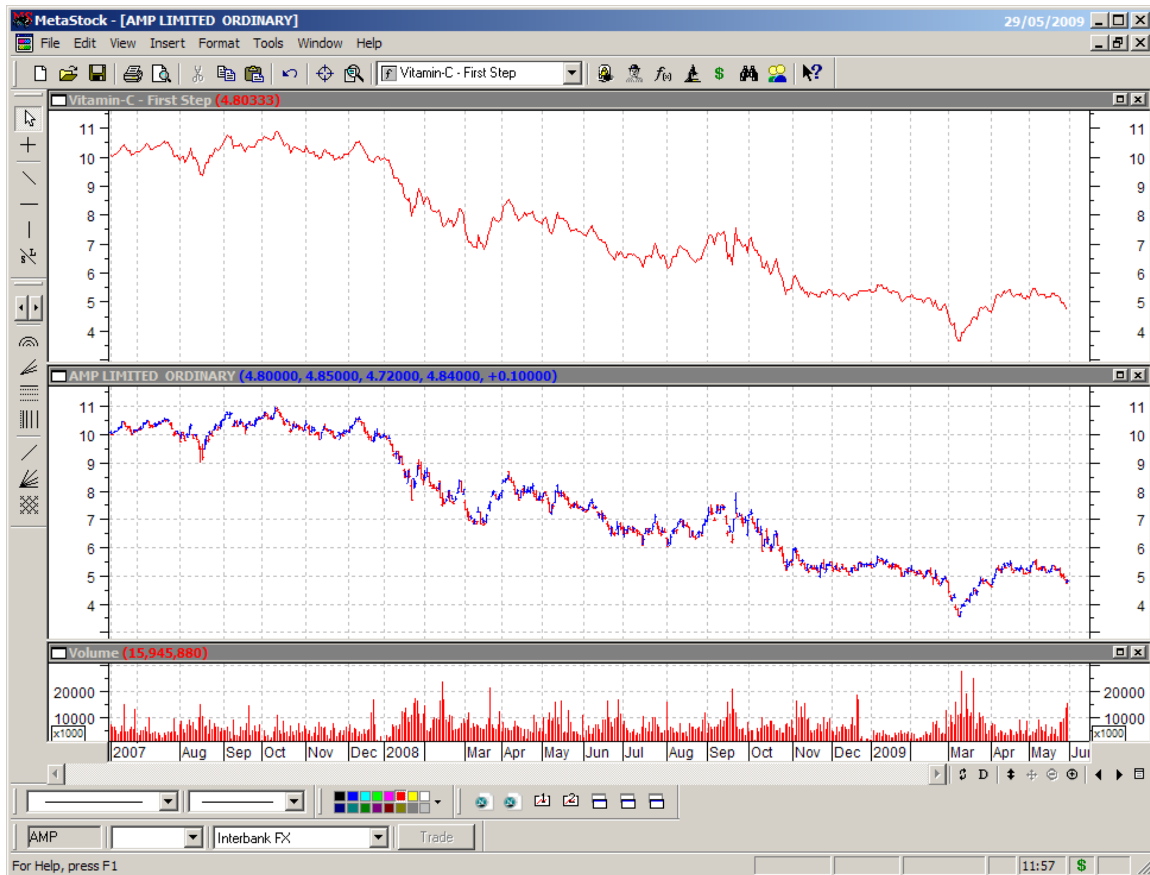
Save the script by clicking on the 'save' tool button in the Vitamin-C window.

Switch back to MetaStock and edit the indicator by double clicking on the indicator chart and change the function call from 'Average' to 'NewAverage'.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Here is what the indicator looks like now.



OK it's hard to see any change so lets modify the code slightly so that we can check if the code is actually doing what it is supposed to do.

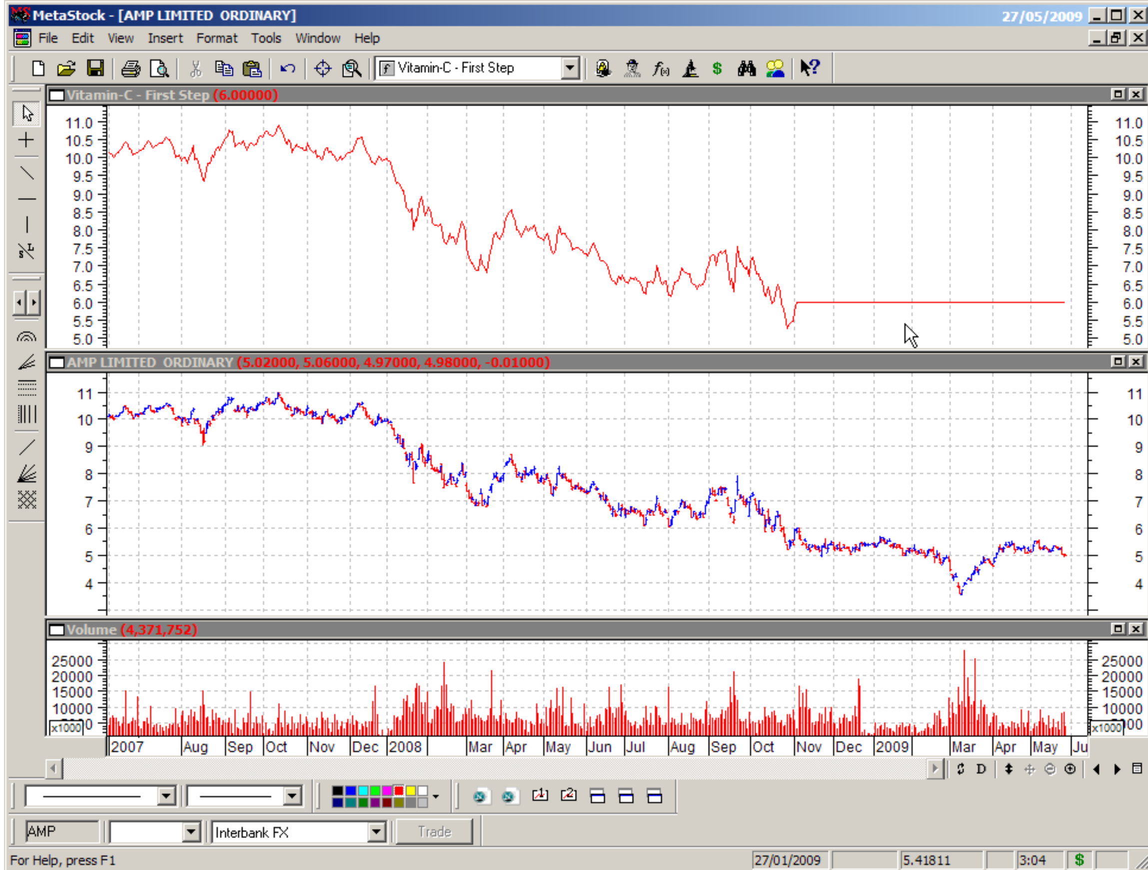
Instead of calculating a different average after the 3rd Nov 2008 we will substitute the average with a constant value of say 6.

This is what the code looks like now.

```
void NewAverage()
{
  for(int i=0;i<BarCount;i++)
  {
    if(GetDate(i) > 20081103)
      Result[i]=6;
    else
      Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
  }
}
```

When we change the Vitamin-C script and update the indicator in MetaStock we get the following.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Yep the code works as expected ! After the 3rd of November 2008 the indicator displays a constant value of 6 !! OK so that was fun so why don't we get a bit smarter and make the indicator display the last value just at the 3rd November 2008 rather than display an arbitrary value of 6.

To do this we need some sort of flag, which is just a fancy word for a variable that contains a value. We declare a flag called 'lastvalue' to be of type float, which is short for floating-point variable, which is the type of value that MetaStock uses for its price and volume arrays.

A floating-point variable consists of both integer part and a decimal part with a decimal point that separates the two. Examples of a floating point number is:

12.4567
-2,000,000.78
$10.6789

Two types of floating point variables exist in the C++ language. This is the float and double which is short for double precision. For most cases in the trading world a float variable is all that is needed to represent price data which also has the added benefit of occupying only half the memory required by the double variable. Indeed all of the price values used in MetaStock are stored using floats rather than doubles. However the double type is still available to be used in the Vitamin-C script if so desired.

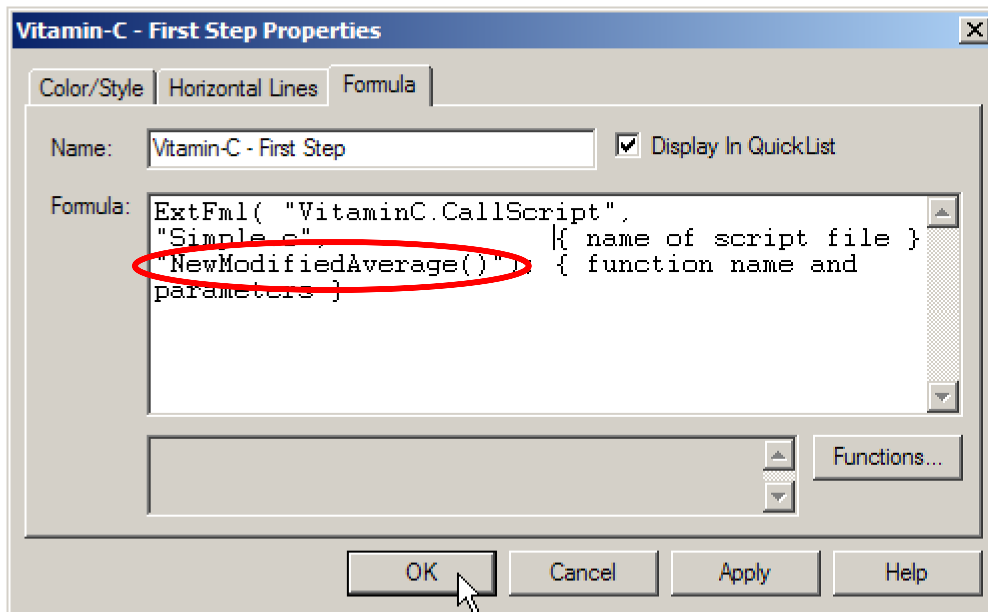
The modified code to do this is as follows:

```
void NewModifiedAverage()
{
  float lastvalue=0;
```

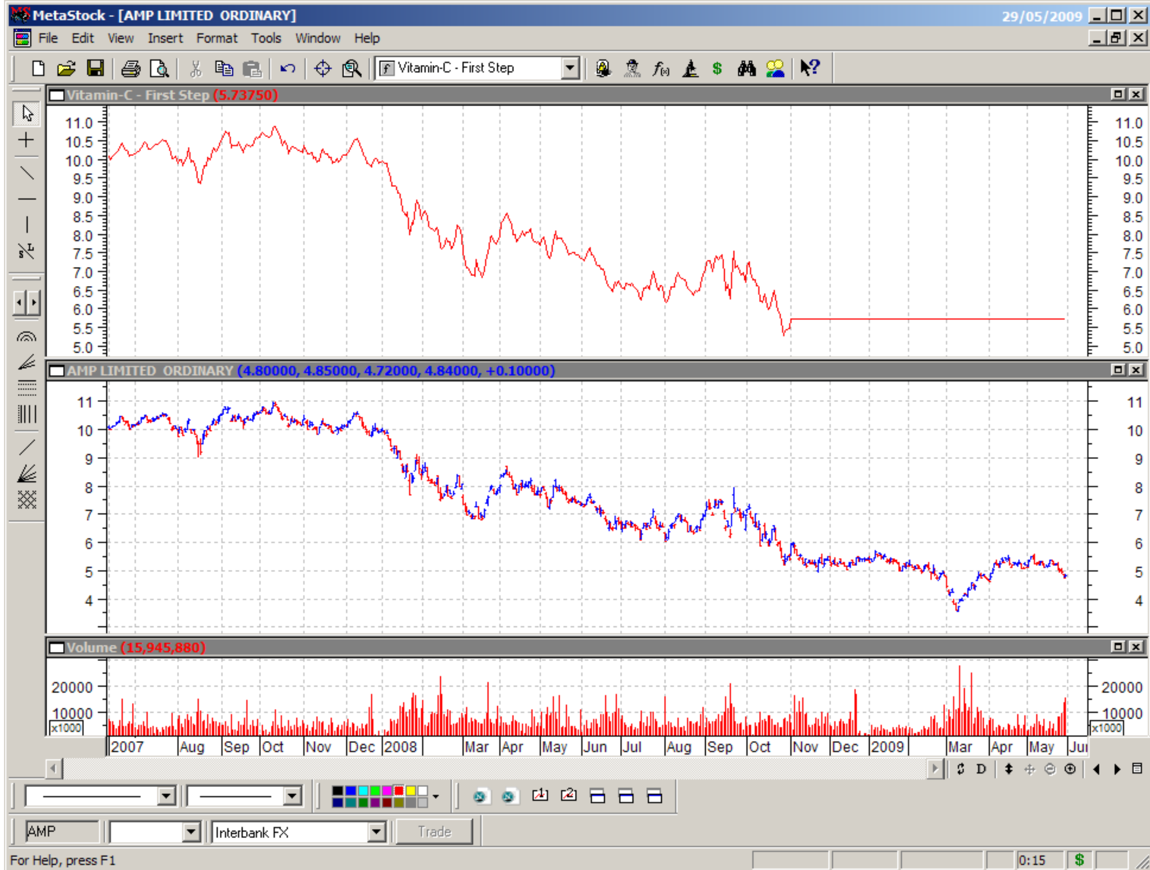
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
  for(int i=0;i<BarCount;i++)
  {
    if(GetDate(i) > 20081103)
      Result[i]=lastvalue;
    else
    {
      Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
      lastvalue=Result[i];
    }
  }
}
```

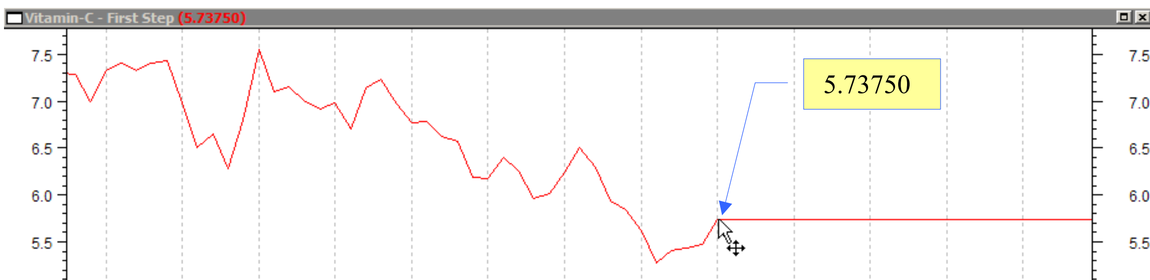
Add the above code to the Vitamin-C script, save it and then update the indicator with new function name as follows.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Lets taker a closer look at the indicator. Where the mouse pointer is, is indeed the 3rd-Nov-2008 and the value of 5.73750 at that date is maintained for the rest of the indicator, which means that the Vitamin-C script is working as expected.



This last example displays one very important concept, which cannot be done easily with the MetaStock formula language (MSFL). The concept of a flag variable, which can be set at a particular date or dates is what gives Vitamin-C its flexibility in creating many different types of indicator that would be difficult if not impossible to do using the MSFL.

Corollary

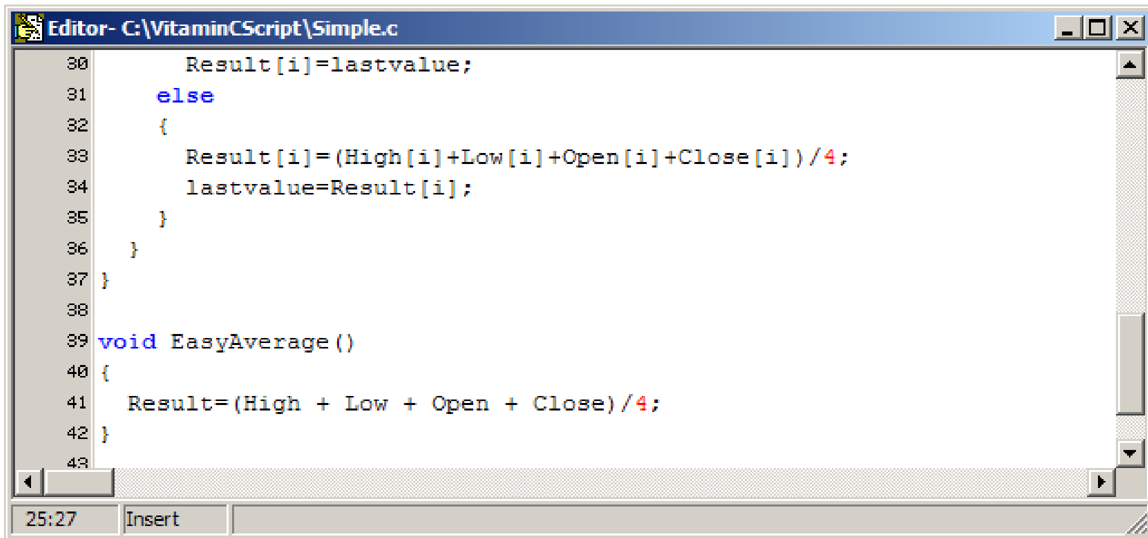
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

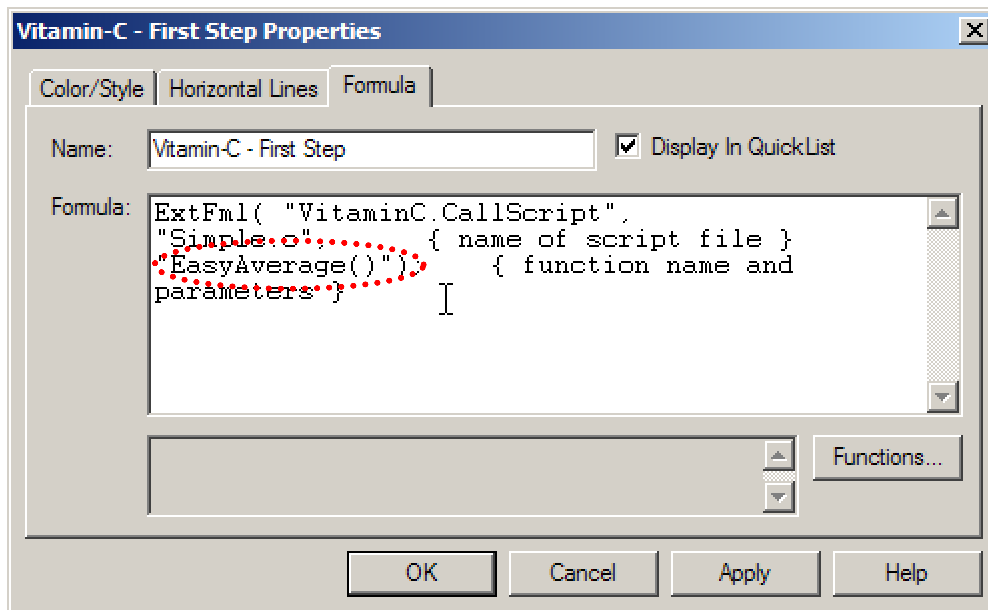
As I have stated earlier, Vitamin-C allows you access to each element in an array variable by using subscripting. You can also treat arrays as a single object in much the same way that MetaStock does. To prove this we will rewrite the original averaging code using array objects.

Create a new function called 'EasyAverage' and paste it into the 'Simple.c' script file. Note the absence of any subscripts on any of the predefined array variables.

```
void EasyAverage()
{
  Result=(High + Low + Open + Close)/4;
}
```

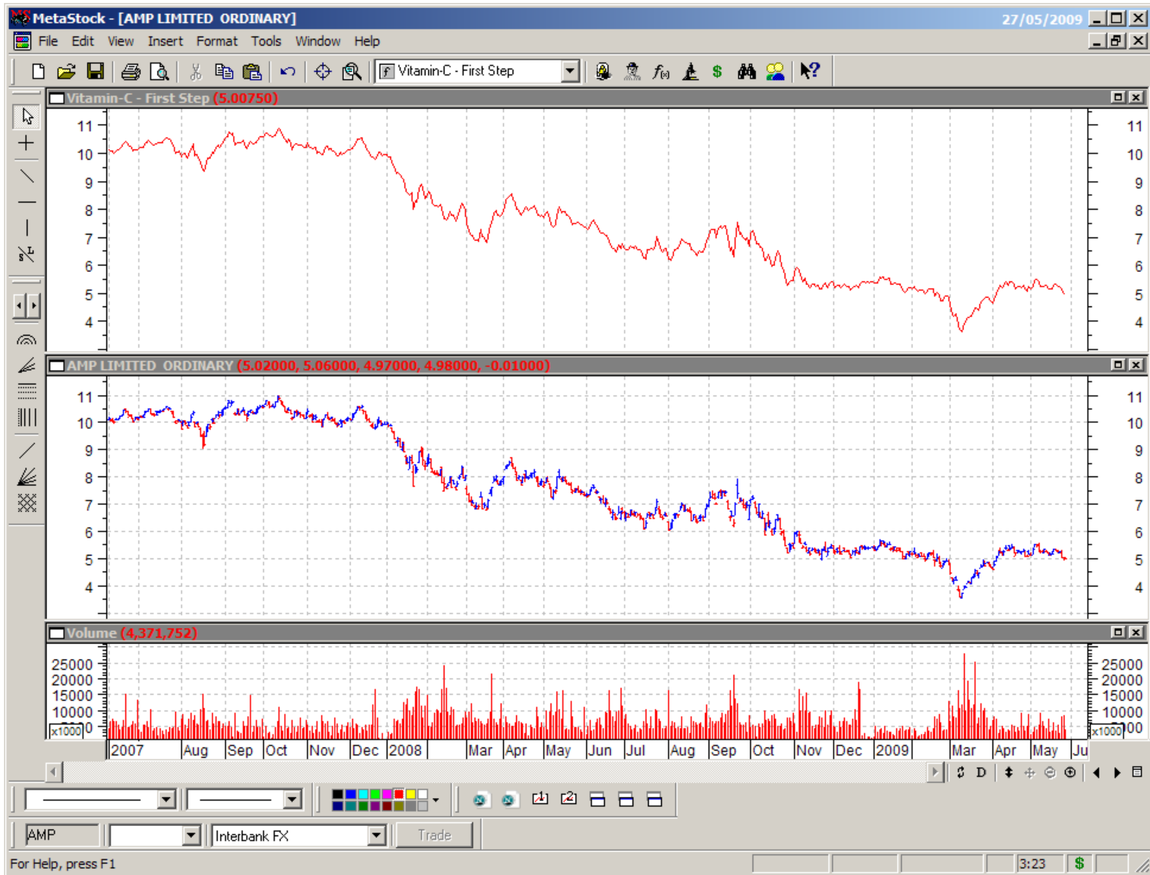
Add the code above to the Vitamin-C script, save it and then update the indicator with new function name as follows.





Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

What do you see ?



OK, the results should be identical to the 'Average' function, which should be identical to the MetaStock code.

Exercise

As an exercise you can create three different version of the same average indicator using

- the MetaStock formula language,
- using Vitamin-C and array subscripting
- using Vitamin-C and array objects.

Then you can overlay the three indicators on top of each other and see if there are any discrepancies. There shouldn't be of course!

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Your Fourth Step

Getting a little bit more serious !

OK it's time to get serious now and do something useful. We will get back to one of our original problems of how to code a simple time stop. Later on we will

Coding a Time Stop

The following code stub represents an example of C++ code that could be used to implement a time stop that exits when the number of bars exceeds a certain limit which we shall call _Nbars. Unlike our attempt at using the MSFL to code a profit stop, the C++ coding actually filters out bogus entry triggers because it is able to set a boolean variable flag at a particular date/time and maintain this value until it is ready to be updated again. This is only possible by being able to iterate through all of the values in the price arrays.

A boolean variable is a variable that can only take one of two values, which are either 'true' or 'false'.

Now lets get down to business and code up a time stop, which is a little bit simpler than the profit stop problem discussed in the very early part of this manual but a bit more complicated than the previous averaging functions. The equivalent Vitamin-C script can be written as follows. Don't worry if this looks alien to you but as you can see, the way things are coded is very different to the MSFL that you are familiar with. This function can be used to stop out a trade after _Nbars. The Vitamin-C function accepts one argument, which is the number of bars.

Functions can accept arguments from the calling program which are essentially constant values of some kind. Within the function block itself these arguments are accessed through the use of parameters, which are a special kind of variable that holds the argument that is being passed to the function.

```
void TimeStop(int _Nbars)
{
  bool InTrade=false;        // boolean variable used for trade flag
  int bars=0;                // variable used to keep track of the bar count

  for(int i=0;i<BarCount;i++,bars++) // loop for all bars
  {
    Result[i]=0;             // initialize result for each bar
    if(!InTrade && User1[i]>0) // check if not in a trade and valid entry trigger
    {
      InTrade=true;          // set the flag
      Result[i]=1;           // mark as entry
      bars=0;                // reset the bar count
    }

    if(InTrade)              // if in the trade do the check
    {
      if(bars >= _Nbars)     // check for a time stop condition
      {
        Result[i]=1;         // threshold reached so mark it as a valid exit condition
        InTrade=false;       // reset the flag
      }
    }
  }
}
```

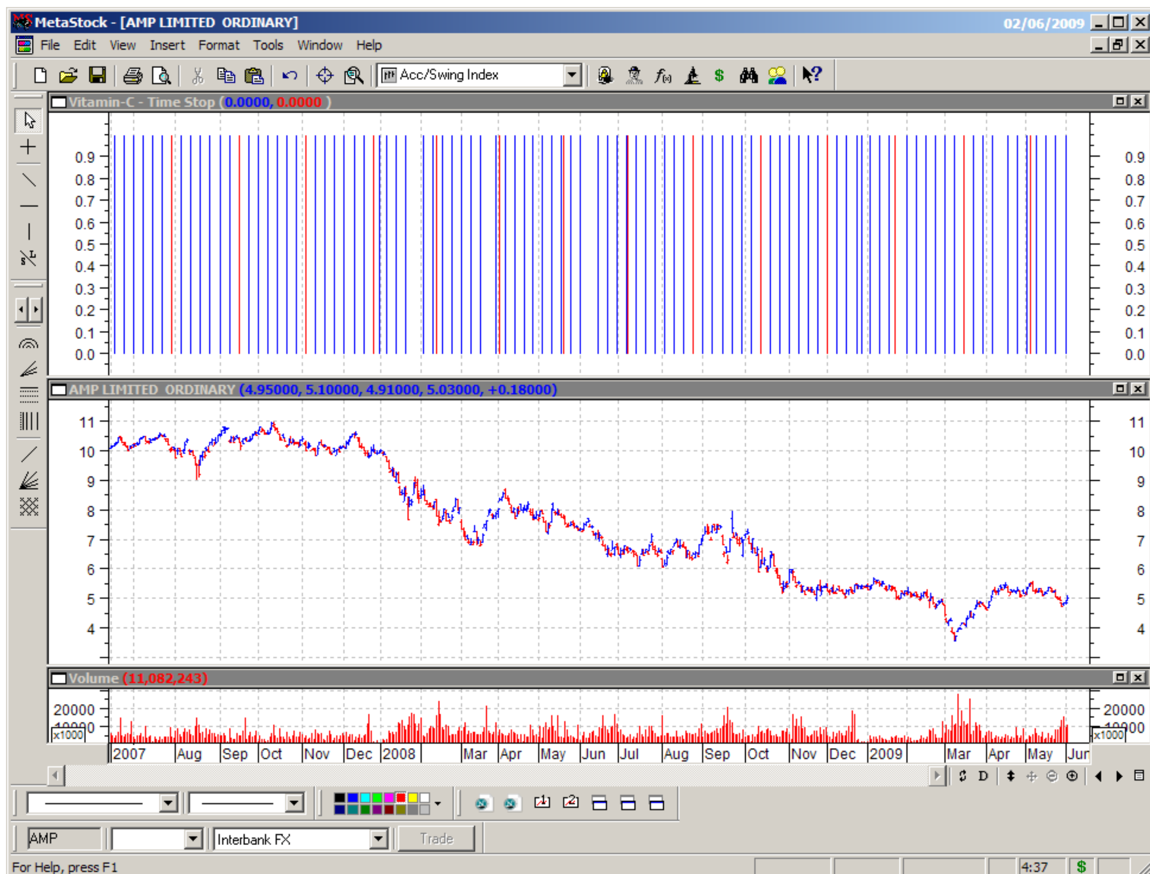
Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Click on 'New' from the file menu and copying and paste the code stub above into the edit window and save it as 'TimeStop.c '

The following MSFL code stub is used to call the above function from your MetaStock code with a 30 bars time stop using an arbitrary weekly trigger.

```
EntryTrigger:=(DayOfWeek()=1);


ExitTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",               { name of script file }
"TimeStop(30)",             { function name and parameter }
EntryTrigger);


EntryTrigger;               { display entry trigger on chart }
ExitTrigger;                { display exit trigger on chart }
```

Create a new indicator in MetaStock and copy and paste the code above into it. The indicator will display all of the Entry and Exit Triggers. Overlaying the indicator on a chart should look like the following.



I have purposely changed the style and colors of the Entry and Exit triggers so that it is easy to distinguish one from the other. However there is still one problem in that the Entry Trigger formula is displaying all entry triggers including the triggers that have been ignored. How can we distinguish the actual entry triggers from the triggers that have been ignored ?

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

What we can do is to encode the Result array so that different values represent different combinations of the entry and exit signals. Because we are representing two signals there are four possible combinations according to the following table:

Value of Result[i]	Description
0	No entry or Exit trigger
1	Entry Trigger only
2	Exit Trigger Only
3	Entry and Exit Trigger at the same time

Now how do we modify the code to achieve this ??

Quite simply we assign '1' to Result[i] for every actual entry trigger we detect and just add the value of '2' to Result[i] every time we detect a valid time stop exit trigger, otherwise every other value in the Result array should be set to '0'. The modified code will look like the following with the changes marked in red:

```
void TimeStop(int _Nbars)
{
  bool InTrade=false;        // boolean variable used for trade flag
  int bars=0;                // variable used to keep track of the bar count

  for(int i=0;i<BarCount;i++,bars++) // loop for all bars
  {
    Result[i]=0;             // initialize result for each bar
    if(!InTrade && User1[i]>0) // check if not in a trade and valid entry trigger
    {
      InTrade=true;          // set the flag
      Result[i]=1;           // mark as entry
      bars=0;                // reset the bar count
    }

    if(InTrade)              // if in the trade do the check
    {
      if(bars >= _Nbars)     // check for a time stop condition
      {
        Result[i]+=2;        // threshold reached so mark it as a valid exit condition
        InTrade=false;       // reset the flag
      }
    }
  }
}
```

The 'C/C++' language has a number of short cuts in the form of compound statements, which makes coding a lot more efficient. For example the line in the above code which adds '2' to the Result;

```
Result[i]+=2;
```

is short for:

```
Result[i] = Result[i] + 2;
```

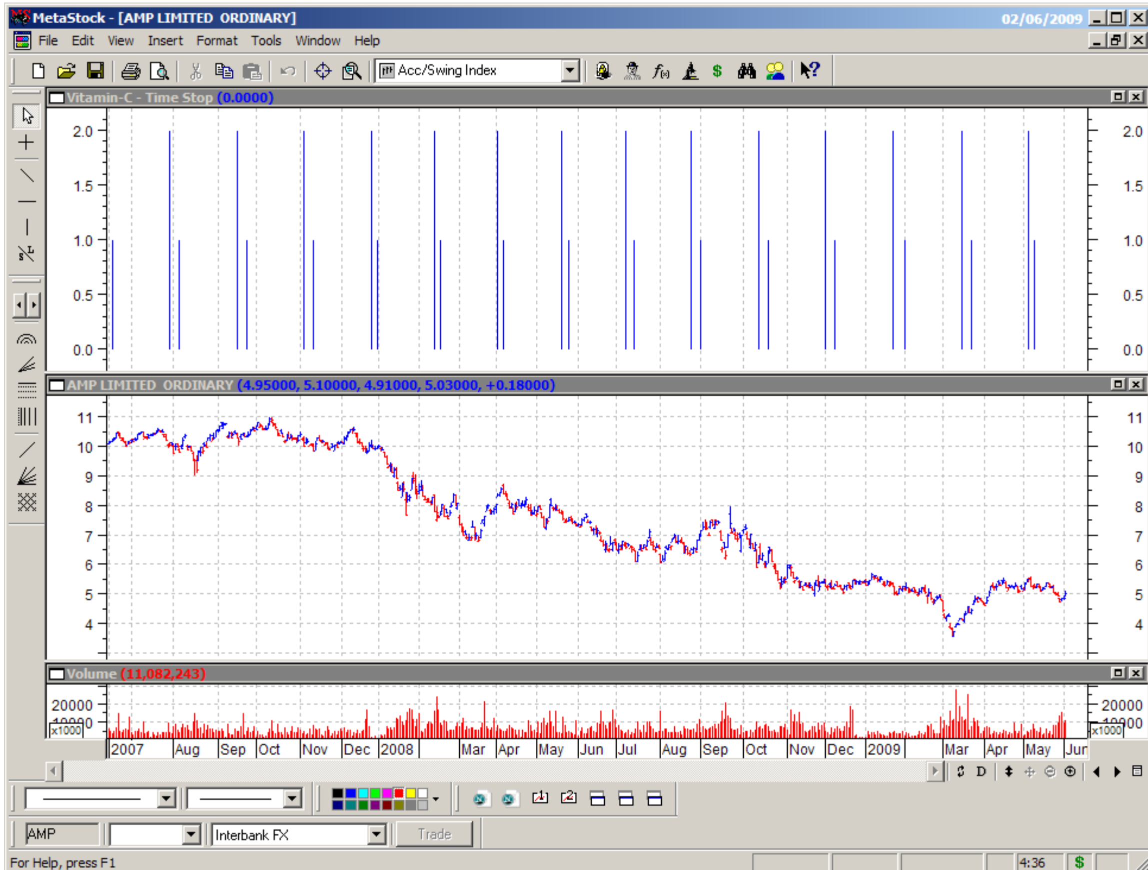
We need to change the indicator code to the following:

```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
"TimeStop.c",                { name of script file }
"TimeStop(30)",              { function name and parameter }
EntryTrigger);

EncodedTrigger;              { display the encoded entry/exit triggers }
```

Overlaying the indicator on the chart will give us something that looks like the following:



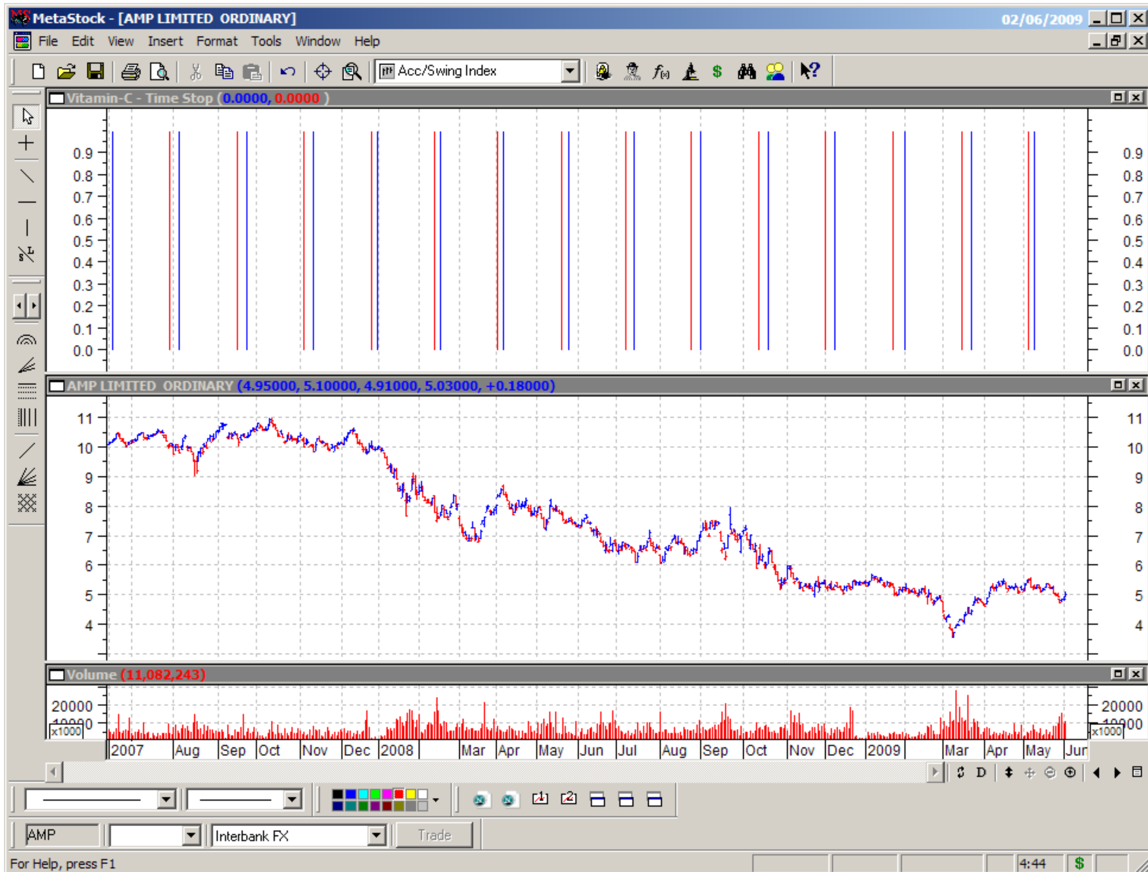
Now we can see the valid entry triggers (bars with a height of '1') and the corresponding timed out exit triggers (bars with a height of '2'). Obviously there are no conditions where entry and exit condition coexist together which would display a bar of height '3'. You are probably wondering why would that situation would happen in the first place ! Wouldn't an exit condition always precede an entry condition by at least _Nbars ? Well quite simply this scenario can occur when you exit a trade on a bar but there is another entry signal on the same bar.

Now lets get a bit more fancy with this example and decode the encoded trigger returned by our TimeStop() function. This will allow us to display the indicator with different colors to represent both actual entry and exit triggers. Use the following MSFL code to do this:

```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",                { name of script file }
```

```
"TimeStop(30)",              { function name and parameter }
EntryTrigger);

ActualEntryTrigger:=(EncodedTrigger=1) OR (EncodedTrigger=3);
ActualExitTrigger:=(EncodedTrigger=2) OR (EncodedTrigger=3);

ActualEntryTrigger;          { display actual entry trigger on chart }
ActualExitTrigger;           { display actual exit trigger on chart }
```

Blue bars represent valid entry conditions and red bars represent valid exit conditions. Of course if there are exit triggers on the same day as valid entry triggers then the red bar will blot out the blue bar so it would be a useful exercise to split the indicator up into two indicators for both entry and exit conditions respectively and display them on a separate chart.



Creating a MetaStock Time Stop Expert

Before moving on, it should be noted that Vitamin-C scripts are not just limited to indicators. In actual fact a Vitamin-C script can be called anywhere the MetaStock Formula language is used. For example you can call a Vitamin-C script(s) from the MS Explorer, built in system tester and Expert for example. The Expert tool is what we want to focus on now so let’s create an expert for our TimeStop function, which displays both the actual entry and exit signals.

In MetaStock create an Expert and call it “Vitamin-C (Time Stop)”.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

In the 'Symbols' tab create a new symbol called 'Entry Trigger' and add the following code. In the 'Graphic' tab select a blue 'Buy Arrow' and label it as "Entry Trigger".

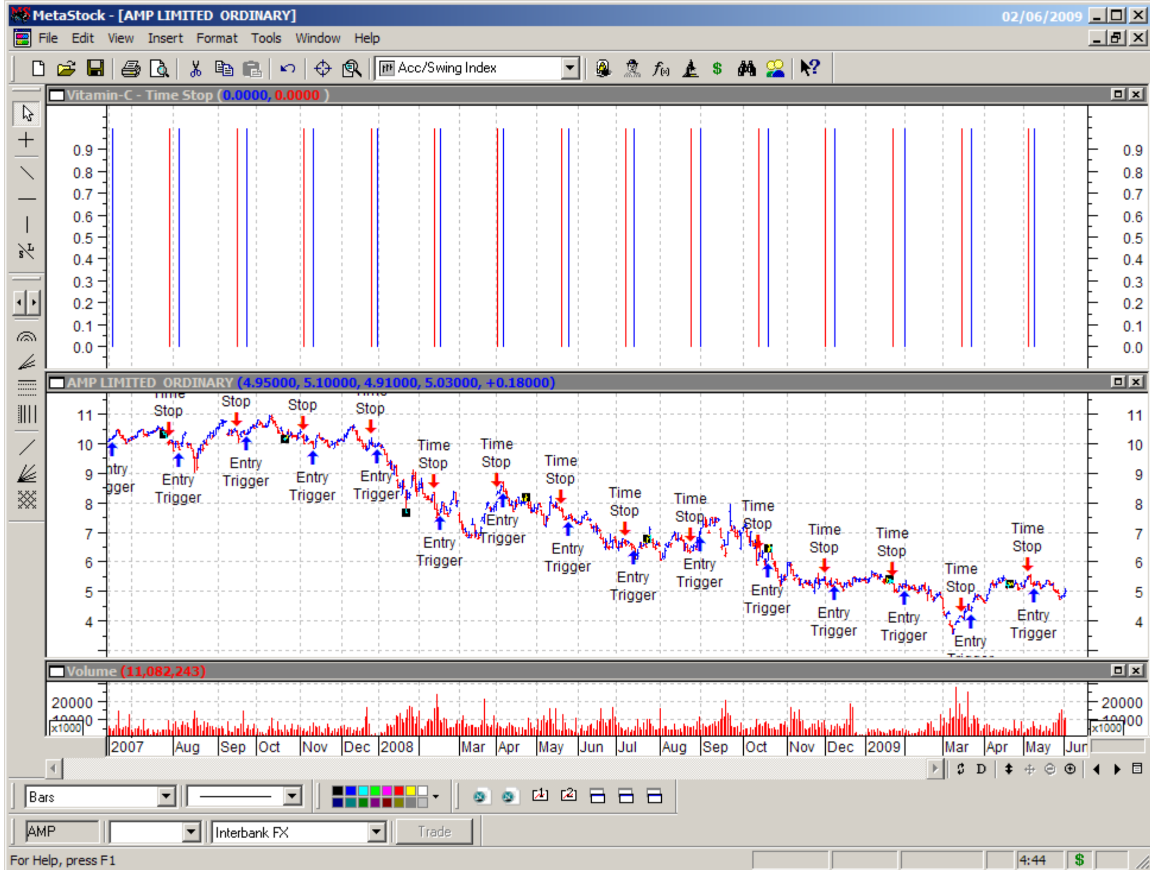
```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",        { name of script file }
"TimeStop(30)",      { Vitamin-C function name and parameter }
EntryTrigger);


EncodedTrigger =1 OR EncodedTrigger =3;
```

In the Symbols tab create a new symbol called 'Exit Trigger' and add the following code. In the 'Graphic' tab select a red 'Sell Arrow' and label it as "Time Stop".

```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",        { name of script file }
"TimeStop(30)",      { Vitamin-C function name and parameter }
EntryTrigger);


EncodedTrigger =2 OR EncodedTrigger =3;
```

Now attach the expert to the chart.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Lets expand the display for a sec and have a look at it in more detail. Note how the expert signals correspond with the indicators, which tells us that everything is working well.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Building a simple Profit Stop Indicator Using Vitamin-C

We have managed to hold off coding the ubiquitous Profit Stop until now, but we needed to cover some basics before we dived in at the deep end. Finally we can get down to business and work on the elusive profit stop that everyone has been waiting for years to code in MetaStock !!

We will design and code a profit stop that simply enters in at the opening price and exits at the closing price when the closing price exceeds the opening price at entry by a certain percentage as specified by a function parameter which can be changed in the call to this function from MetaStock. This will form the foundations of a more elaborate profit stop function but at the same time will illustrate some of the power and flexibility of the C++ language in writing these kinds of functions.

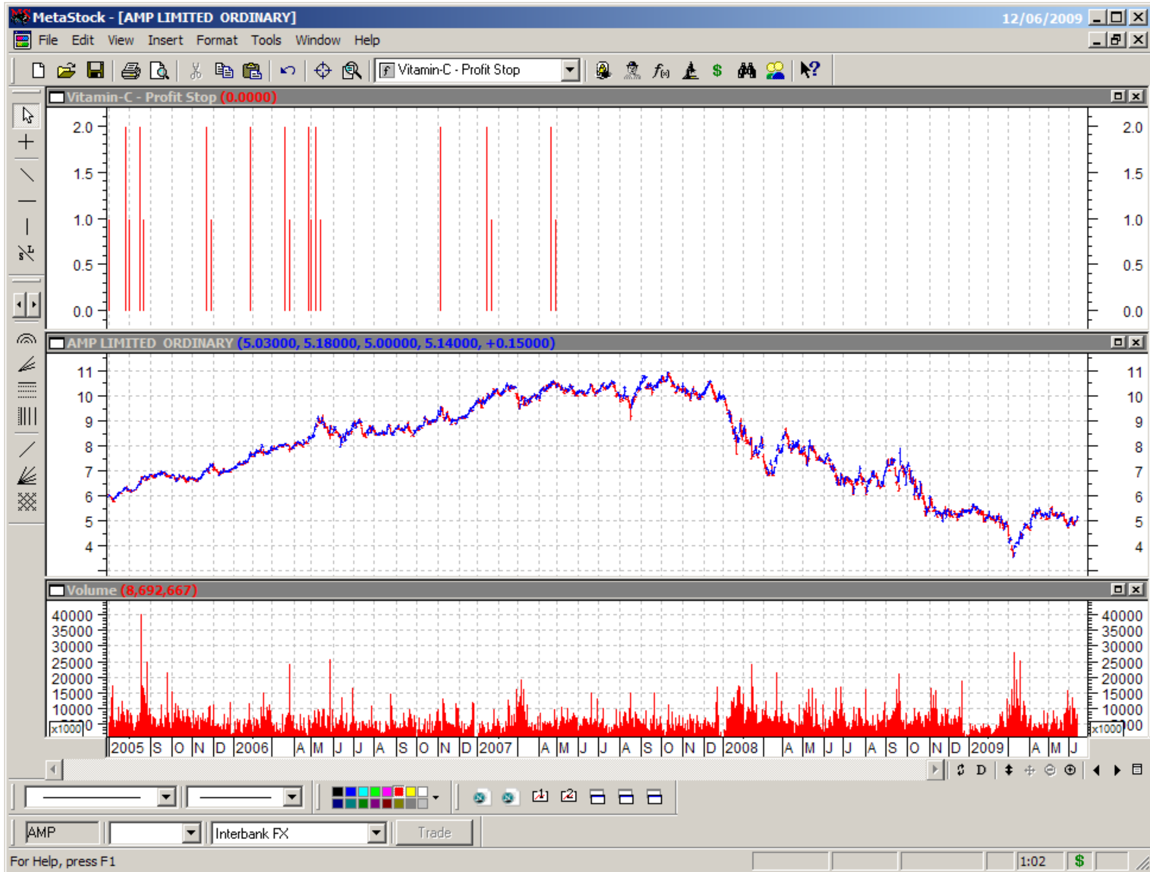
```
int ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;            // boolean variable used for trade flag
  float EntryPrice=0;            // variable used to store the entry price on trade entry
  float ProfitThreshold;         // variable used to hold the profit threshold price

  for(int i=0;i<BarCount;i++)   // loop for all bars
  {
    Result[i]=0;                 // initialize result for each bar

    if(!InTrade && User1[i]>0)   // check if not in a trade and valid entry trigger
    {
      InTrade=true;              // set the flag
      EntryPrice=Open[i];        // set the entry price
                                 // calculate the profit threshold price
      ProfitThreshold=EntryPrice*(1 + _PercentThreshold/100);
      Result[i]=1;               // mark a valid entry condition
    }

    if(InTrade)                  // if in the trade do the check
    {
      if(Close[i] >= ProfitThreshold)    // check for a profit stop condition
      {
       Result[i]+=2;             // threshold reached so mark a valid exit condition
        InTrade=false;          // reset the flag
      }
    }
  }
}
```

Create an indicator with the following MS code.

```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"ProfitStop.c",     { name of script file }
"ProfitStop(5)",    { function name and profit threshold argument }
EntryTrigger        { User defined entry trigger }
);


EncodedTrigger;
```

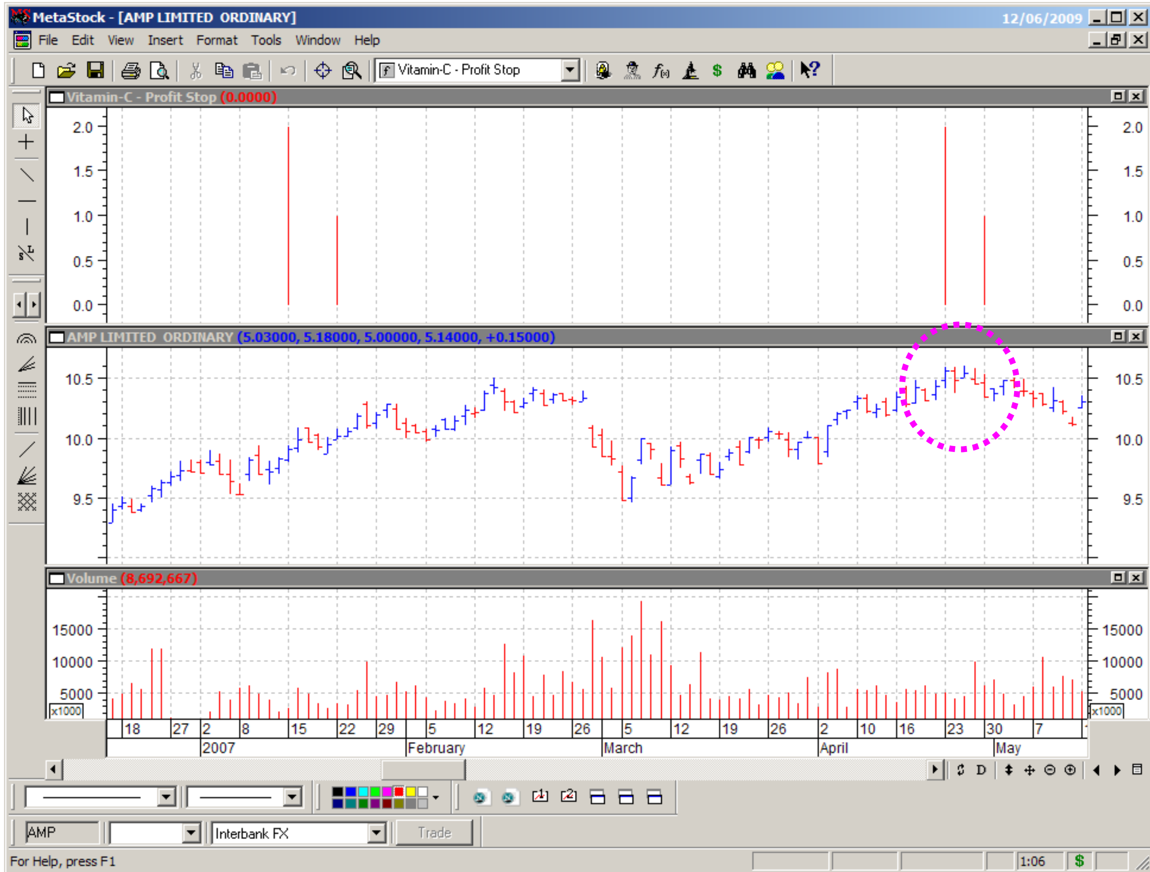
Apply the indicator to the chart.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



As you can there were quite a few exit triggers up until the bull market collapsed when the bears took over at the end of 2008. Now lets take a closer look at the chart to see if the stop is working as expected. We have expanded the chart around the last entry-exit pair as follows.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Date	Status	Price
22-Jan-2007	Entry	Open = 9.98088
23-April-2007	Exit	Close = 10.5584

Profit Gain = (10.5584 – 9.98088)/9.98088 * 100 = 5.78%

Which meets the criteria for a profit stop of 5% or more.

Now lets get a bit more critical. How can we tell what the profit gain was for the preceding bars before the exit bar just so that we can put our mind to rest that our Vitamin-C script is actually calculating things properly ? The simplest way would be to display the profit data for each bar between and including the entry and exit bar. We can do this using the built in equivalent of the 'C' printf() function which is designed to display information to a command consol or in the case of Vitamin-C, a dedicated debugging consol which is just a fancy name for a window.

In Vitamin-C the equivalent of the 'C' printf statement is called dprint and instead of displaying data to the standard consol, when called, it displays data in a dedicated debug log window within the Vitamin-C environment.

In our profit stop function will add the following code marked in grey background color. Don't worry if it looks cryptic to you at the moment, as we will discuss it in more depth later on. To put your mind at ease the C++ printf function has been described in depth in many books on the C++ language. It is usually the first C++ function that is discussed in any textbook on the C++ language.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
int ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;            // boolean variable used for trade flag
  float EntryPrice=0;            // variable used to store the entry price on trade entry
  float ProfitThreshold;         // variable used to hold the profit threshold price

  for(int i=0;i<BarCount;i++)  // loop for all bars
  {
    Result[i]=0;                 // initialize result for each bar

    if(!InTrade && User1[i]>0)   // check if not in a trade and valid entry trigger
    {
      InTrade=true;              // set the flag
      EntryPrice=Open[i];        // set the entry price
                                 // calculate the profit threshold price
      ProfitThreshold=EntryPrice*(1 + _PercentThreshold/100);
      Result[i]=1;               // mark a valid entry condition
      dprintf("Date=%d, Opening Price=%f\n",GetDate(i),(double)EntryPrice);
    }

    if(InTrade)                  // if in the trade do the check
    {
      dprintf("Date=%d, Closing Price=%f, Profit=%f\n"
,GetDate(i),(double)Close[i],(double)(Close[i]-EntryPrice)/EntryPrice*100);
      if(Close[i] >= ProfitThreshold)    // check for a profit stop condition
      {
        dprintf("*******************************************************\n");
       Result[i]+=2;            // threshold reached so mark a valid exit condition
        InTrade=false;          // reset the flag
      }
    }
  }
}
```

The first call to dprintf displays the entry conditions such as the date and opening price in the debug log, whilst the next call displays each closing price, along with the date and the actual profit gain relative to the entry price or opening price in our case. The third call to dprintf just prints a line of asterix's so it is easy to visually separate the next set of trade data.

If you run the code above you will note that the debug log window appears within the Vitamin-C window along with all of the debug data written via the dprintf function..

If we focus on the exit date of 23-04-2007 you can see that all of the preceding bars before the last bar fall below the profit threshold of 5%, and the last bar is indeed the one.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Debug Consol

Clear Log | Append | Save Debug

```
439 Date=20070410, Closing Price=10.336992, Profit=3.567986
440 Date=20070411, Closing Price=10.221495, Profit=2.410797
441 Date=20070412, Closing Price=10.240745, Profit=2.603665
442 Date=20070413, Closing Price=10.192621, Profit=2.121510
443 Date=20070416, Closing Price=10.346617, Profit=3.664416
444 Date=20070417, Closing Price=10.279244, Profit=2.989401
445 Date=20070418, Closing Price=10.423615, Profit=4.435878
446 Date=20070419, Closing Price=10.317743, Profit=3.375128
447 Date=20070420, Closing Price=10.433240, Profit=4.532307
448 Date=20070423, Closing Price=10.558362, Profit=5.785925
449 ***************************************************************************
450 Date=20070430, Opening Price=10.462114
451 Date=20070430, Closing Price=10.346617, Profit=-1.103960
452 Date=20070501, Closing Price=10.375491, Profit=-0.827970
453 Date=20070502, Closing Price=10.481364, Profit=0.183996
454 Date=20070503, Closing Price=10.433240, Profit=-0.275990
455 Date=20070504, Closing Price=10.394741, Profit=-0.643974
456 Date=20070507, Closing Price=10.336992, Profit=-1.195954
457 Date=20070508, Closing Price=10.279244, Profit=-1.747925
458 Date=20070509, Closing Price=10.317743, Profit=-1.379941
```

Spicing up the Profit Stop.

Out first attempt at the profit stop forced as to enter in at the opening price and exit when the closing price exceeded a certain percentile profit threshold.

What happens if we want to an arbitrary price for the entry(reference) and exit(threshold) price ??

Lets modify the previous code to add this additional functionality. We need two more user arrays to hold our entry and exit prices. We will use the User2 array for the entry or reference price and User3 array for the exit or threshold price. The modifications are marked with grey background in the following code.

```
int ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;            // boolean variable used for trade flag
  float EntryPrice=0;            // variable used to store the entry price on trade entry
  float ProfitThreshold;         // variable used to hold the profit threshold price

  for(int i=0;i<BarCount;i++)  // loop for all bars
  {
    Result[i]=0;                 // initialize result for each bar

    if(!InTrade && User1[i]>0)   // check if not in a trade and valid entry trigger
    {
      InTrade=true;              // set the flag
      EntryPrice=Open[i];        // set the entry price
                                 // calculate the profit threshold price

      ProfitThreshold=User2[i]*(1 + _PercentThreshold/100);
      Result[i]=1;               // mark a valid entry condition
    }

    if(InTrade)                  // if in the trade do the check
    {
      if(User3[i] >= ProfitThreshold)     // check for a profit stop condition
      {
       Result[i]+=2;             // threshold reached so mark a valid exit condition
        InTrade=false;          // reset the flag
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
      }
    }
  }
}
```

To call this from MetaStock we need to use the variant of the CallScript function that supports three array parameters:

```
EntryTrigger:=(DayOfWeek()=1);
EntryPrice:=Open;  { entry or reference price }
ExitPrice:=Close;  { exit or threshold price }

EncodedTrigger:=ExtFml( "VitaminC.CallScript3",
"ProfitStop.c",     { name of script file }
"ProfitStop(5)",    { function name and profit threshold argument }
EntryTrigger,       { User defined entry trigger }
EntryPrice,         { User defined entry or reference price }
ExitPrice,          { User defined exit or threshold price }
);

EncodedTrigger;
```

Handling the Short side of the Market

So far we have dealt with the situation of going long but what about situations where the trader wants to go short ? We will modify our profit stop in the last section such that when a negative value of the *PercentThreshold* is passed to the function then it assumes that we want to profit stop on the short side. The modified code for this is as follows:

```
int ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;             // boolean variable used for trade flag
  float EntryPrice=0;             // variable used to store the entry price on trade entry
  float ProfitThreshold;          // variable used to hold the profit threshold price

  for(int i=0;i<BarCount;i++)   // loop for all bars
  {
    Result[i]=0;                  // initialize result for each bar

    if(!InTrade && User1[i]>0)    // check if not in a trade and valid entry trigger
    {
      InTrade=true;               // set the flag
      EntryPrice=Open[i];         // set the entry price
                                  // calculate the profit threshold price
      ProfitThreshold=User2[i]*(1 + _PercentThreshold/100);
      Result[i]=1;                // mark a valid entry condition
    }

    if(InTrade)                   // if in the trade do the check
    {
      if(_PercentThreshold >=0)   // long side
      {
        if(User3[i] >= ProfitThreshold)) // check for a profit stop condition on the long
side
        {
          Result[i]+=2;           // threshold reached so mark a valid exit condition
          InTrade=false;          // reset the flag
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
        }
      }
      else                        // short side
      {
        // check for a profit stop condition on the short side
        if(User3[i] <= ProfitThreshold))
        {
          Result[i]+=2;           // threshold reached so mark a valid exit condition
          InTrade=false;          // reset the flag
        }
      }
    }
  }
}
```

We shall call this function from MetaStock to check for 5% profit stops on the short side.

```
EntryTrigger:=(DayOfWeek()=1);
EntryPrice:=Open;  { entry or reference price }
ExitPrice:=Close;  { exit or threshold price }

EncodedTrigger:=ExtFml( "VitaminC.CallScript3",
"ProfitStop.c",    { name of script file }
"ProfitStop(-5)",  { function name and profit threshold argument (5% on the
short side) }
EntryTrigger,      { User defined entry trigger }
EntryPrice,        { User defined entry or reference price }
ExitPrice,         { User defined exit or threshold price }
);

EncodedTrigger;
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Building a Trailing Stop function with Vitamin-C

In this section we plan to build a trailing stop function using Vitamin-C. In fact we plan on duplicating the functionality of the Fast Trailing Stop function built into the TradeSim/MetaStock plugin. In this way you will be able to customize or modify it to the way you want it. Details of the Fast Trailing Stop can be found in the TradeSim document library that is installed with TradeSim or available from the 'Articles' section of our website.

http://www.compuvision.com.au/Articles.htm

We digress a bit here and to familiarize ourselves with the TradeSim trailing stop function.

The TradeSim Trailing Stop Function Description

The external trailing stop function available in the TradeSim plugin to MetaStock has the following syntax:

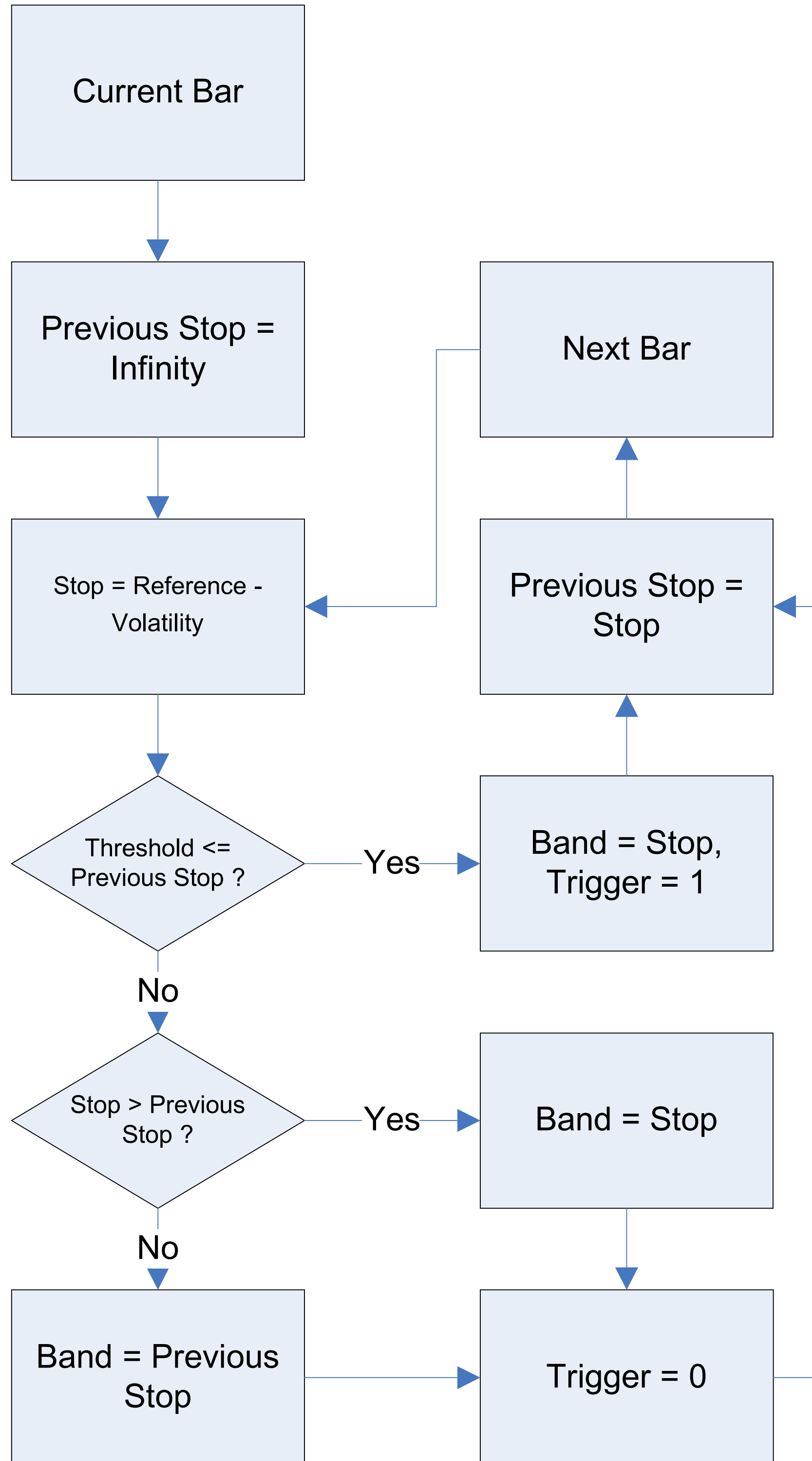
```
ExtFml("TradeSim.TrailingStop",     { 1. Name of external function }
        Mode,                       { 2. Mode}
        TradePosition,              { 3. Trade Position Type }
        VolatilityFunction,         { 4. Volatility Function Array }
        RefPoint                    { 5. Reference Point Array }
        ThresholdPoint              { 6. Threshold Point Array }
};
```

Parameter	Description	Allowable Values
1	The name of the external function along with the name of the external DLL that contains it	"TradeSim.TrailingStop "
2	The Mode parameter determines whether the function is used to generate a trailing stop band indicator or a binary stop trigger function.	BAND – displays a Trailing Stop band indicator. TRIGGER – generates a binary trigger where a value of '1' indicates a stop trigger.
3	The TradePosition parameter specifies whether the trade is on the long or short side.	LONG or SHORT
4	The VolatilityFunction parameter is a user defined volatility function for determining the type of trailing stop.	For example 3*ATR(10) gives a 10 bar ATR trailing stop with an average range constant of 3.
5	The RefPoint parameter determines the point or price where the trailing stop is calculated. It is typically set to the closing price however it could be set to HIGH for the long side or to LOW for the short side.	CLOSE, OPEN, HIGH, LOW or any arbitrary data array.
6	The ThresholdPoint parameter determines the point at which the stop band has been breached and the the trailing stop band is reset.	LOW, CLOSE, HIGH, or any arbitrary data array. Typically you would use LOW for the long side and HIGH for the side short side.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Trailing Stop Algorithm on the Long Side

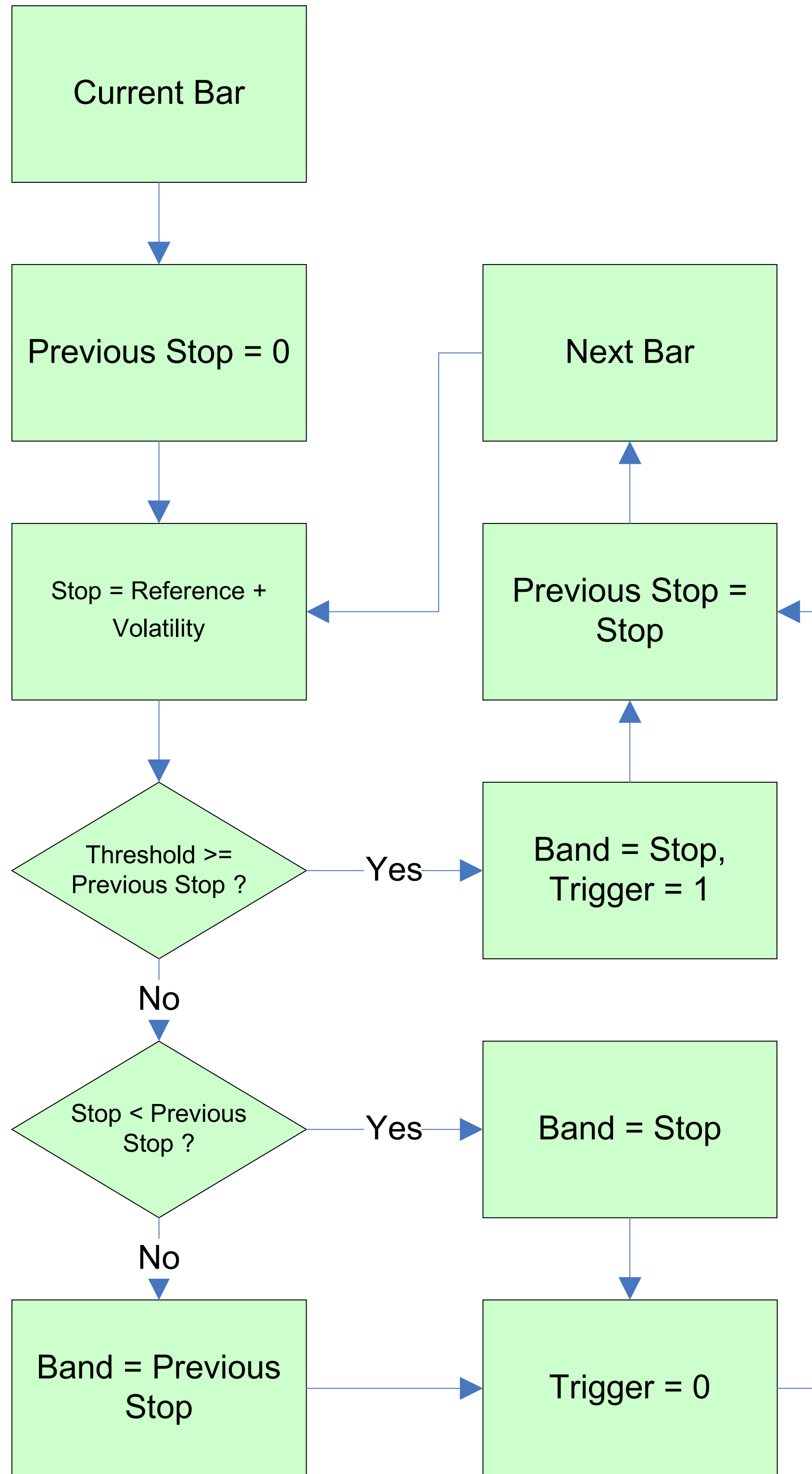
The following flowchart illustrates the algorithm used to compute the trailing stop on the long side.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Trailing Stop Algorithm on the Short Side

The following flowchart illustrates the algorithm used to compute the trailing stop on the short side.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

In Vitamin-C the calling convention in MetaStock would be re-written as follows:

```
ExtFml("VitaminC.CallScript3",     { 1. Name of external function }
"TrailStop.c"                      { 2. name of script file }
"TrailingStop(_Mode,_Position)"    { 3. function name and parameters }
VolatilityFunction,                { 4. Volatility Function Array }
RefPoint                           { 5. Reference Point Array }
ThresholdPoint                     { 6. Threshold Point Array }
};
```

With some slight modifications, the Vitamin-C script used to implement the Trailing Stop below is essentially the same code we used to code the Trailing Stop in the TradeSim.DLL plug-in. This is one of the advantages of conforming to a code standard such as 'C'. It more or less guarantees portability between different platforms. In this case we were able to take code that was written for a DLL using the Borland C++ environment and port it across to run under Vitamin-C.

```
enum ModeEnum {
  BAND,
  TRIGGER
};

enum PositionEnum {
  LONG,
  SHORT
};

void TrailingStop(ModeEnum _Mode,PositionEnum _Position)
{
  float Band=0;
  float Prev=0;
  float Stop;

  if(_Position==SHORT)
    Prev=1000000;            // need this to initialize properly for short condition

  for(int i=0;i<BarCount;i++)
  {
    if(_Position==LONG)      // Long Side
    {
      Stop=User2[i]-User1[i];
      if(User3[i]<=Prev)
      {
        Band=Stop;
        if(_Mode==BAND)      // Band
          Result[i]=Band;
        else                 // Trigger
          Result[i]=1;
      }
      else
      {
        if(Stop>Prev)        // check for new high
          Band=Stop;         // move the trailing stop band
        else
          Band=Prev;         // keep it the same as before

        if(_Mode==BAND)      // Band
          Result[i]=Band;
        else                 // Trigger
          Result[i]=0;
      }
    }
    else                     // Short side
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
    {
      Stop=User2[i]+User1[i];
      if(User3[i]>=Prev)
      {
        Band=Stop;
        if(_Mode==BAND)       // Band
          Result[i]=Band;
        else                  // Trigger
          Result[i]=1;
      }
      else
      {
        if(Stop<Prev)         // check for new low
          Band=Stop;          // move the trailing stop band
        else
          Band=Prev;          // keep it the same as before

        if(_Mode==BAND)       // Band
          Result[i]=Band;
        else                  // Trigger
          Result[i]=0;
      }
    }

    Prev=Band;

  }
}
```

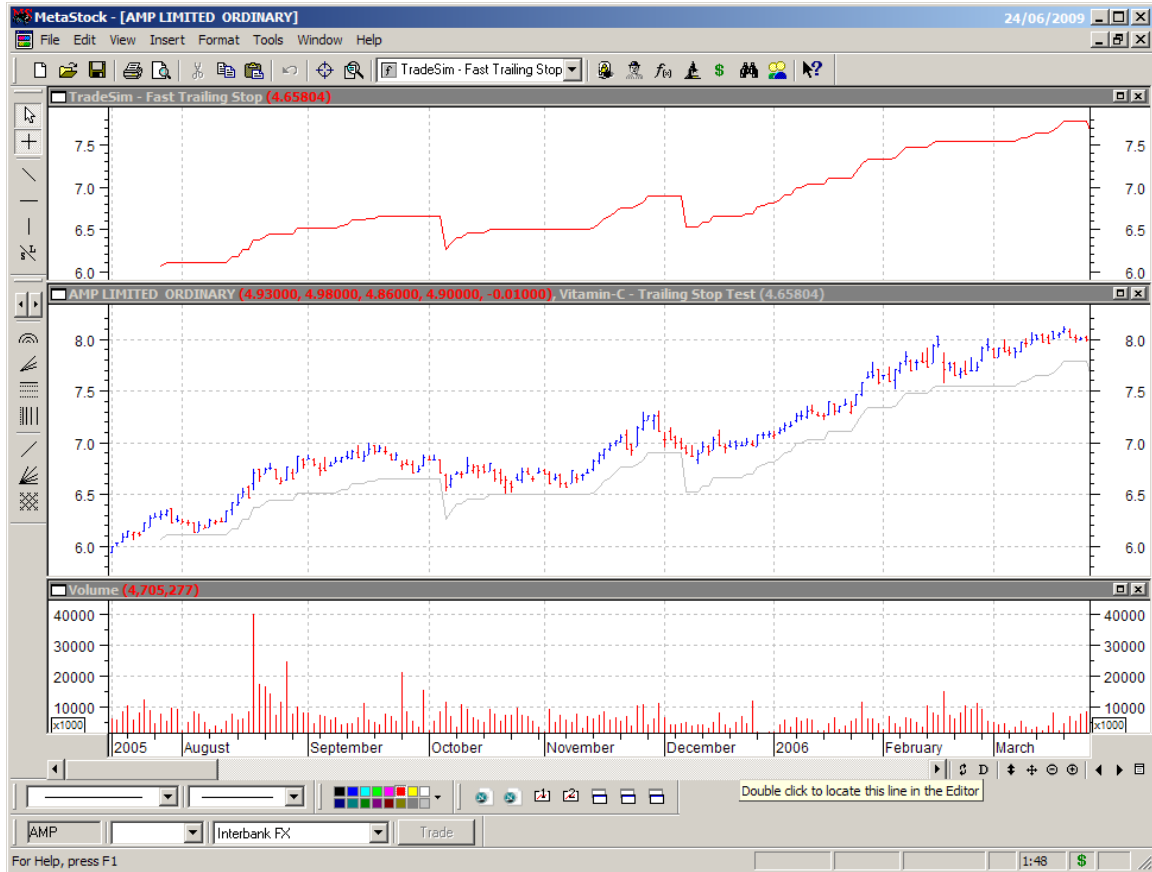
Running the Vitamin-C Trailing Stop code

The following MetaStock code was used to create an indicator, which displays a trailing stop band on the short side. This is essentially the same code used to call the Trailing Stop in the TradeSim(DLL) plug-in except for the first line where TradeSim is replaced with VitaminC.

```
ExtFml("VitaminC.CallScript3",
"TrailingStop.c",
"TrailingStop(BAND,LONG)",
3*ATR(10),
CLOSE,
LOW);
```

We overlayed this indicator directly on top of an existing chart to illustrate the trailing stop band function correctly. We also created the same indicator using the TrailingStop function built into the TradeSim plug-in library and added this on top of the bar chart. As you can see the two indicators are identical.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Exercise

As an Exercise change the MetaStock code to display both the long and short trailing stop bands. You need to add the following code to the indicator in addition to the existing code for the long band.

```
ExtFml("VitaminC.CallScript3",
"TrailingStop.c",
"TrailingStop(BAND,SHORT)",
3*ATR(10),
CLOSE,
HIGH);
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Implementing Moving Averages with Vitamin-C

If you have coded some systems or indicators with the MS formula language then you will most likely be familiar with the moving average formula, which is described in the online help as:

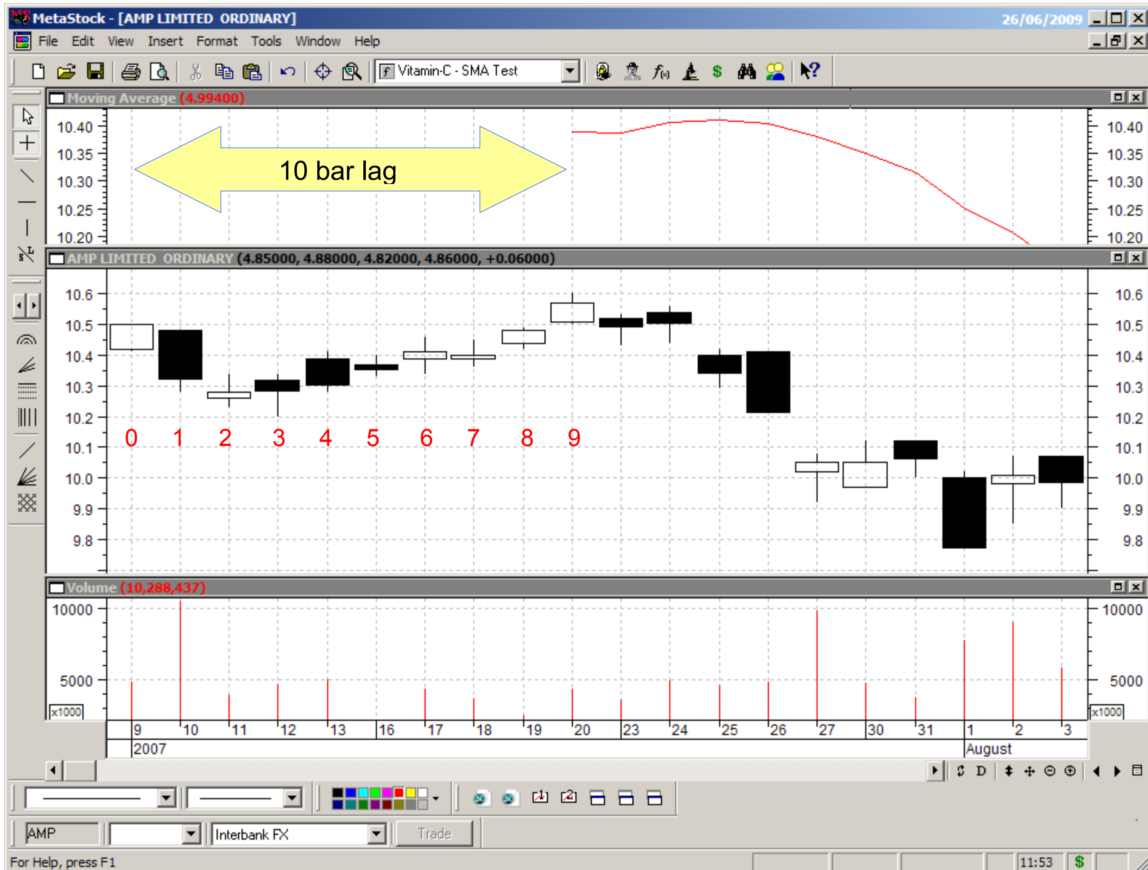
SYNTAX mov(DATA ARRAY, PERIODS, METHOD)

FUNCTION Calculates a PERIODS moving average of DATA ARRAY using METHOD calculation method.

Valid methods are EXPONENTIAL, SIMPLE, TIMESERIES, TRIANGULAR, WEIGHTED, VARIABLE, AND VOLUMEADJUSTED (these can be abbreviated as E, S, T, TRI, W, VAR, and VOL).

EXAMPLE The formula "mov(CLOSE, 25, EXPONENTIAL)" returns the value of a 25-period exponential moving average of the closing prices.

If we were to write our own moving average function with Vitamin-C script how would we do this ? Before we answer this we need to digress and talk a bit about lagging functions. These are functions, which do not produce a valid result until a number of bars of data have been processed. For example the simple moving average does not produce a valid result until it has processed the number of bars represented by its period. To prove this, open up a chart with MetaStock and add the Simple Moving Average Indicator with a period of 10 bars. As you can see the moving average doesn't plot values until the 10th bar.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved

What happens is that MetaStock knows that the SMA does not produce a valid response until the 10th bar so instead of displaying the bogus data, it actually suppresses it so that it is not displayed or used in any future computations. But how can we achieve the same thing with Vitamin-C ? The solution to this comes about because each Vitamin-C Array class object has a pair of member functions which control the first and last valid indexes for the array so it is just a matter of calling the Array::SetFirstIndex() function for lagging functions and Array::SetLastIndex() for leading functions. To illustrate the use of these functions lets start with the code for a simple moving average.

The Simple Moving Average (SMA)

First lets take a look at how the simple moving average (SMA) is computed. The simple moving average just calculates a running average of the last Nbars of data for each bar on the chart. It does this by summing the last Nbars of data and dividing by the number of bars. However it can only do this when there is more than Nbars worth of data to process. For example a 4 bar moving average will not be able to compute valid data until the fourth bar (bar=3) as in the following example:

Index →	0	1	2	3	4	5	6	7	8	9
Price	10.21	10.31	10.45	10.15	9.87	9.91	9.89	10.01	10.05	10.10
Σ(i-3…i)	-	-	-	41.12	40.78	40.38	39.82	39.68	39.86	40.05
SMA(4)	-	-	-	10.28	10.195	10.095	9.955	9.92	9.965	10.0125

To calculate the simple moving average we use two for-loops to do this. The outer loop (iterator-i) iterates through each bar in the chart whilst the inner loop (iterator-j) computes the average of Nbars for each value of iterator-i. The code for this is shown below:

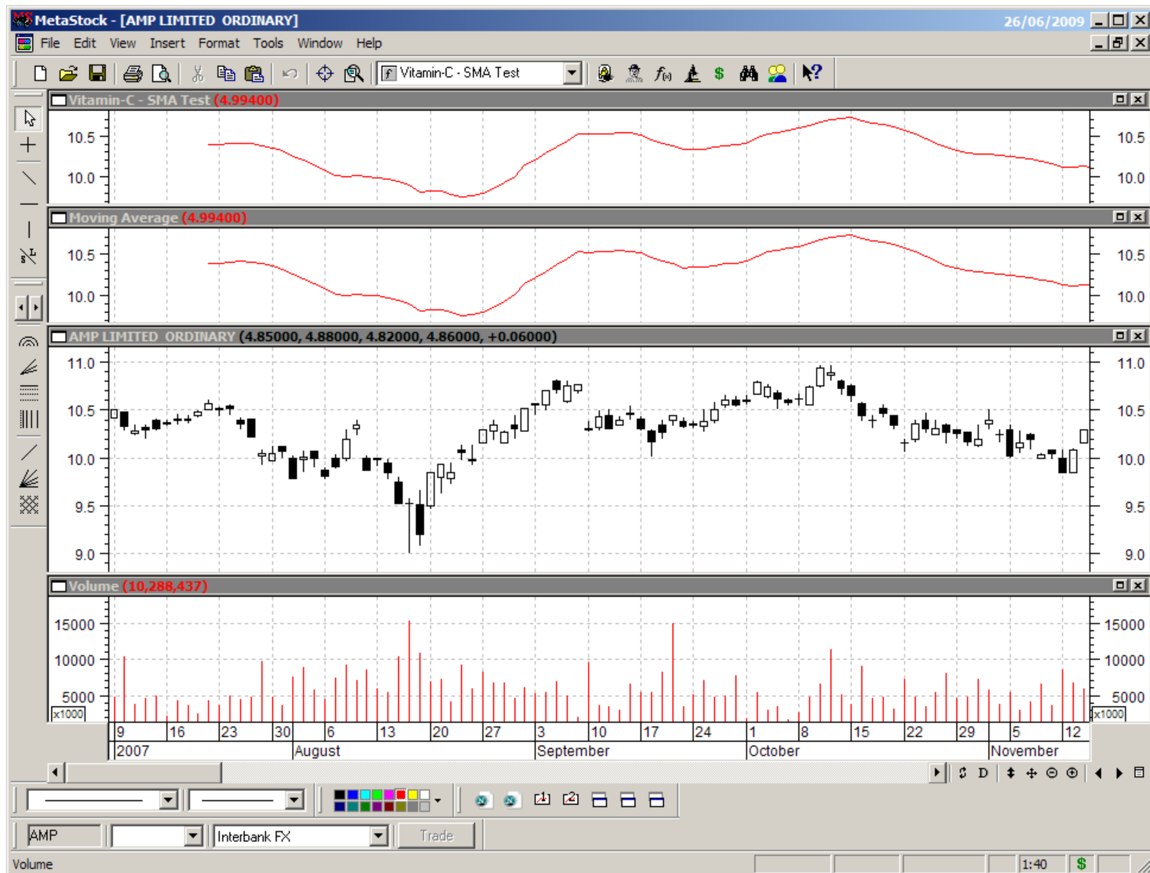
```
void SMA(int _Nbars)
{
  if(_Nbars<2)                        // check for invalid periods less than 2
    ReportError("Period (%d) is less than 2",_Nbars);

  double sum;                         // summation variable (double precision)

  for(int i=0;i<BarCount;i++)         // loop for all bars
  {
    sum=0;                            // initialize sum
    for(int j=0;j<_Nbars;j++)         // add up the data for last N bars
      sum+=User1[i-j];
    Result[i]=sum/_Nbars;             // compute the average
  }

  Result.SetFirstIndex(_Nbars-1);     // adjust for the lag
}
```

We save this code in a file called SMA.c and use the following indicator code to call it from MetaStock. In this example we wish to calculate a 10 bar SMA on the closing price.

```
ExtFml( "VitaminC.CallScript1",
"SMA.c",       { name of script file }
"SMA(10)",     { function name and SMA parameter }
CLOSE);        { data array }
```

In the following screen shot we have displayed both the Vitamin-C SMA and the MetaStock indicator configured with the same parameters. As you can see they are both identical with the same lag of 10 bars.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Improving the SMA

Is it possible to improve on the SMA algorithm used in the preceding example of the SMA code ? As it stands there appears to be a lot of redundant computation each time the average is calculated. If we observe the algorithm to compute the SMA more closely you should note that for each bar in the chart an Nbar SMA uses the same bar in the calculation of the SMA, Nbar times.

In fact for an Mbar=1000 bar chart, and a Nbar=100 bar SMA, this would require MxN or 1000x100 = 100,000 computations. For an indicator overlaid on a single chart this may not be a big deal but for an exploration containing 100's if not 1000's of securities as well as multiple calls to the SMA this will add up in terms of computational time.

Taking advantage of this redundancy means that we only have to subtract the first bar and add the new bar to an existing summation each time we need to calculate the new summation. This replaces the Nbar summation with just a 2 bar summation so the MxN computation now only requires Mx2 computations which is a massive savings in computational time for large SMA periods !!

The code to do this is shown below. As you can see it is much simpler requiring one main loop to iterate through all of the bars !!

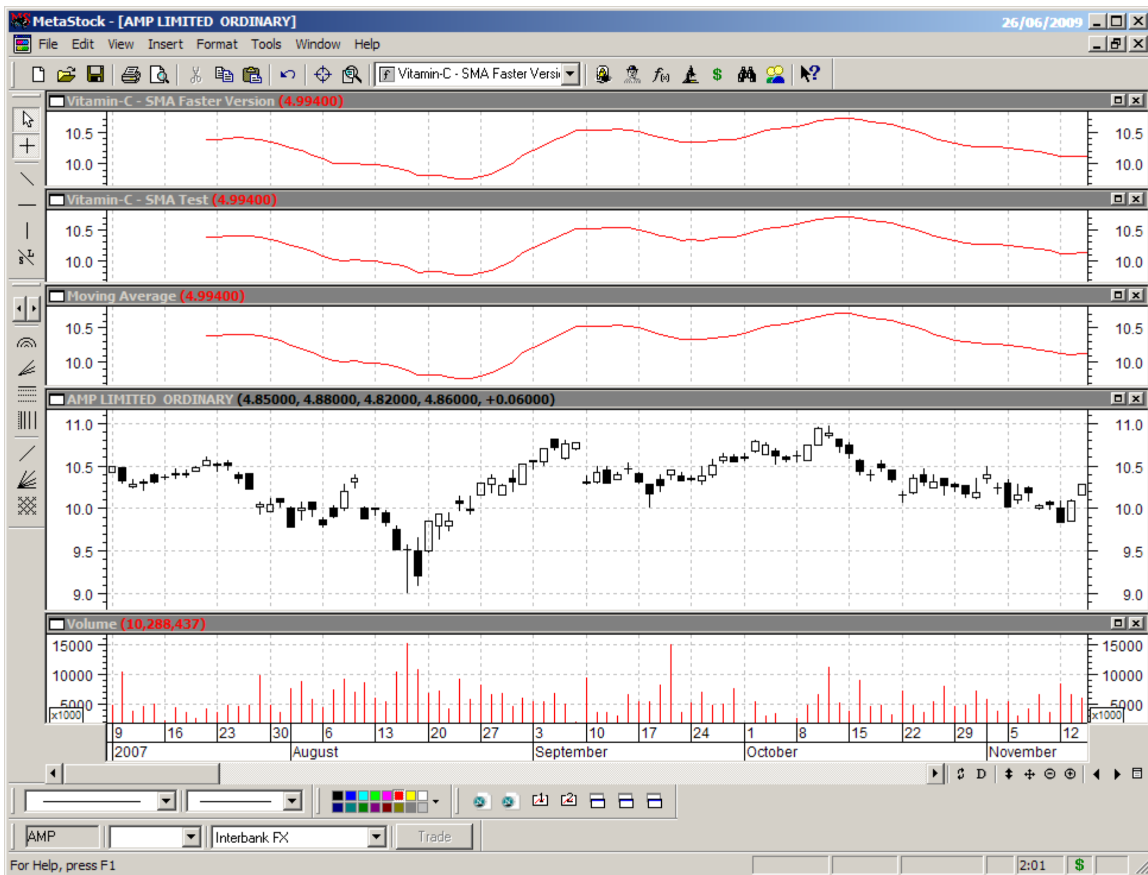
```
void SMA_Faster(int _Nbars)
{
  if(_Nbars<2)                        // check for invalid periods less than 2
    ReportError("Period (%d) is less than 2",_Nbars);

  double sum=0;                       // summation variable (double precision)

  for(int i=0;i<BarCount;i++)        // loop for all bars
  {
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
    sum+=User1[i]-User1[i-_Nbars];     // calculate the new summation
    Result[i]=sum/_Nbars;              // compute the new average
  }

  Result.SetFirstIndex(_Nbars-1);      // adjust for the lag
}
```

We use the following indicator code to call the new function with the same parameters as we used previously. Now we can overlay all three versions on the same bar chart. As you can see they are all identical to each other !!

```
ExtFml( "VitaminC.CallScript1",
"SMA.c",              { name of script file }
"SMA_Faster(10)",     { function name and SMA parameter }
CLOSE);               { data array }
```



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The Exponential Moving Average

Next we come to the exponential moving average (EMA), which is used a lot in technical analysis. From an engineering perspective the EMA is a form of an Infinite Impulse Response discrete time filter. Without getting to technical what this means from a layman's perspective is that each value of the EMA contains information from the past values all of the way back to the beginning of the time series, albeit as time goes back the contribution to the current value diminishes. The consequences of this from a computational point of view is that to calculate the EMA for the current bar requires a portion of the previous bars EMA value.

From Alexander Elder's book "Trading for a living" the EMA is computed as follows:

$$EMA_{tod} = P_{tod} * K + EMA_{yest}*(1-K)$$

where, $K = 2/(N + 1)$
N is the number of days in the EMA
P_{tod} = today's price
EMA_{yest} = the EMA of yesterday

Now lets code this using Vitamin-C:

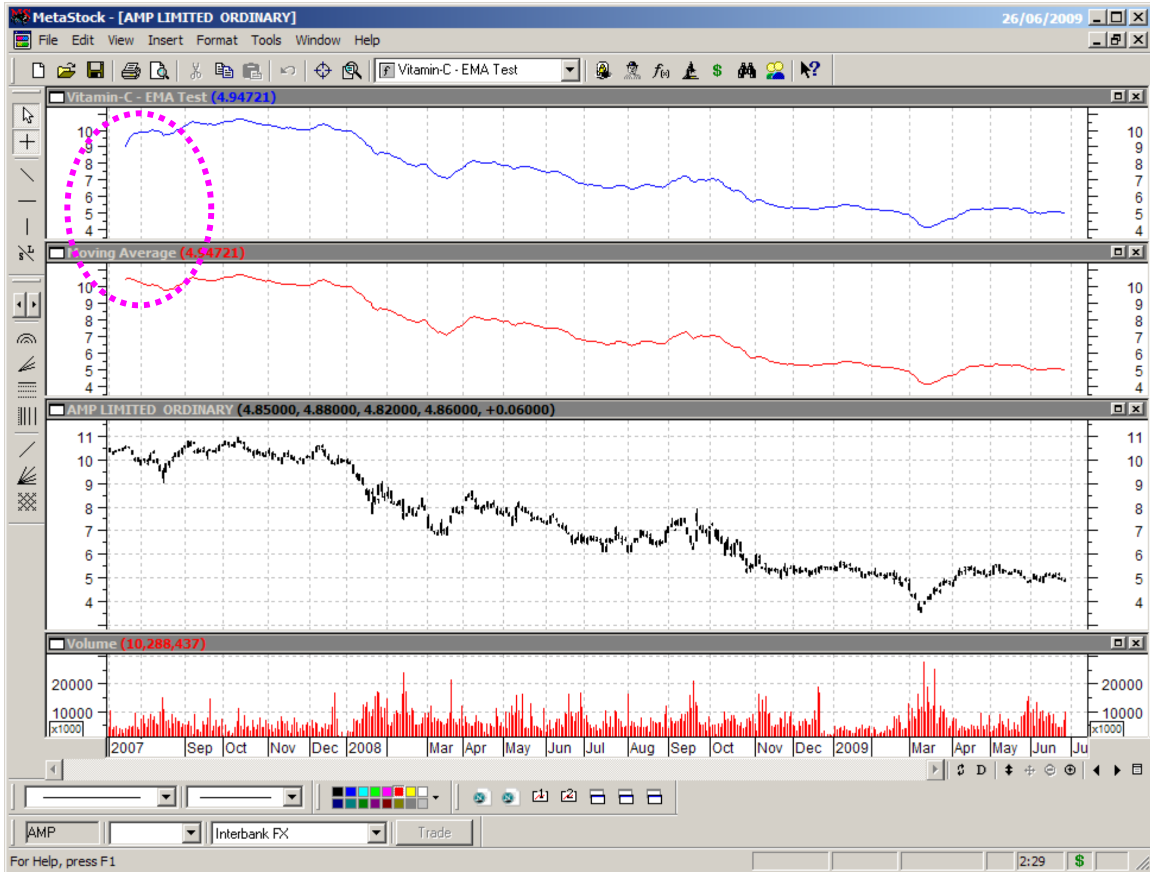
```
void EMA(int _Nbar)
{
  double K=2.0/(_Nbar+1);

  for(int i=0;i<BarCount;i++)       // loop for all bars
    Result[i]=User1[i]*K+Result[i-1]*(1-K);

  Result.SetFirstIndex(_Nbar-1);    // adjust for lag
}
```

We use the following indicator code to call the new function whilst overlaying the built in moving average indicator on the same chart to check if we are generating the correct values.

```
ExtFml( "VitaminC.CallScript1",
"EMA.c",            { name of script file }
"EMA(10)",          { function name and SMA parameter }
CLOSE);             { data array }
```

As you can see both look identical except at the start where the two diverge. This is highlighted by the dashed purple circle so lets zoom in and have a closer look at what's going on !

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



In the next screen shot I have merged the two indicators to see where the discrepancies are. It's only within the first 20 bars that there is significant error. This is due to the fact the Vitamin-C environment does not have access to data before the start of the chart so this causes some errors due to the windowing or truncating of the data. As shown the in the next screen shot after 30 bars the error approaches negligible levels and by the end of the chart the error is well within the numeric precision of 32 bit floating point numbers used to represent price data.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

We can improve this situation somewhat by using the first value of the data array that we want to smooth as an approximation to the EMA for the first bar (don't confuse this with the first bar of the EMA displayed on the chart). Lets change the code to do this:-

```
void EMA(int _Nbars)
{
  if(_Nbars<2)                         // check for invalid periods less than 2
    ReportError("Period (%d) is less than 2",_Nbars);

  double K=2.0/(_Nbars+1);
  float lastvalue=User1[0];            // kludge to compensate for no bars before first bar
  float newvalue;

  for(int i=0;i<BarCount;i++)          // loop for all bars
  {
    newvalue=User1[i]*K+lastvalue*(1-K);
    Result[i]=newvalue;
    lastvalue=newvalue;
  }

  Result.SetFirstIndex(_Nbars-1);      // adjust for lag
}
```

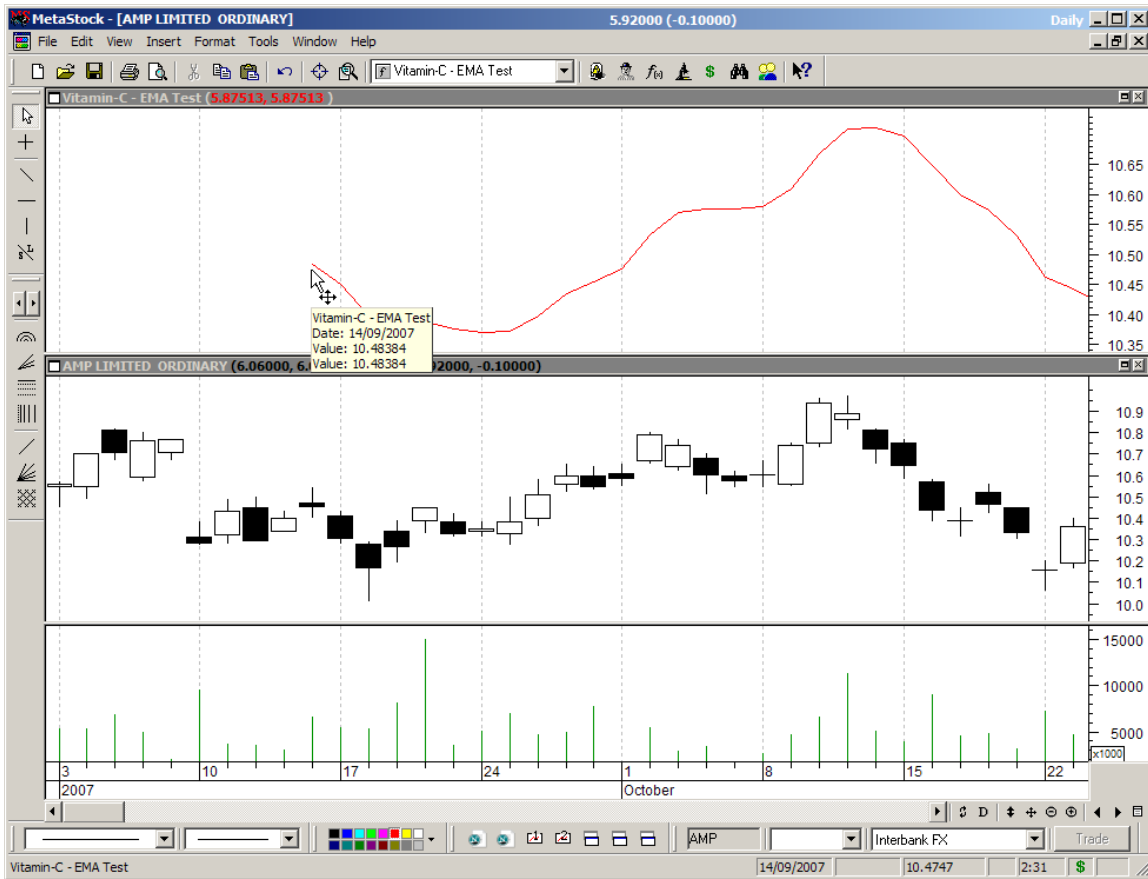
In the indicator code we have also added the MSFL equivalent of an EMA whicj will be overlayed on top of the Vitamin-C EMA.

```
Mov(C,10,E);  { MSFL EMA }


ExtFml( "VitaminC.CallScript1",
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
"EMA.c",        { name of script file }
"EMA(10)",      { function name and parameter }
CLOSE);         { data array }
```

As you can see the two EMA's overlayed on top of each other are indistinguishable from one another, the only hint that both exist on the same chart is the popup hint box displaying both values as being identical.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved

Porting Code from other Charting Platforms to Vitamin-C

It is possible to utilize code written for other charting platforms provided that the code closely resembles the C++ standard and doesn't use system dependent calls. For example, whilst the Amibroker Formula Language (AFL) is not industry standard C++ code, it shares some similarities, so much so that it is possible to port some examples across to the Vitamin-C environment. This is only possible if the code does not reference vendor specific functions such as charting functions. In the following sections we provide two examples on how to port code from Amibroker.

Example 1: Adaptive Moving Average (AMA)

Developed by Perry Kaufmann, this indicator is an EMA (exponential moving average) using an Efficiency Ratio to modify the smoothing constant, which ranges from a minimum of fast length to a maximum of slow length.

The Amibroker website describes an Adaptive Moving Average function which takes two user array arguments. The first argument is the data array that you want to smooth and the second argument is the smoothing factor array, which is normally a constant value in other moving averages.

output = AMA(input, factor)

is equivalent to the following looping code:

```
for( i = 1; i < BarCount; i++ )
{
  output[ i ] = factor[ i ] * input[ i ] + ( 1 - factor[ i ] ) * output[ i -1];
}
```

Rewriting this in Vitamin-C is fairly straightforward:

```
void AMA()
{
  for(int i=1;i<BarCount;i++)          // loop for all bars
  {
    Result[i] = User2[i] * User1[i] + ( 1 - User2[i] ) * Result[i-1];
  }

  Result.SetFirstIndex(1);             // ignore the first bar at the very minimum
}
```

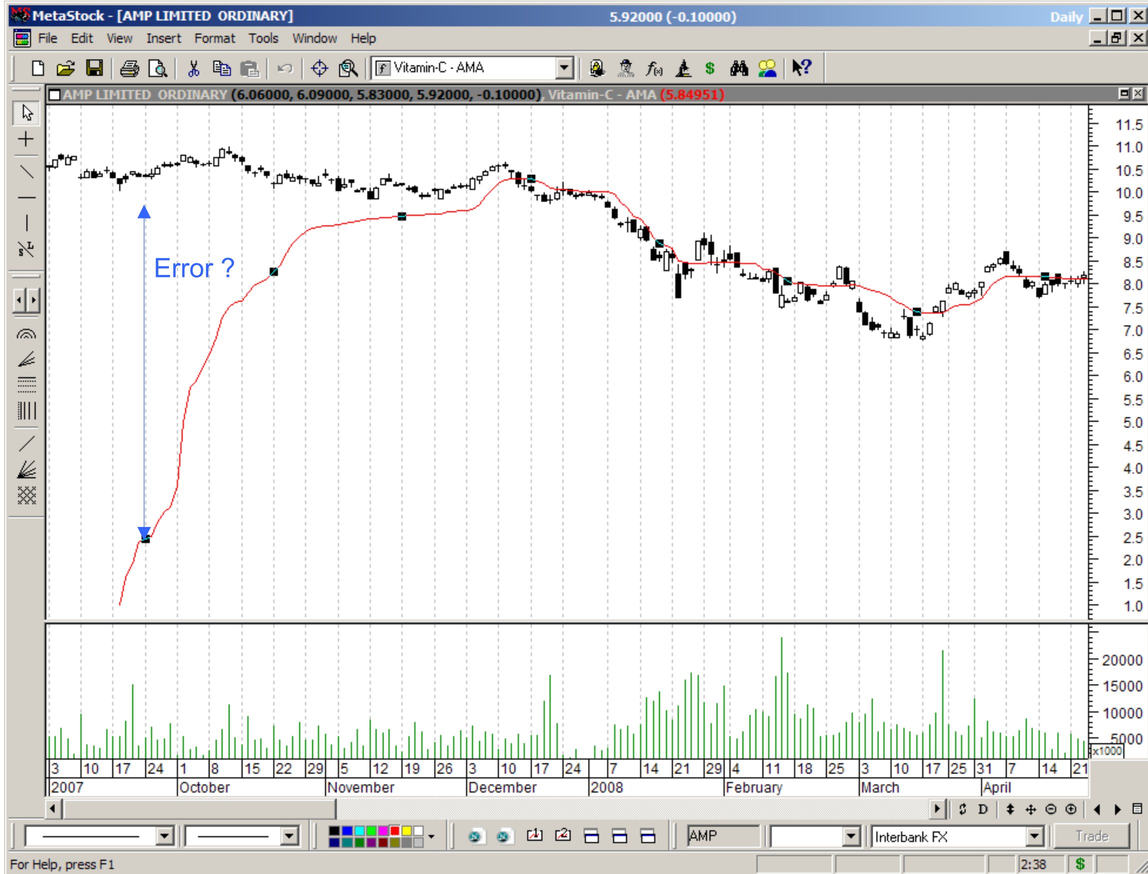
The AFL code example converted to MetaStock code is:

```
fast := 2/(2+1);
slow := 2/(30+1);
dir:=Abs(CLOSE-Ref(CLOSE,-10));
volatility:=Sum(Abs(CLOSE-Ref(CLOSE,-1)),10);
ER:=dir/volatility;
sc:=Power(ER*(fast-slow)+slow,2);


ExtFml( "VitaminC.CallScript2",
"AMA.c",      { name of script file }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
"AMA()",        { function name and optional parameters }
CLOSE,          { User1 data array }
sc);            { User2 data array }
```

The following screen shot shows the indicator above overlaid on a chart.

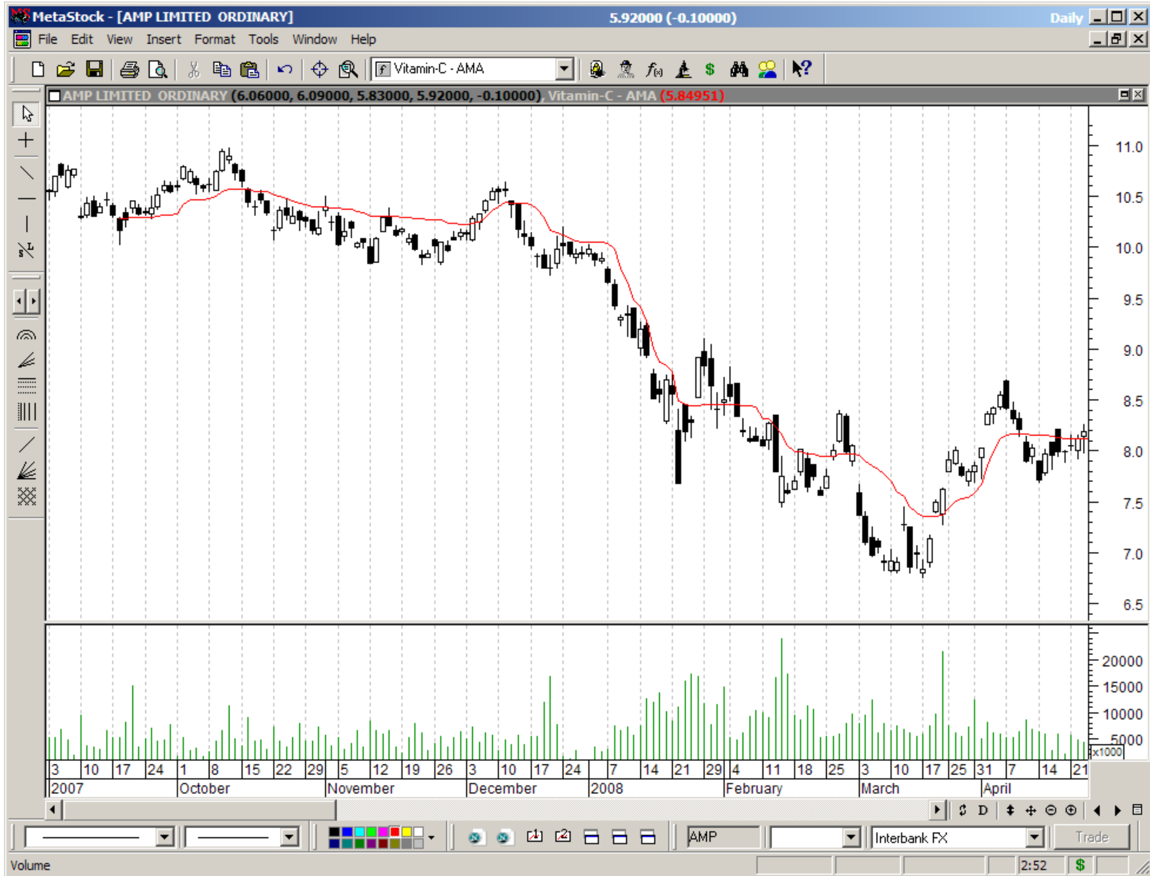


As you can see we still have a problem with the windowing or truncation of data causing errors at the start of the AMA. We can solve this in a similar way that we solved it for the EMA example in the previous section by using the first value of the data array we wish to smooth as the first value of the AMA. The modified code to do this is as follows:

```
void AMA()
{
  float lastvalue=User1[0];           // kludge to compensate for no bars before first
bar
  float newvalue;

  for(int i=1;i<BarCount;i++)         // loop for all bars
  {
    // AFL code output[i] = factor[i] * input[i] + ( 1 - factor[i] ) * output[i-1];
    newvalue = User2[i] * User1[i] + ( 1 - User2[i] ) * lastvalue;
    Result[i]=newvalue;
    lastvalue=newvalue;
  }

  Result.SetFirstIndex(1);            // ignore the first bar at the very minimum
}
```

As you can see in the next screen shot the problem has been solved.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Example 2: The Guppy Count back Line Trailing Stop Indicator

In the document "Looping in Amibroker AFL" is described a function which computes the Guppy Count back Line and a Trailing Stop. The AFL function to compute the Guppy count back line like is described in the document as:

```
function Cbl(bars)                    // Function definition
{
  cblArr = Null;                      // Initialise CBL array with nulls
  if (bars > 0)                       // Number of bars = 0 is invalid
  {
    for (i = 1; i < BarCount; i++) // Loop over all bars excluding first one
    {
      steps = bars - 1;               // Number of steps back is bars-1
      mostLow = Low[i];               // Start with low of current bar as lowest
      for (j = i-1; j >= 0 && steps > 0; j--) // Loop backwards over all ...
      {                               // ... previous bars or until steps = 0
        if (Low[j] < mostLow)         // If this bar is lower than current lowest
        {
          mostLow = Low[j];           // Set this low as new lowest
          steps--;                    // Decrement number of steps remaining
        }                             // End of if statement
      }                               // End of inner for loop using 'j'
      cblArr[i] = mostLow;            // CBL is the lowest low after steps back
    }                                 // End of outer for loop using 'i'
  }                                   // End of if (bars > 0)
  return cblArr;                      // Return CBL array to caller
}                                     // End of function
```

The AFL function to compute the Trailing Stop is:

```
function TrailingStop(data)           // Passed array has data for trailing stop
{
  stop[0] = data[0];                  // Set first bar's stop value
  for (i = 1; i < BarCount; i++)      // Loop through all other bars
  {
    if (Close[i] >= stop[i-1])        // If not stopped out yet
      stop[i] = Max(data[i], stop[i-1]); // Set stop level for this bar
    else                              // Else if is stopped out now
      stop[i] = data[i];              // Reset to current value of stop data
  }                                   // End of for loop
  return stop;                        // Return trailing stop array
}                                     // End of function
```

The code was modified so that it conforms to the C++ standard and the equivalent code re-written for Vitamin-C. As you can see it is very similar to the original except that it now conforms to the C++ standard.

```
Array CBL(int bars)                        // Function definition
{
  Array cblArr;                            // declare CBL array
  if (bars > 0)                            // Number of bars = 0 is invalid
  {
    for (int i = 1; i < BarCount; i++)     // Loop over all bars excluding first one
    {
      int steps = bars - 1;                // Number of steps back is bars-1
      float mostLow = Low[i];              // Start with low of current bar as lowest
      for (int j = i-1; j >= 0 && steps > 0; j--) // Loop backwards over all ...
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

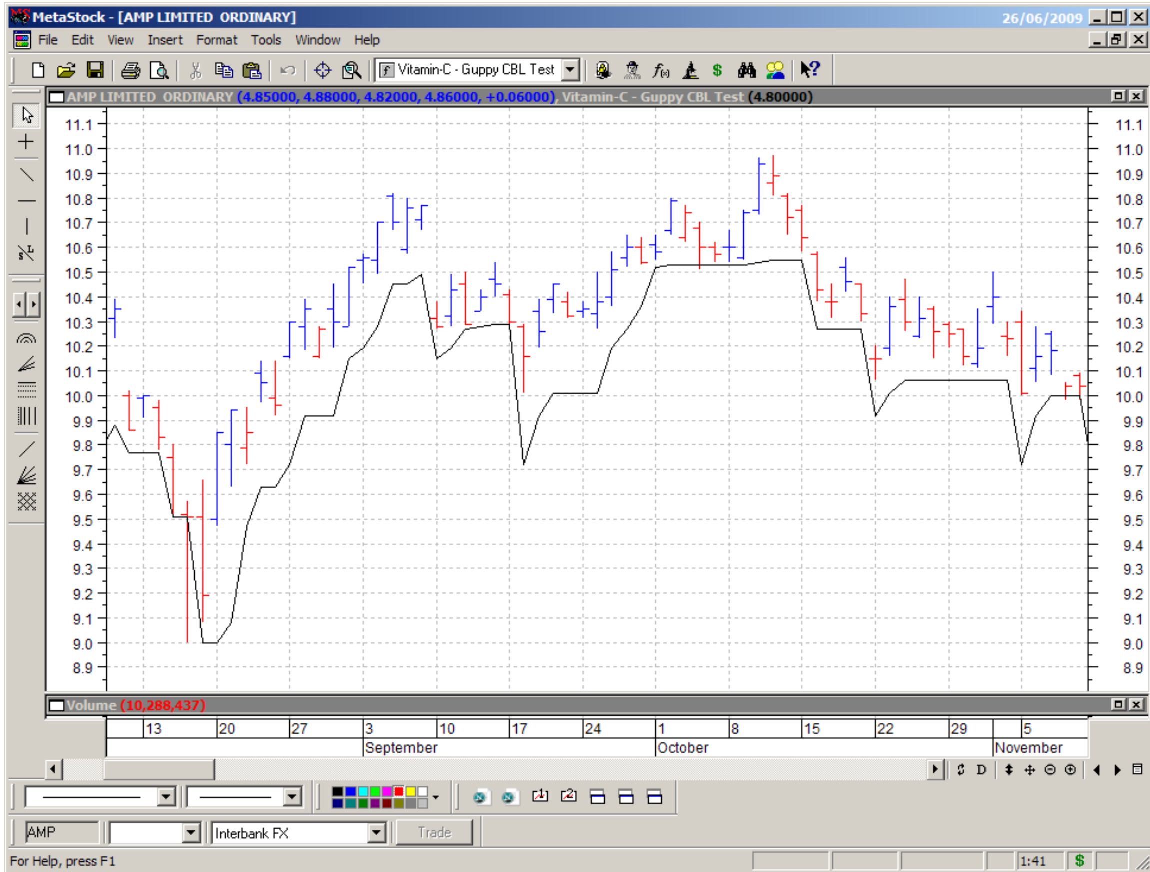
```
      {                                         // ... previous bars or until steps = 0
        if (Low[j] < mostLow)                   // If this bar is lower than current lowest
        {
          mostLow = Low[j];                     // Set this low as new lowest
          steps--;                              // Decrement number of steps remaining
        }                                       // End of if statement
      }                                         // End of inner for loop using 'j'
      cblArr[i] = mostLow;                      // CBL is the lowest low after steps back
    }                                           // End of outer for loop using 'i'
  }                                             // End of if (bars > 0)

  cblArr.SetFirstIndex(1);
  return cblArr;                                // Return CBL array to caller
}

Array TrailingStop(Array data)                  // Passed array has data for trailing stop
{
  Array stop;                                   // declare stop array
  stop[0] = data[0];                            // Set first bar's stop value
  for(int i = 1; i < BarCount; i++)             // Loop through all other bars
  {
    if (Close[i] >= stop[i-1])                  // If not stopped out yet
      stop[i] = data[i]>stop[i-1]?data[i]:stop[i-1];  // Set stop level for this bar
    else                                        // Else if is stopped out now
      stop[i] = data[i];                        // Reset to current value of stop data
  }                                             // End of for loop
  stop.SetFirstIndex(1);
  return stop;                                  // Return trailing stop array
}

void CBLTrailingStop(int bars)                  // main calling function from MetaStock
{
  Result=TrailingStop(CBL(bars));
}
```

To call the Guppy Count Back Line Trailing Stop function from MetaStock the following indicator code is used:

```
ExtFml( "VitaminC.CallScript",
"GuppyCBL.c",                       { name of script file }
"CBLTrailingStop(3)");              { function name and parameter }
```

Shown in the screen shot below is the Guppy Count Back Line Trailing Stop overlaid on a chart.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Mission accomplished!! ☺

Multi Dimensional basic arrays in Vitamin-C

You've already been introduced to the special Array Class in earlier sections, which are used to create, manipulate or access the existing price data arrays from MetaStock. This particular class wraps a layer around a standard single dimensional array of floating point numbers and is essentially the only type of array that you will need to deal with. However seasoned C++ programmers know that you can construct arrays in C++ out of any data types including fundamental, structure and class objects. To use arrays you first need to declare them.

Basic Array Declarations

In C and C++ it is possible to have any dimension array, however any array dimension above two may not have any practical use as far as technical analysis applications are concerned. In the following we list some of the more basic array declarations.

Single Dimensional Arrays

Single dimension arrays can be declared as follows where size is an integer type:

```
type var-name[size];
```

A 1-D array can be visualized as a line structure where the size represents the length of the line.

- **Examples**

```
int scores[10];                // array of 10 integers
float price[20];               // array of 20 floating point numbers
float price[]={1.26, 1.45, 1.33, 1.39};  // array of floats with
initialization
char name[80];                 // array of 80 characters
char symbol[]="ANZ";           // array of characters automatically
                             // sized to fit the character string
```

Two Dimensional Arrays

Two-dimensional arrays can be declared as follows where size is an integer type:

```
type var-name[rows][cols];
```

A 2-D array can be visualized as a flat square structure where the rows represent the side of the square and the columns represent the length of the square.

- **Examples**

```
int TwoDArray[20][10]        // 2D array of integers 20 rows by 10 columns
```

Three Dimensional Arrays

Three-dimensional arrays can be declared as follows where size is an integer type:

```
type var-name[layers][rows][cols];
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

A 3-D array can be visualized as a cube structure where the layers represent the depth of the cube, rows represent the height of the cube and the columns represent the width of the cube.

- **Examples**

```
float ThreeDArray[10][5][7];  // 3D array of floats of 10 layers by 5 rows
by 7 columns
```

N-Dimension Arrays

In C++ you can declare N-dimensional arrays of objects, but for most purposes regarding trading applications arrays above 3 dimensions are not really required. Having said that it is possible to visualize a 4th dimension array as a line of cubes where the size of the 4th dimension represents the length of the line. Similarly a 5th dimension array would be a 2-D array of cubes and a 6th dimension array would be represented by a cube array of cubes etc

- **Examples**

```
float FourDarray[20][10][5][7];       // 4-D array

float FiveDarray[30][20][10][5][7];   // 5-D array
```

Using basic arrays in Vitamin-C

Using arrays in Vitamin-C is no different than using them in any other C++ environment. Please refer to the C/C++ references at the back of this guide for more information on how to use them.

2-D basic array example

The following example demonstrates how to declare a 2-D array of integers, write values to it and then read and display the values to the Debug Log.

```
#define COLS 10
#define ROWS 10

// this function displays the contents of a 2D array
void Display2Darray(int _array2[][COLS])
{
  // read values from the array
  for(int row=0;row<ROWS;row++)
  {
    for(int col=0;col<COLS;col++)
      dprintf("Row=%d Column=%d value=%d\n",row,col,_array2[row][col]);
  }
}

// this function writes integer values into a 2D array and then calls another
// function to display the values
void Test2Darray()
{
  int array2[ROWS][COLS];

  // write values into the array
  int count=0;
  for(int row=0;row<ROWS;row++)
  {
    for(int col=0;col<COLS;col++)
    {
      array2[row][col]=count;
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
      count++;
    }
  }

  Display2Darray(array2);
}
```

The MetaStock Indicator code used to call this function is:

```
ExtFml( "VitaminC.CallScript",
"2DArray.c",            { name of script file }
"Test2Darray()");       { function name and parameter }
```

Here are the results from the debug log:

```
Debug Log
Clear Log   Append   Save Debug
 1 Row=0 Column=0 value=0
 2 Row=0 Column=1 value=1
 3 Row=0 Column=2 value=2
 4 Row=0 Column=3 value=3
 5 Row=0 Column=4 value=4
 6 Row=0 Column=5 value=5
 7 Row=0 Column=6 value=6
 8 Row=0 Column=7 value=7
 9 Row=0 Column=8 value=8
10 Row=0 Column=9 value=9
11 Row=1 Column=0 value=10
12 Row=1 Column=1 value=11
13 Row=1 Column=2 value=12
14 Row=1 Column=3 value=13
15 Row=1 Column=4 value=14
```

3-D basic array example

The following example demonstrates how to declare a 3-D array of floating point numbers, write values to it and then read and display the values to the debug log.

```
#define COLS 10
#define ROWS 10
#define LAYERS 10

// this function displays the contents of a 3D array of floating point numbers
void Display3Darray(float _array3D[][ROWS][COLS])
{
  // read values from the array
  for(int layer=0;layer<LAYERS;layer++)
  {
    for(int row=0;row<ROWS;row++)
    {
      for(int col=0;col<COLS;col++)
        dprintf("Layer=%d Row=%d Column=%d
value=%f\n",layer,row,col,(double)_array3D[layer][row][col]);
    }
  }
}
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

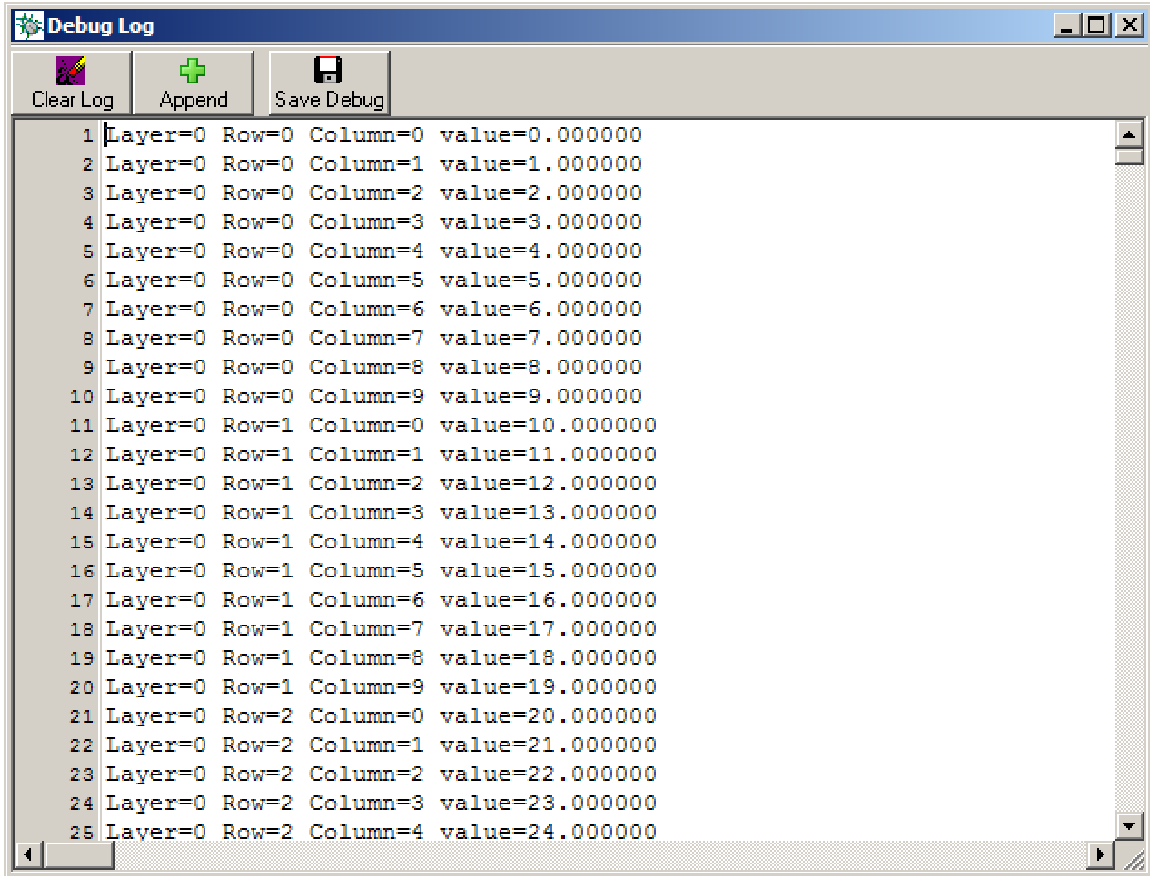
```
// this function writes float values into a 2D array and then calls another
// function to display the values
void Test3Darray()
{
  float array3D[LAYERS][ROWS][COLS];

  // write values into the array
  int count=0;
  for(int layer=0;layer<LAYERS;layer++)
  {
    for(int row=0;row<ROWS;row++)
    {
      for(int col=0;col<COLS;col++)
      {
        array3D[layer][row][col]=count;
        count++;
      }
    }
  }

  Display3Darray(array3D);
}
```

The MetaStock Indicator code used to call this function is:

```
ExtFml( "VitaminC.CallScript",
"3DArray.c",          { name of script file }
"Test3Darray()");     { function name and parameter }
```

Here are the results from the Debug Log:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Using the Standard 'C' library from Vitamin-C

Most of the standard C-library functions are available from within the Vitamin-C environment. We shall not discuss all of the functions as this is beyond the scope of this document and is better left to the texts on 'C/C++' in the reference section cited at the end of this User Guide. However we will illustrate a couple of simple examples.

Transcendental Example

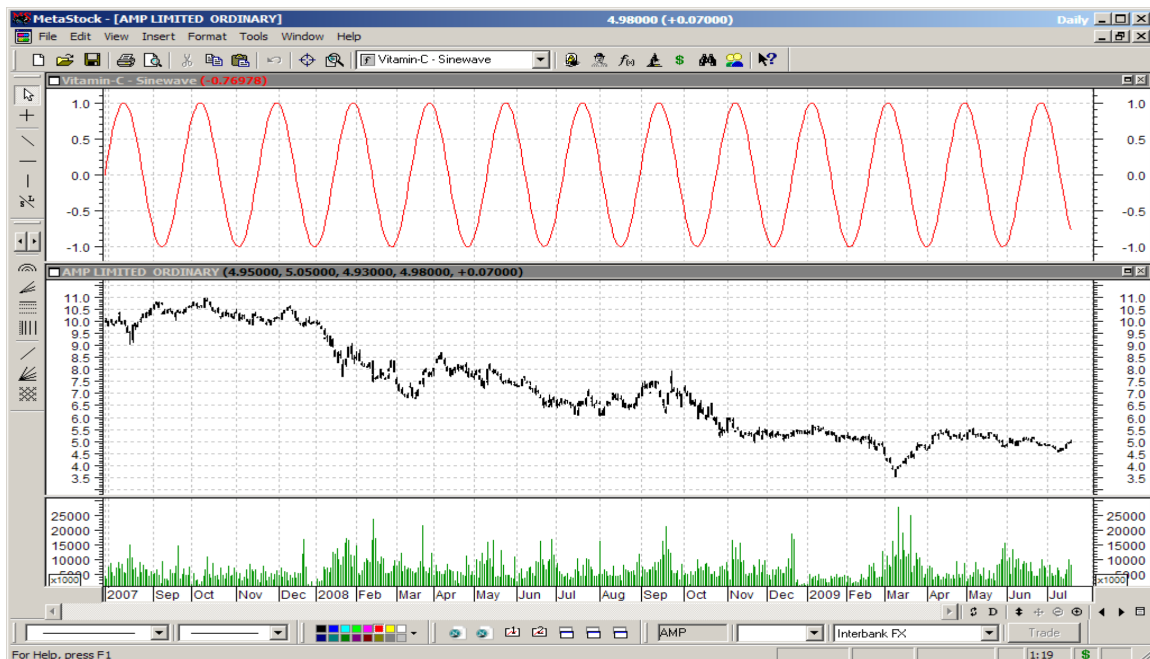
We shall use the transcendental trigonometric sine() function available from the standard 'C' library to plot a sinewave on a chart. The Vitamin-C code to this is as follows:

```
#define PI 3.1415926

void Sinewave(void)
{
  for(int i=0;i<BarCount;i++)
  {
    Result[i]=sin(i/(2*PI));
  }
}
```

The MetaStock indicator code used to call the Vitamin-C script is as follows:

```
ExtFml( "VitaminC.CallScript",
"Sinewave.c",        { name of script file }
"Sinewave()");       { function name and parameter }
```

Placing the indicator on a chart yields the following results:



File IO example

The standard 'C' library provides a rich set of functions that enable file creation and manipulation. In this example we will take the Profit Stop that we coded earlier and add some file handling routines that allow us to write out the trade data to a file called "trades.csv" which will be saved in the "c:\VitaminCScript directory". The modified code is shown below with the additions marked in grey background.

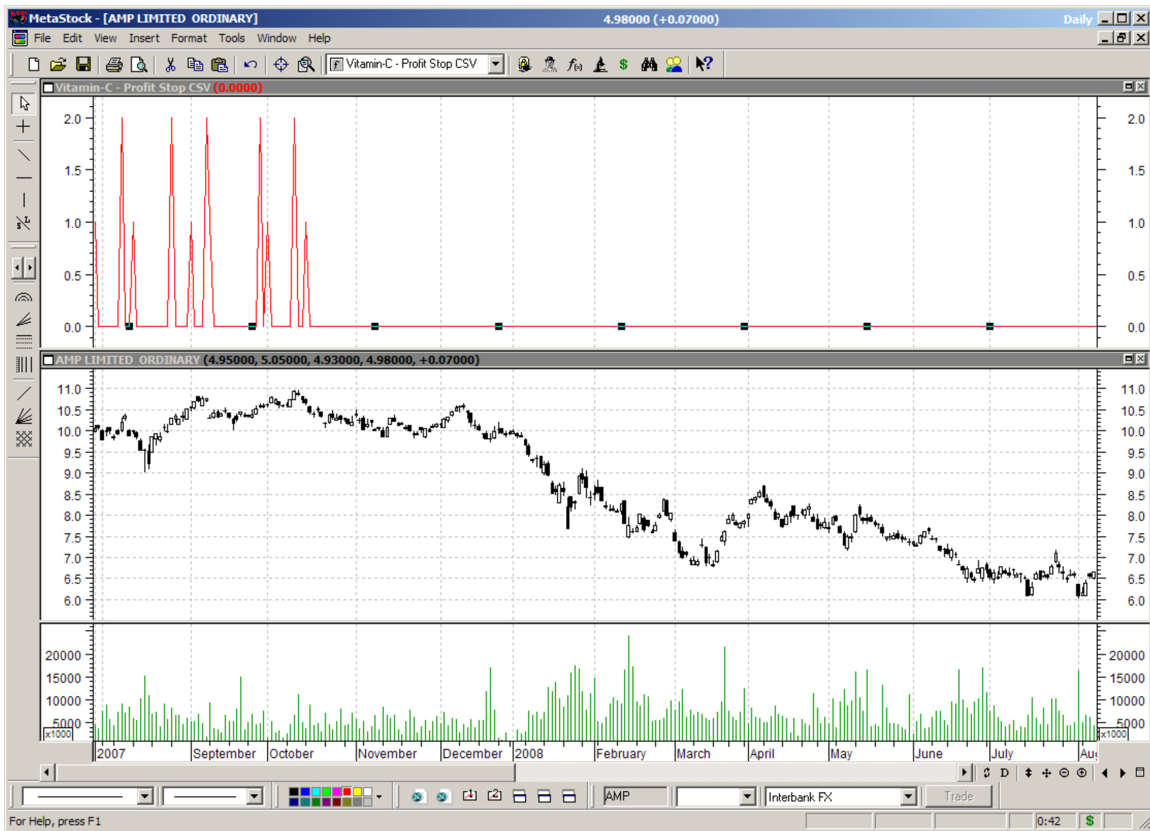
```
void ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;                 // boolean variable used for trade flag
  float EntryPrice=0;                 // variable used to store the entry price on trade
entry
  float ProfitThreshold;              // variable used to hold the profit threshold price
  long EntryDateIndex;                // entry date index

                                      // open file ready for writing text data. You should
                                      // specify a complete file path otherwise the file
                                      // will be created in the MetaStock directory
  FILE *fileptr=fopen("C:\\VitaminCScript\\trades.csv","wt");

  if(fileptr!=NULL)                   // check if successful open
  {
    // the following writes out the column headings to the file
    fprintf(fileptr,"%4s,%12s,%12s,%12s,%12s\n",
      "Sym",
      "Entry Date",
      "Entry Price",
      "Exit Date",
      "Exit Price");

    for(int i=0;i<BarCount;i++)       // loop for all bars
    {
      Result[i]=0;                    // initialize result for each bar

      if(!InTrade && User1[i]>0)      // check if not in a trade and valid entry trigger
      {
        InTrade=true;                 // set the flag
        EntryPrice=Open[i];           // set the entry price
                                      // calculate the profit threshold price
        ProfitThreshold=User2[i]*(1 + _PercentThreshold/100);
        Result[i]=1;                  // mark a valid entry condition
        EntryDateIndex=i;
      }

      if(InTrade)                     // if in the trade do the check
      {
        if(User3[i] >= ProfitThreshold)     // check for a profit stop condition
        {
          Result[i]+=2;               // threshold reached so mark a valid exit condition
          InTrade=false;              // reset the flag
                                      // the following writes out the data to the file
                                      // with proper column formatting
          fprintf(fileptr,"%4s,%12s,%12f,%12s,%12f\n",
            GetSymbol(),                    // symbol
            GetDateString(EntryDateIndex),  // Entry date
            (double)EntryPrice,             // Entry Price
            GetDateString(i),               // Exit date
            (double)User3[i]);              // Exit price
        }
      }
    }
    fclose(fileptr);                  // close the file
  }
  else
  {                                   // else report an error
    ReportError("Trade file could not be opened");
  }
}
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The MS indicator code is as follows:

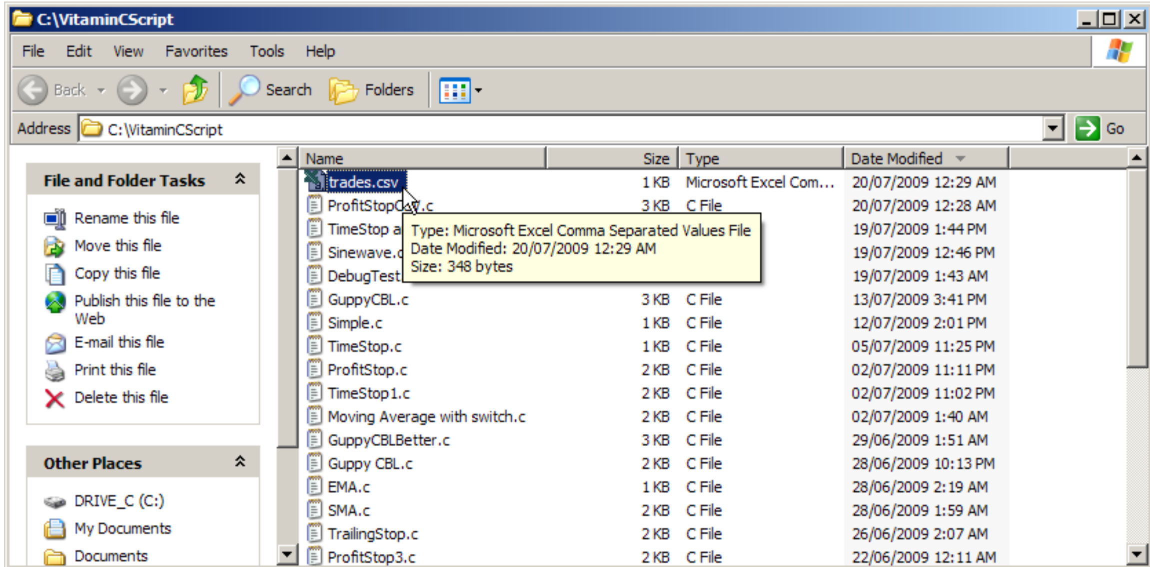
```
EntryTrigger:=(DayOfWeek()=1);
EntryPrice:=OPEN;   { entry or reference price }
ExitPrice:=CLOSE;   { exit or threshold price }

EncodedTrigger:=ExtFml( "VitaminC.CallScript3",
"ProfitStopCSV.c",
"ProfitStop(2)",
EntryTrigger,
EntryPrice,
ExitPrice);

EncodedTrigger;
```

When we apply the indicator to the chart there is only a small amount of trading activity.

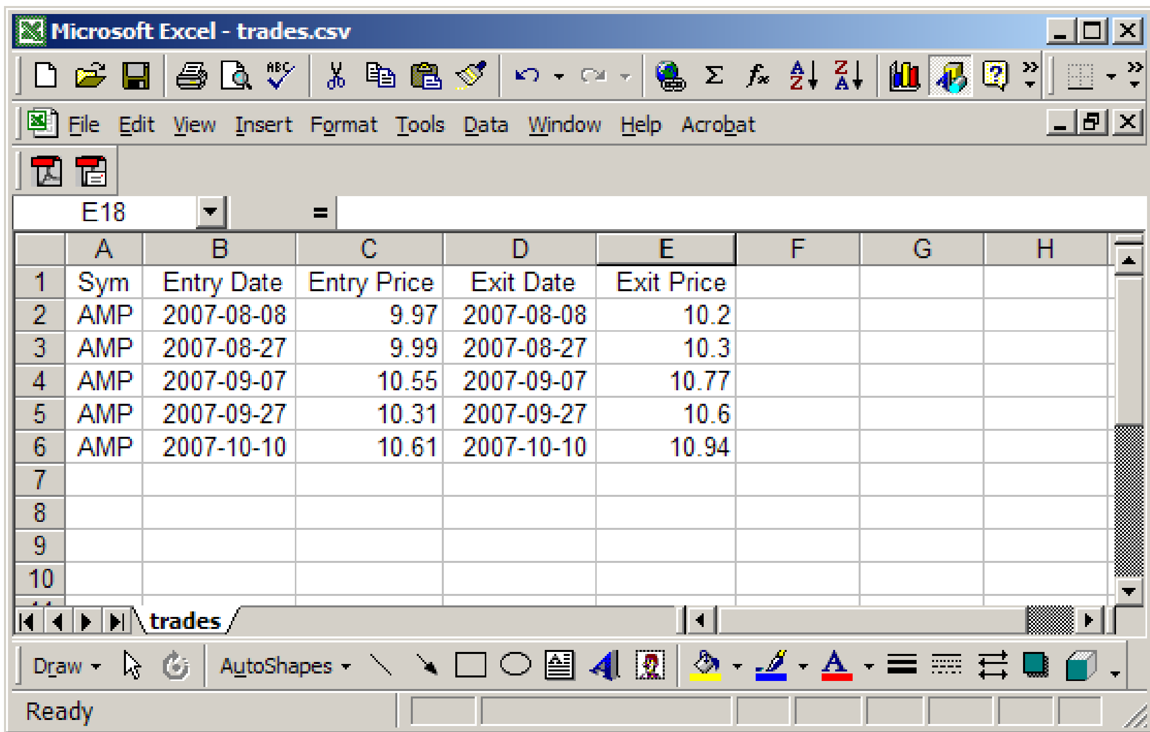


Using the Microsoft File Explorer we see that “trades.csv” has been created in the “c:\VitaminCScript” directory as we expected it to be !

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



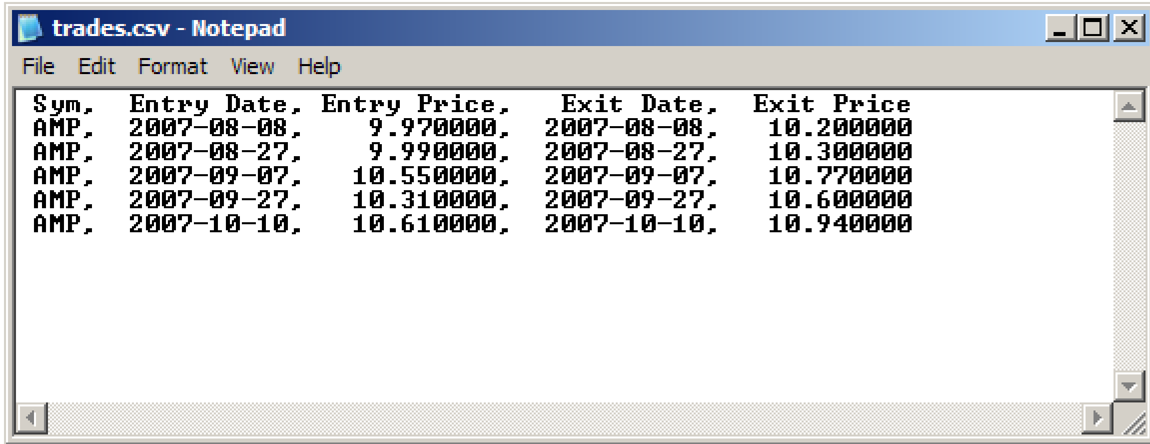
Because files with the csv extension have been registered to open using Excel we can just double click on this file and it will be loaded into Excel.



	A	B	C	D	E
1	Sym	Entry Date	Entry Price	Exit Date	Exit Price
2	AMP	2007-08-08	9.97	2007-08-08	10.2
3	AMP	2007-08-27	9.99	2007-08-27	10.3
4	AMP	2007-09-07	10.55	2007-09-07	10.77
5	AMP	2007-09-27	10.31	2007-09-27	10.6
6	AMP	2007-10-10	10.61	2007-10-10	10.94

Alternatively since the file is text based we could view it in NotePad or any other text based viewer or editor as shown below:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



trades.csv - Notepad

Sym	Entry Date	Entry Price	Exit Date	Exit Price
AMP	2007-08-08	9.970000	2007-08-08	10.200000
AMP	2007-08-27	9.990000	2007-08-27	10.300000
AMP	2007-09-07	10.550000	2007-09-07	10.770000
AMP	2007-09-27	10.310000	2007-09-27	10.600000
AMP	2007-10-10	10.610000	2007-10-10	10.940000

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Using Vitamin-C with TradeSim

You can use Vitamin-C with TradeSim in one of two ways. You can use the Vitamin-C scripts to generate Entry/Exit triggers and prices just as you would do for other MetaStock formulas or plug-in's. The second way to use Vitamin-C with TradeSim is to actually create the trade database from within your Vitamin-C code. Because Vitamin-C comes with the standard 'C' library, the entire file IO functions are available from within the Vitamin-C environment so it is quite easy to create a text based trade database that can be loaded directly into TradeSim. In the next sections we cover examples of both methods of using Vitamin-C with TradeSim.

Method 1: Generating Entry and Exit Triggers using Vitamin-C

In this case we use the TradeSim.RecordTrades() function from the TradeSim.dll plugin to create a binary trade database file, but we use some Vitamin-C code to create our entry and exit triggers for a profit stop based system. We shall take the profit stop from an earlier chapter and use this as the basis of a simple trading system, albeit a theoretical one, as nobody would realistically trade a system without a protective stop.

```
void ProfitStop(float _PercentThreshold)
{
  bool InTrade=false;             // boolean variable used for trade flag
  float EntryPrice=0;             // variable used to store the entry price on trade entry
  float ProfitThreshold;          // variable used to hold the profit threshold price

  for(int i=0;i<BarCount;i++)     // loop for all bars
  {
    Result[i]=0;                  // initialize result for each bar

    if(!InTrade && User1[i]>0)    // check if not in a trade and valid entry trigger
    {
      InTrade=true;               // set the flag
      EntryPrice=Open[i];         // set the entry price
                                  // calculate the profit threshold price

      ProfitThreshold=User2[i]*(1 + _PercentThreshold/100);
      Result[i]=1;                // mark a valid entry condition
    }

    if(InTrade)                   // if in the trade do the check
    {
      if(User3[i] >= ProfitThreshold)     // check for a profit stop condition
      {
        Result[i]+=2;             // threshold reached so mark a valid exit condition
         InTrade=false;          // reset the flag
      }
    }
  }
}
```

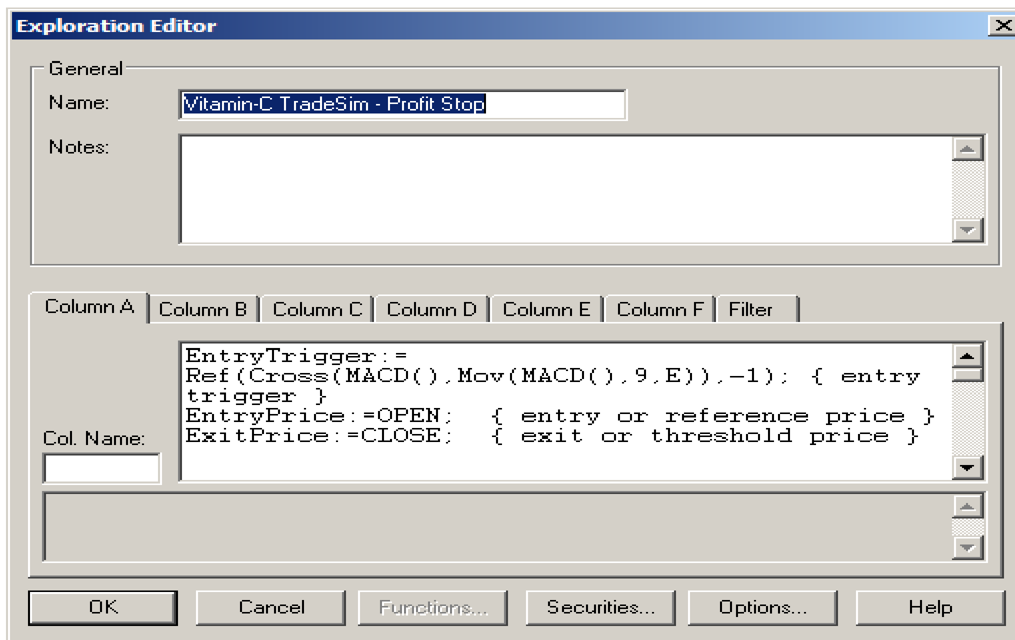
To create a trade database exploration the following code is used in an exploration. We use the MACD Entry Trigger to time the entries into the trade. Note that, not all entry triggers will be taken, as this will depend on the exit conditions because of the way the RecordTrades function works to filter out subsequent entry triggers until a subsequent exit condition is found.

```
EntryTrigger:= Ref(Cross(MACD(),Mov(MACD(),9,E)),-1); { entry trigger }

EntryPrice:=Open;  { entry or reference price }

ExitPrice:=Close;  { exit or threshold price }


EncodedTrigger:=ExtFml( "VitaminC.CallScript3",
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

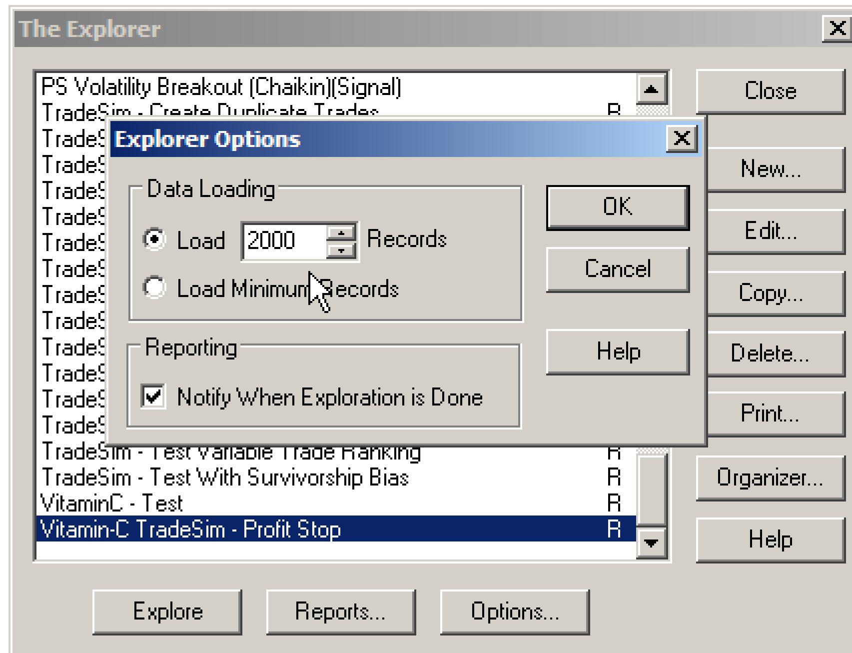
```
"ProfitStop.c",      { name of script file }
"ProfitStop(5)",     { function name and profit threshold (5%) argument }
EntryTrigger,        { User defined entry trigger }
EntryPrice,          { User defined entry or reference price }
ExitPrice            { User defined exit or threshold price }
);

ActualEntryTrigger:=EncodedTrigger=1 OR EncodedTrigger=3;
ActualExitTrigger:=EncodedTrigger=2 OR EncodedTrigger=3;
InitialStop:=0;

ExtFml( "TradeSim.Initialize");

ExtFml( "TradeSim.RecordTrades",
        "Vitamin-C Profit Stop",{ Trade Database Filename }
         LONG,                  { Trade Position Type }
         ActualEntryTrigger,    { Entry Trigger }
         EntryPrice,            { Entry Price }
         InitialStop,           { Optional Initial Stop }
         ActualExitTrigger,     { Exit Trigger }
         ExitPrice,             { Exit Price }
         START);                { Recorder Control }
```

Insert the code into the MS Explorer according to the TradeSim procedure.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

You may have to select a reasonable number of bars to take advantage of the periods of a rising market.



Run the exploration and load the corresponding Trade Database into TradeSim.

Trade Database Manager

Trade	Sys ID	Pos	Symbol	Periodicity	CE	SBFTE	Entry Date-Time	Exit Date-Time	P-Group	P-Level	Re-entry Type
24	0	Long	BHP	Daily	Yes	Yes	31/12/2001	07/01/2002	2	0	Base
25	0	Long	RIO	Daily	Yes	Yes	31/12/2001	01/02/2002	2	0	Base
26	0	Long	TLS	Daily	Yes	Yes	31/12/2001	23/07/2009	2	0	Base
27	0	Long	BHP	Daily	Yes	Yes	25/01/2002	14/02/2002	3	0	Base
28	0	Long	QBE	Daily	Yes	Yes	05/02/2002	14/02/2002	2	0	Base
29	0	Long	NCM	Daily	Yes	Yes	06/02/2002	20/03/2002	3	0	Base
30	0	Long	BXB	Daily	Yes	Yes	05/03/2002	20/03/2006	3	0	Base
31	0	Long	WOW	Daily	Yes	Yes	05/03/2002	19/04/2002	2	0	Base
32	0	Long	BHP	Daily	Yes	Yes	11/03/2002	16/10/2003	4	0	Base
33	0	Long	CBA	Daily	Yes	Yes	21/03/2002	03/05/2002	2	0	Base
34	0	Long	SUN	Daily	Yes	Yes	22/03/2002	27/02/2004	2	0	Base
35	0	Long	WES	Daily	Yes	Yes	02/04/2002	07/09/2004	2	0	Base
36	0	Long	QBE	Daily	Yes	Yes	08/04/2002	01/11/2002	3	0	Base
37	0	Long	RIO	Daily	Yes	Yes	11/04/2002	17/01/2005	3	0	Base
38	0	Long	WBC	Daily	Yes	Yes	16/04/2002	26/04/2002	2	0	Base
39	0	Long	NAB	Daily	Yes	Yes	18/04/2002	20/05/2002	3	0	Base
40	0	Long	AMP	Daily	Yes	Yes	22/04/2002	23/07/2009	3	0	Base
41	0	Long	NCM	Daily	Yes	Yes	30/04/2002	07/05/2002	4	0	Base
42	0	Long	WOW	Daily	Yes	Yes	06/05/2002	11/11/2004	3	0	Base
43	0	Long	WPL	Daily	Yes	Yes	09/05/2002	17/12/2003	2	0	Base
44	0	Long	ORG	Daily	Yes	Yes	24/05/2002	27/08/2002	2	0	Base

Start Entry Date 17/10/2001 Stop Entry Date 20/03/2009 272 trades selected from a total of 272 trades

As you can see, all trades except for the open trades meet their profit targets which means that the Vitamin-C code is doing its job in generating the trade data.

The problem with a profit stop based system is that it always looks good on a closed trade equity chart because the trades always exit with a minimum profit gain. However their maybe sustained periods of severe draw down, which is not shown up on the closed trade chart. As an example we ran a standard simulation using the following parameters:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Trade Parameters

Trade Parameters | Preferences

Position Size Model

- (•) Equal Dollar Units
- ( ) Equal Percent Dollar Units
- ( ) Fixed Dollar Risk
- ( ) Fixed Percent Risk
- ( ) Fixed Dollar Volatility
- ( ) Fixed Percent Volatility
- [ ] Pyramid Profits
- [ ] Pyramid Trades

Simulation Type

- (•) Portfolio Simulation
- ( ) Portfolio Simulation (Ignore Dates)
- ( ) Basket test
- ( ) Monte Carlo Analysis

Simulation Options

- (•) Use Original Ordering
- ( ) Random Walk

Trade Parameters (Stocks)

Initial Trading Capital: $50000.00
Transaction Cost (each way): $0.000
[ ] Use Transaction Cost from Trade Database
Portfolio Limit: 100.00%
Margin Requirement: 100.00%
[ ] Use Margin Req from Trade Database
Total Maximum Open Positions: 100
[ ] Magnify Position Size(and Risk) according to Margin Requirement
Maximum allowable daily orders: 20

Parameters - Equal Dollar Units

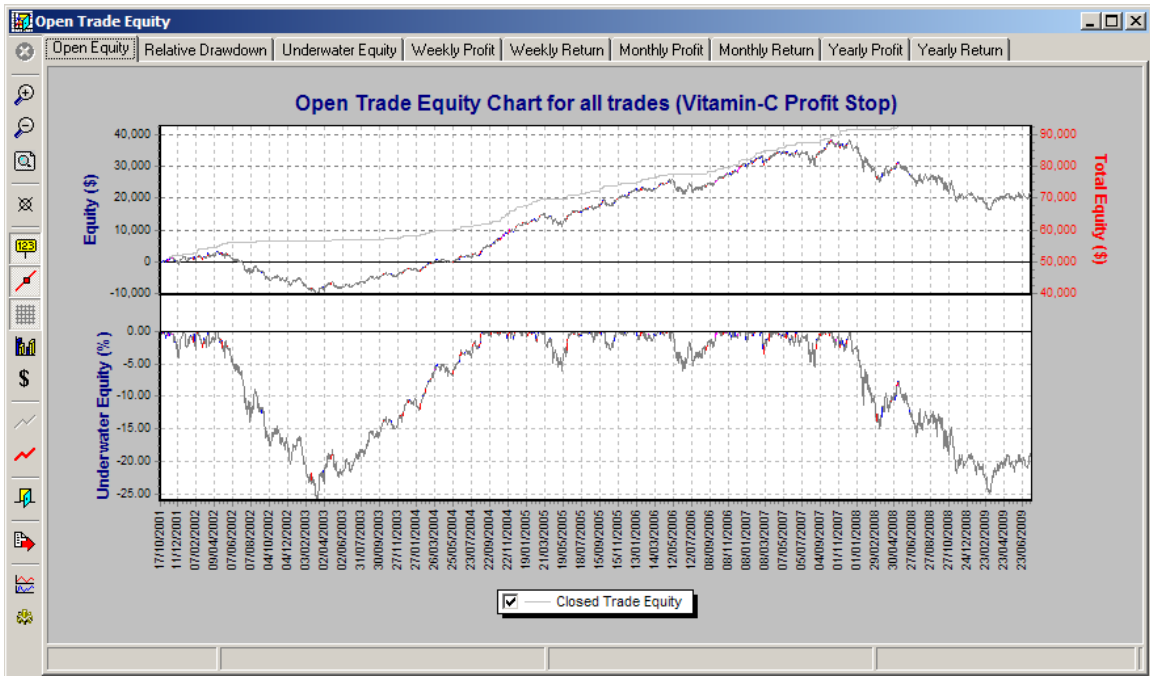
Capital per Trade: $5000.00

Margin Requirement Interest Rate

Long Trades (Debit): 0.0000%
Short Trades (Credit): 0.0000%
[ ] Specify Daily Interest Rate

- [ ] Select Position Size Model from Trade Database
- [ ] Enable Provisional Trades
- [ ] Enable Survivorship Bias Filter

We then plotted the open equity chart along with the closed trade equity chart:



As you can see the under water equity displays severe retracements in equity for long periods of time.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Method 2: Creating a Text Trade Database.

Document TB-2 that comes with TradeSim describes the Universal Text Trade Database File Format, which is the ASCII, or text based trade database format that can be loaded directly into TradeSim. If you have installed TradeSim you should already have this document in your library otherwise it can be downloaded directly from the articles section of our website at:

http://www.compuvision.com.au/Articles.htm

The Professional and Enterprise Editions have the ability to read and import ASCII text file versions of the trade database. The Universal text trade database file format must meet certain requirements otherwise the file will be rejected when it is loaded into TradeSim. The data within the text file is organized into columns and is separated by at least one white space character. Some columns are required to have valid data fields whereas the optional columns should be left blank if the correct data cannot be generated. The data should be separated by at least one space character and the file name must be appended with a ".trt" file extension for TradeSim to accept it. The format of each line of data is described in the following table:

Column	Definition	Required	Format	Valid Values	Example	Place Holder
1	Symbol	Yes	Character String		ABC	-
2	Trade Position	Yes	L=long or S=short	L or S	L	-
3	Entry Date	Yes	YYYYMMDD		19970502	-
4	Exit Date	Yes	YYYYMMDD		19970725	-
5	Initial Stop**	Yes	Floating point	> 0	17.5900	0
6	Entry Price	Yes	Floating point	> 0	18.0400	-
7	Exit Price	Yes	Floating point	> 0	19.0000	-
8	Low Entry Price	Optional	Floating point	> 0	17.7500	0
9	High Entry Price	Optional	Floating point	> 0	18.2500	0
10	Low Exit Price	Optional	Floating point	> 0	18.5300	0
11	High Exit Price	Optional	Floating point	> 0	19.1600	0
12	Traded Volume	Optional	Integer	> 0	1200000	0
13	Trade Rank	Optional	Floating point	> 0	20	0
14	Point Value	Optional	Floating point	> 0	30	0
15	Initial Margin	Optional	Floating point	> 0	30000	0

For this example we shall only be interested in the required data (columns 1-7) and ignore the optional data (columns 8-15). The following Vitamin-C code to this is as follows:

```
void ProfitStop(float _PercentThreshold)
{
                                     // trade database file name with full path specifier
  const char* TradeFileName="C:\\TradeSimData\\ProfitStop.trt";
  bool InTrade=false;                // boolean variable used for trade flag
  float EntryPrice=0;                // variable used to store the entry price on trade
entry
  float ProfitThreshold;             // variable used to hold the profit threshold price

  long EntryDateIndex;               // entry date index
                                     // open file ready for writing text data. You should
                                     // specify a complete file path otherwise the file
                                     // will be created in the MetaStock directory
  FILE *fp;                          // trade database file pointer

  if(IsSymbol("AMP"))                // check if first security in the list and create a
new file
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved

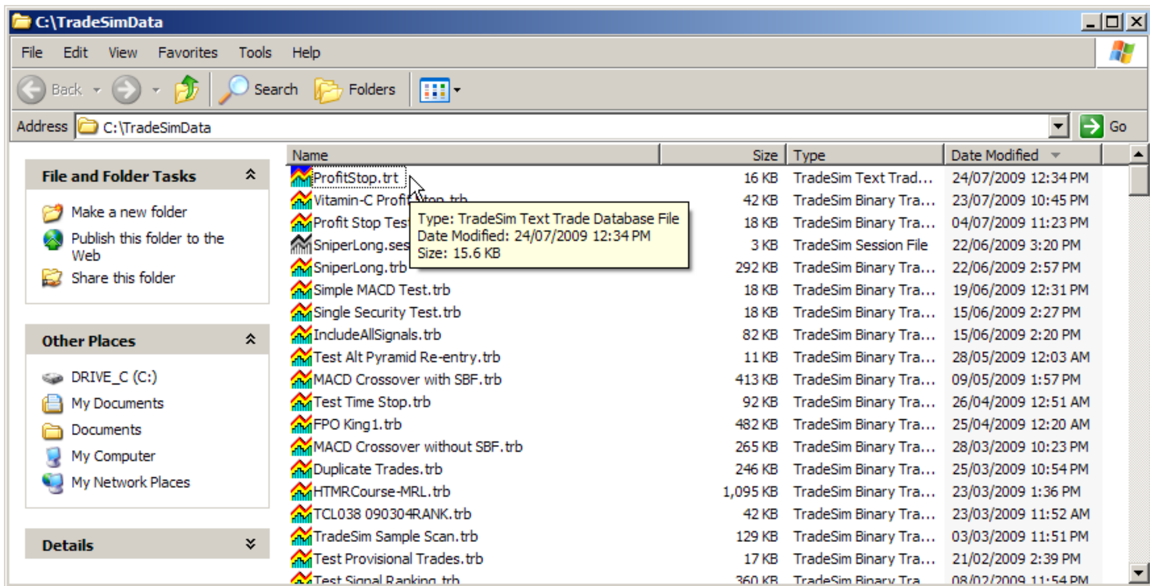
```
    fp=fopen(TradeFileName,"wt");      // open and create new file
  else
    fp=fopen(TradeFileName,"at");      // otherwise open and append data to the existing
file

  if(fp!=NULL)                         // check if successful open
  {
    for(int i=0;i<BarCount;i++)        // loop for all bars
    {
      Result[i]=0;                     // initialize result for each bar

      if(!InTrade && User1[i]>0)       // check if not in a trade and valid entry trigger
      {
        InTrade=true;                  // set the flag
        EntryPrice=Open[i];            // set the entry price
                                       // calculate the profit threshold price
        ProfitThreshold=User2[i]*(1 + _PercentThreshold/100);
        Result[i]=1;                   // mark a valid entry condition
        EntryDateIndex=i;              // save date index
      }

      if(InTrade)                      // if in the trade do the check
      {                                // check for a profit stop condition or
                                       // check for open trade
        if(User3[i]>=ProfitThreshold || i==BarCount-1)
        {
          Result[i]+=2;                // threshold reached so mark a valid exit condition
          InTrade=false;                   // reset the flag
                                       // the following writes out the data to the file
                                       // with proper column formatting
          fprintf(fp,"%-4s %c %ld %ld %10f %10f %10f\n",
            GetSymbol(),               // symbol
            'L',                       // trade position
            GetDate(EntryDateIndex),   // Entry date
            GetDate(i),                // Exit date
            0,                         // Initial Stop not used
            (double)EntryPrice,        // Entry Price
            (double)User3[i]);         // Exit price
        }
      }
    }
    fclose(fp);                        // close the file
  }
  else
  {                                    // else report an error
    ReportError("Trade file could not be opened");
  }
}
```

The trade database exploration code used in MetaStock is as follows:

```
EntryTrigger:= Ref(Cross(MACD(),Mov(MACD(),9,E)),-1); { entry trigger }
EntryPrice:=OPEN;  { entry or reference price }
ExitPrice:=CLOSE;  { exit or threshold price }


ExtFml( "VitaminC.CallScript3",
"ProfitStopTS.c",  { name of script file }
"ProfitStop(5)",   { function name and profit threshold argument }
EntryTrigger,      { User defined entry trigger }
EntryPrice,        { User defined entry or reference price }
ExitPrice          { User defined exit or threshold price }
);
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

After you run the trade database exploration code above, the file "ProfitStop.trt" trade database file should have been created in the TradeSim data directory "c:\TradeSimData"



You can double click on this file to load it up into TradeSim or load it from within TradeSim itself. As you can see, both trade database files from both methods have identical trades (including open trades). Both yield the same simulation results for a given set of trade parameters.

Trade Database Manager

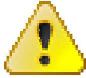
Trade	Sys ID	Pos	Symbol	Periodicity	CE	SBFTE	Entry Date-Time	Exit Date-Time	P-Group	P-Level	Re-entry Type
24	0	Long	BHP	?	Yes	Yes	31/12/2001	07/01/2002	0	0	Base
25	0	Long	RIO	?	Yes	Yes	31/12/2001	01/02/2002	0	0	Base
26	0	Long	TLS	?	Yes	Yes	31/12/2001	23/07/2009	0	0	Base
27	0	Long	BHP	?	Yes	Yes	25/01/2002	14/02/2002	0	0	Base
28	0	Long	QBE	?	Yes	Yes	05/02/2002	14/02/2002	0	0	Base
29	0	Long	NCM	?	Yes	Yes	06/02/2002	20/03/2002	0	0	Base
30	0	Long	BXB	?	Yes	Yes	05/03/2002	20/03/2006	0	0	Base
31	0	Long	WOW	?	Yes	Yes	05/03/2002	19/04/2002	0	0	Base
32	0	Long	BHP	?	Yes	Yes	11/03/2002	16/10/2003	0	0	Base
33	0	Long	CBA	?	Yes	Yes	21/03/2002	03/05/2002	0	0	Base
34	0	Long	SUN	?	Yes	Yes	22/03/2002	27/02/2004	0	0	Base
35	0	Long	WES	?	Yes	Yes	02/04/2002	07/09/2004	0	0	Base
36	0	Long	QBE	?	Yes	Yes	08/04/2002	01/11/2002	0	0	Base
37	0	Long	RIO	?	Yes	Yes	11/04/2002	17/01/2005	0	0	Base
38	0	Long	WBC	?	Yes	Yes	16/04/2002	26/04/2002	0	0	Base
39	0	Long	NAB	?	Yes	Yes	18/04/2002	20/05/2002	0	0	Base
40	0	Long	AMP	?	Yes	Yes	22/04/2002	23/07/2009	0	0	Base
41	0	Long	NCM	?	Yes	Yes	30/04/2002	07/05/2002	0	0	Base
42	0	Long	WOW	?	Yes	Yes	06/05/2002	11/11/2004	0	0	Base
43	0	Long	WPL	?	Yes	Yes	09/05/2002	17/12/2003	0	0	Base
44	0	Long	ORG	?	Yes	Yes	24/05/2002	27/08/2002	0	0	Base

Start Entry Date 17/10/2001 Stop Entry Date 20/03/2009 272 trades selected from a total of 272 trades

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Using Vitamin-C with BullCharts

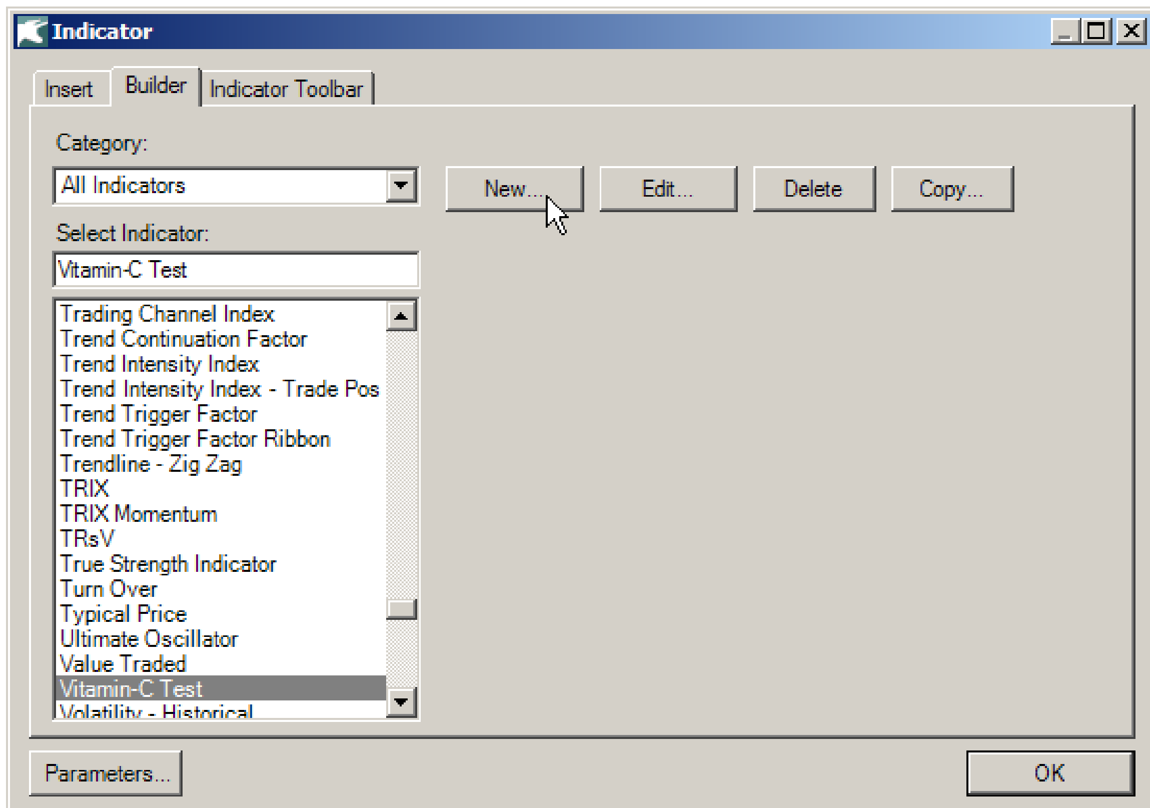
You can use BullCharts with Vitamin-C in the same way that you use BullCharts with TradeSim. Because BullCharts is compatible with MetaStock DLL plugins it is quite easy to use BullCharts with Vitamin-C. In the following example we will construct an indicator that calls the Guppy Count Back line Trailing Stop indicator that we translated from the AFL code in an earlier section on porting code from other charting platforms.



The following example assumes that the Vitamin-C installer has installed the Vitamin-C DLL plugin into the Bullcharts External Formula directory. This can be checked by displaying the external formula directory using the Windows File Explorer and checking for the existence of VitaminC.dll in the following directory:

C:\Program Files\BullSystems\BullCharts\External Function DLLs

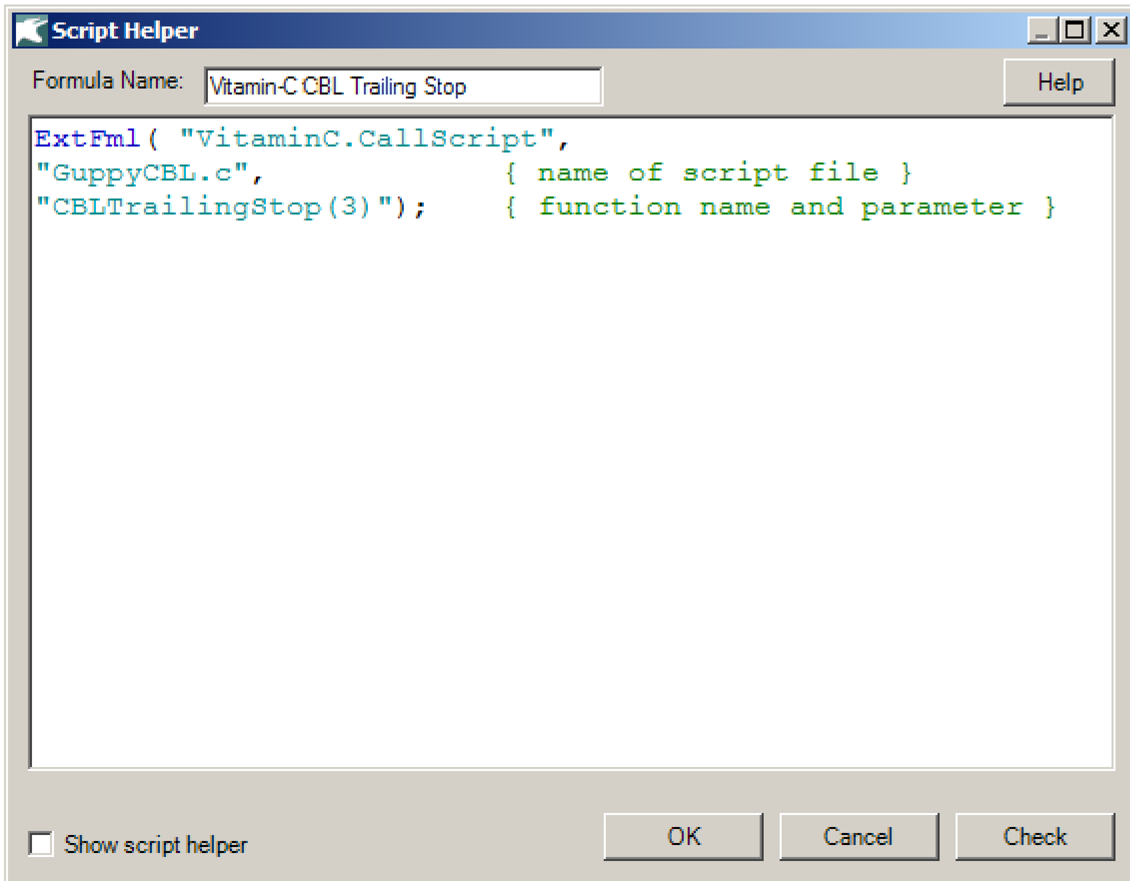
Run BullCharts and open up the indicator builder from the 'Insert' menu.



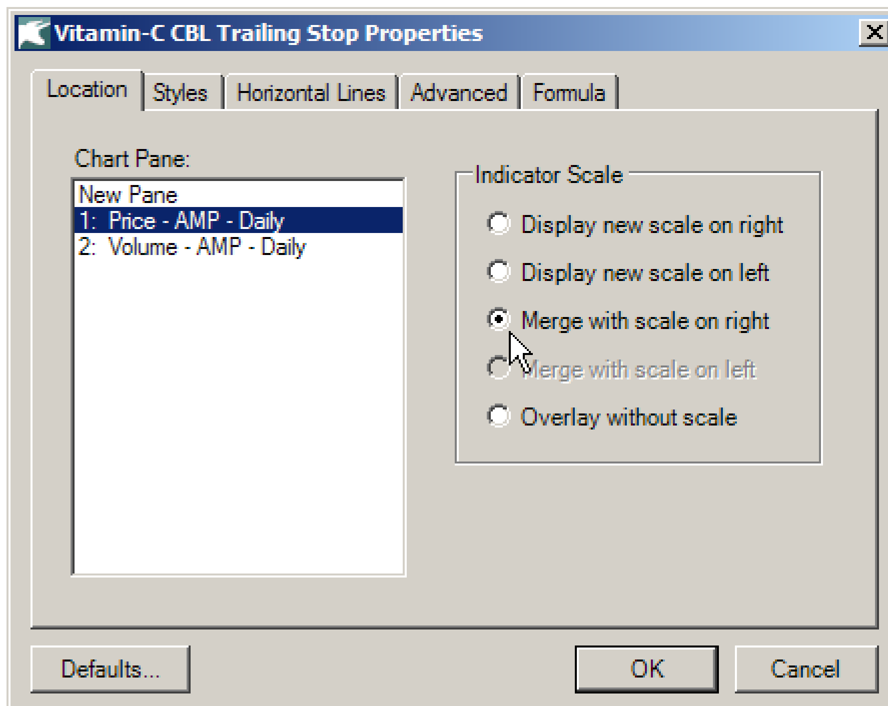
Click on 'New' and insert the following code into the editor and call it Vitamin-C CBL Trailing Stop:-

```
ExtFml( "VitaminC.CallScript",
"GuppyCBL.c",                        { name of script file }
"CBLTrailingStop(3)");               { function name and parameter }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

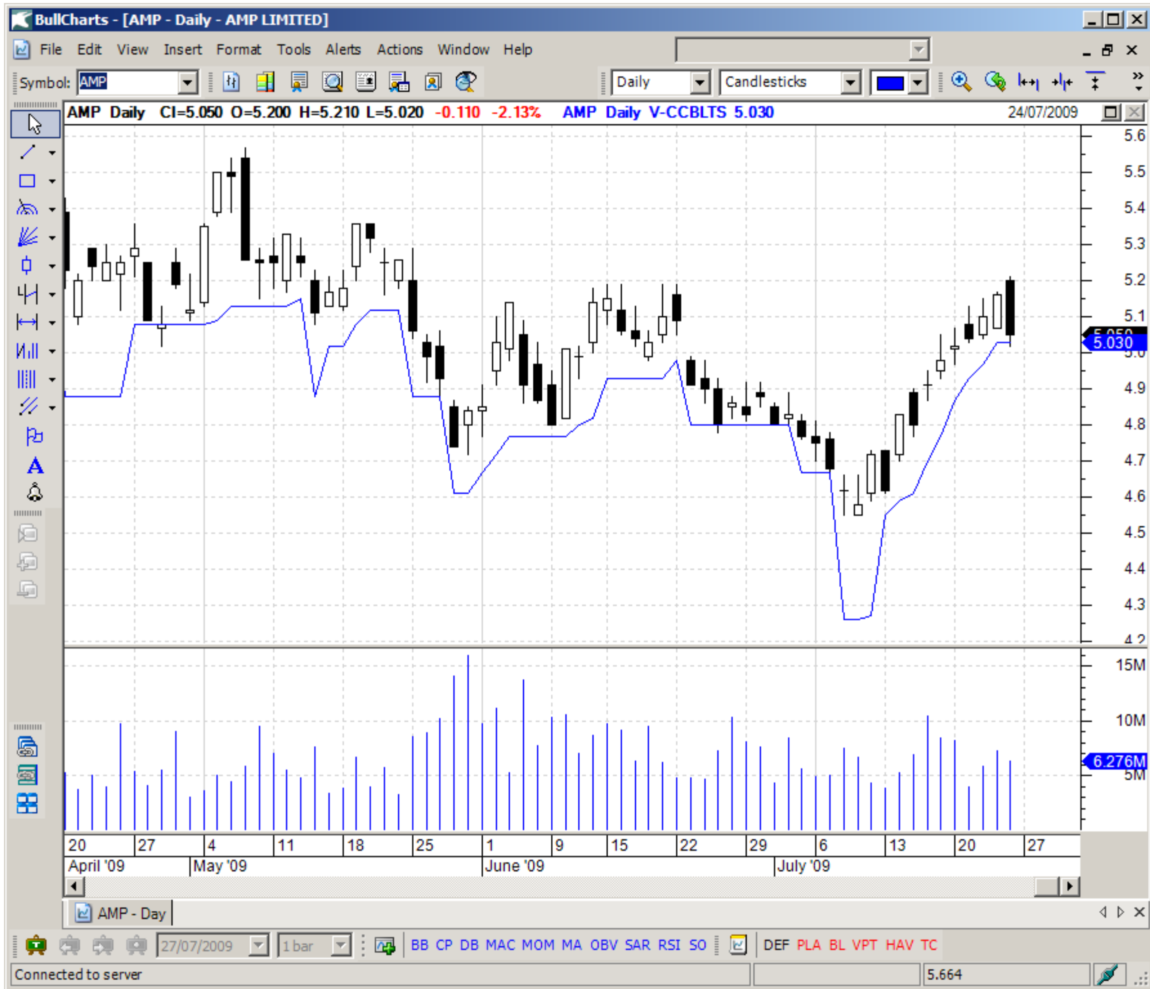


Now insert the indicator onto a chart and merge with the existing chart.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved

Shown below is the Vitamin-C Guppy CBL Trailing Stop function overlaid on a chart in BullCharts !



You will observe that the results are identical to MetaStock(shown below for comparison) with the same Vitamin-C indicator overlaid on the same chart !

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

MetaStock - [AMP LIMITED ORDINARY]
5.16000 (+0.11000)
Daily
File Edit View Insert Format Tools Window Help
Vitamin-C - Guppy CBL Test
AMP (5.13000, 5.18000, 5.10000, 5.16000, +0.11000), Vitamin- (5.03000)
5.65
5.60
5.55
5.50
5.45
5.40
5.35
5.30
5.25
5.20
5.15
5.10
5.05
5.00
4.95
4.90
4.85
4.80
4.75
4.70
4.65
4.60
4.55
4.50
4.45
4.40
4.35
4.30
4.25
4.20
15000
10000
5000
x1000
20 27 4 11 18 25 1 9 15 22 29 6 13 20 27
May
June
July
AMP
Interbank FX
Trade
For Help, press F1
3:34

Advanced Topics

This chapter presents some of the more advanced C++ features that are available from with the Vitamin-C environment and as such will be of interest only to experienced C++ programmers.

Using the Standard Template Library

The Standard Template Library (STL) provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures. For example it includes support for strings, vectors, lists, queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

- **Example 1**
  This example uses the STL basic_string type demonstrate how to concatenate literal strings and string variables.

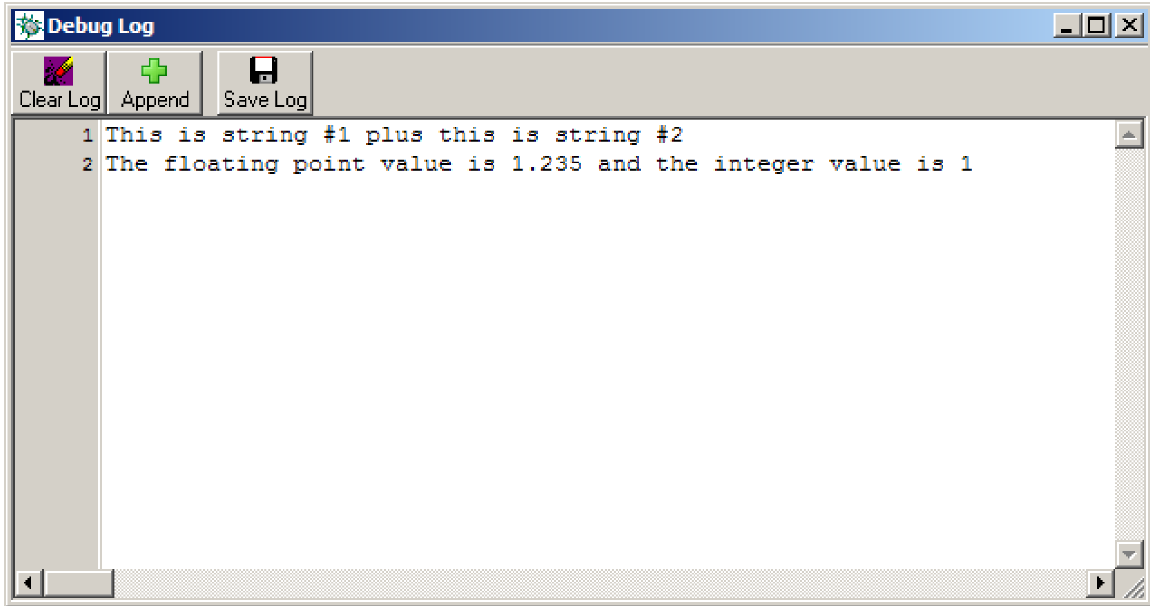
```
#include <string>

using namespace std;

void Test(void)
{
  string s1,s2,s3;

  s1="This is string #1";
  s2="this is string #2";

  s3=s1+" plus "+s2+'\n';

  dprintf("%s",s3.c_str());

  float value=1.235;

  s3="The floating point value is "+ftostring(value)+" and the integer value is
"+itostring(value);

  dprintf("%s\n",s3.c_str());
}
```

MSFL code used to call the code above:

```
ExtFml( "VitaminC.CallScript",
"STL_string.c",       { name of script file }
"Test()");            { function name and parameter }
```

The results of placing the indicator on a chart produce some text strings in the debug log as follows:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Appendix A

A Brief Introduction to C/C++

This chapter discusses some of the fundamental aspects of the C language. If you are a seasoned C programmer then you can skip this section. For newcomers we suggest that you read it because it will form the basis of understanding the material presented in the other chapters. This discussion of the C/C++ language is by no means exhaustive and only really scrapes the surface of its true capabilities. Whilst the C/C++ language gives you plenty of rope to swing on, the same bit of rope can also be used to hang yourself with, so bear this in mind before trying use every bit of the language to do something that could be done simply. At the end of this section if you would like to know more then please refer to the end of this User Guide for more references on this subject.

Adding Comments to your C-script

Just a few little house rules first. You will note that I have added some comments to the C/C++ code to make things a little bit easier to follow as well as self-documenting the code. Of course the 'C++' standard expects the Vitamin-C scripting engine to ignore these comments as long as they conform to certain requirements.

The comments are displayed in light grey type in this document, which is exactly how they are displayed in the Vitamin-C editor. All C++ comments must start with a double forward slash '//' or be enclosed with forward slash-asterix(/*) and asterix-forward(*/) slash pair. The double forward slash(//) is used for single line comments whereas the slash-asterix pairs are used for multi-lined comments.

For example the following constitute valid comments.

```
/* This is a comment
/* and so is this */
```

Forward Slash-Asterix comment pairs can span multiple lines. Double slash comments can't.

For example the following double slash comment will produce an error because I tried to continue the comment over the following lines.

```
// This is a comment
that I want to span across multiple lines
but will produce an error
```

This example is the correct way to produce multi-lined comments

```
/* This is a comment
that I want to span across multiple lines
that will be accepted */
```

What Is a Variable?

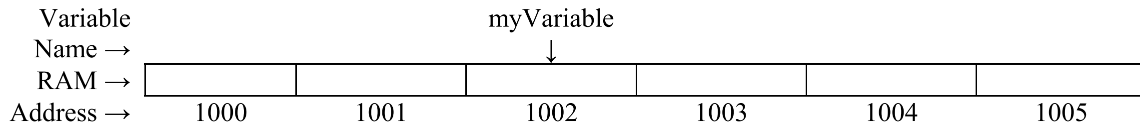
In C/C++, a *variable* is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Storing Data in Memory

Your computer's memory can be viewed as a series of postboxes. Each post box is one of many, many such boxes all lined up. Each post box—or memory location—is numbered sequentially. These numbers are known as *memory addresses*. A variable reserves one or more postboxes in which you can store a value.

Your variable's name (for example, myVariable) is a label on one of these post boxes so that you can find it easily without knowing its actual memory address. The diagram shown below is a schematic representation of this idea. As you can see from the diagram below, myVariable starts at memory address 1002. Each postbox has a fixed size of 1 byte, which is the minimum size of each post box. Depending on the size of myVariable, it can take up one or more memory addresses or bytes of data.



Allocating Storage for Variables

When you define a variable in C++, you must tell the compiler what kind of variable it is (this is usually referred to as the variable's "type"): an integer, a floating-point number, a character, and so forth. This information tells the compiler how much room to set aside and what kind of value you want to store in your variable. It also allows the compiler to warn you or produce an error message if you accidentally attempt to store a value of the wrong type in your variable (this characteristic of a programming language is called "*strong typing*").

Each postbox is one byte in size. If the type of variable you create is four bytes in size, it needs four bytes of memory, or four postboxes. The type of the variable (for example, integer) tells the compiler how much memory (how many postboxes) to set aside for the variable.

Size of Variables

On any one computer, each variable type takes up a single, unchanging amount of room. That is, an integer might be two bytes on one machine and four on another, but on either computer it is always the same, day in and day out. Single characters—including letters, numbers, and symbols—are stored in a variable of type **char**. A **char** variable is most often one byte long.

For smaller integer numbers, a variable can be created using the short type. A **short** integer is two bytes on most computers, a **long** integer is usually four bytes, and an integer (without the keyword **short** or **long**) is usually two or four bytes. You'd think the language would specify the exact size that each of its types should be; however, C++ doesn't. All it says is that a short must be less than or equal to the size of an **int**, which, in turn, must be less than or equal to the size of a **long**.

The size of an integer is determined by the processor (16 bit, 32 bit, or 64 bit) and the compiler you use. On a 32-bit computer with an Intel Pentium processor, using modern compilers, integers are *four* bytes.

Signed and Unsigned Integers

All integer types come in two varieties: **signed** and **unsigned**. Sometimes, you need negative numbers, and sometimes you don't. Any integer without the word "unsigned" is assumed to be **signed**. **signed** integers can be negative or positive. **unsigned** integers are always positive.

Integers, whether **signed** or **unsigned** are stored in the same amount of space. Because of this, part of the storage room for a **signed** integer must be used to hold information on whether the number is negative or positive. The result is that the largest number you can store in an **unsigned** integer is twice as big as the largest positive number you can store in a **signed** integer.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

For example, if a short integer is stored in two bytes, then an **unsigned short** integer can handle numbers from 0 to 65,535. Alternatively, for a **signed short**, half the numbers that can be stored are negative; thus, a **signed short** can only represent positive numbers up to 32,767. The **signed short** can also, however, represent negative numbers giving it a total range from –32,768 to 32,767.

Volatile and Non-Volatile storage

Notice that variables are used for temporary storage. When you exit a program or turn the computer off, the information in variables is lost. This is what is meant by volatile, non-permanent or temporary storage. Permanent or non-volatile storage stores data either to non-volatile memory, or to a file on a disk. For example the global parameters in the Vitamin-C Integrated Development Environment (IDE) are stored in a file called 'VitaminC.ini'. Each time Vitamin-C is run this data is read from the hard drive into memory and vice-versa when the program is terminated, so even though the variables in the program used to store the initialization data are temporary the action of saving this information to disks ensures that this information is never lost even when the program and/or the computer are no longer running.

Keywords

Some words are reserved by C++, and you cannot use them as variable names. These keywords have special meaning to the C++ compiler. Keywords include if, while, for, and main. A list of keywords defined by C++ is presented in the following table.

The C++ Keywords							
asm	auto	bool	break	case	catch	char	class
const	const_cast	continue	default	delete	do	double	dynamic_cast
else	enum	explicit	export	extern	false	float	for
friend	goto	if	inline	int	long	mutable	namespace
new	operator	private	protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast	struct	switch	this
throw	true	try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while	template	
In addition, the following words are reserved:							
And	and_eq	bitand	bitor	compl	not	not_eq	or
or_eq	xor	xor_eq					

Fundamental Variable Types

Several variable types are built in to C++. They can be conveniently divided into integer variables (the type discussed so far), floating-point variables, and character variables. Floating-point variables have values that can be expressed as fractions—that is, they are real numbers. Character variables hold a single byte and are generally used for holding the 256 characters and symbols of the ASCII and extended ASCII character sets.

The ASCII character set is the set of characters standardized for use on computers. ASCII is an acronym for American Standard Code for Information Interchange. Nearly every computer operating system supports ASCII, although many support other international character sets as well.

The types of variables used in C++ programs are described in the below. This table shows the variable type, how much room the type generally takes in memory, and what kinds of values can be stored in these variables. The values that can be stored are determined by the size of the variable types. Shown in the table below is a list of the fundamental types available in the C++ language. The ones highlighted with pale green will be the ones most commonly used in your Vitamin-C programs whereas the ones highlighted with pale orange will be less likely used and the un-highlighted ones will be rarely used if at all.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Type	Also known as	Size	Allowable Values
bool		1 byte	true or false
char		1 byte	256 character values
unsigned short int	unsigned short	2 bytes	0 to 65,535
short int	short	2 bytes	–32,768 to 32,767
int		4 bytes	–2,147,483,648 to 2,147,483,647
unsigned int	unsigned	4 bytes	0 to 4,294,967,295
unsigned long int	unsigned long	4 bytes	0 to 4,294,967,295
long int	long	4 bytes	–2,147,483,648 to 2,147,483,647
float		4 bytes	1.2e–38 to 3.4e38
double		8 bytes	2.2e–308 to 1.8e308

Storage Classes

Variables have one of the following storage classes:

auto,
register,
external,
static,
external static

auto variables are created each time the function containing them is evoked, and they vanish each time the function finishes. The others last for the duration of the program.

Declaring Variables

A variable can be singled-valued (a scalar variable) or contain several values (an array). A variable has a type (**int, char,** etc.) and *storage class* (**auto, static,** etc.). See above. A declaration statement declares these attributes. (By default, a variable is storage class **auto** if declared inside a function and **extern** if declared outside a function.) A general form for declaring a scalar variable is:

storage-class   *type-specifier*   *variable-name*

An array is indicated by following the variable name with square brackets containing the number of elements. This should not be confused with the Vitamin-C built in Array type, which is a derived Class type that encapsulates the price arrays, used in MetaStock. You'll learn a bit more about this later on.

Case Sensitivity

C/C++ is case sensitive. In other words, uppercase and lowercase letters are considered to be different. A variable called age is different from Age, which is different from AGE.

Examples of variable declarations are:

```
int X;                 // declare an integer variable
float Y=2.9;           // declare a floating variable and initialize it
bool flag=false;       // declare a boolean variable amd initialize it
int x,y,z;             // declare three integer variables
char periodicity='W';  // declare a character variable and initialize it
char symbol[]="AMP";   // declare a character array and initialize it
int scores[20];        // declare an array of 20 integers
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Typical Program Forms

A simple program used in Vitamin-C consists of at least one function. The statements of the function are enclosed between opening and closing braces and there can be more than one statement as in the following example:

```
void FunctionName()                        // function name and declaration
{                                          // opening brace
  int i;                                   // variable declaration statement

  for(i=0;i<BarCount;i++)                  // main body of code
    Result[i]=(Open[i]+Close[i])/2;        // single statement
}                                          // closing brace
```

Programs may include preprocessor directives and more than one function.

Statements

Simple statements consist of an instruction followed by a semicolon. Some examples of statements are shown below:

```
float x;                                   // declaration statement
x = 3.1415926;                             // assignment statement
printf("number = %f",x);                   // function call statement
```

Structured statements typically consist of a keyword (if, while, for, do-while, etc) followed by a condition within parentheses.

A statement block consists of one or more statements enclosed in braces. It counts as one statement and is used in structured statements to allow more than one action to be included in the statement. The following example demonstrates a statement block.

```
for(i=0;i<BarCount;i++)
{                                        // start of statement block
  average=(Open[i]+Close[i])/2;
  Result[i]=average;
  dprintf("Average[%d]=%f\n",i,average);
}                                        // end of statement block
```

It is possible to have more than on statement on each line separated by semicolons although this is not recommended out of consideration for keeping the code tidy as well as for debugging purposes. For example the above code statement block can be re-written as:

```
for(i=0;i<BarCount;i++)
{
  average=(Open[i]+Close[i])/2;Result[i]=average;dprintf("Average[%d]=%f\n",i,average);
}
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Anatomy of a function.

A function is a section of code, separate from the main program that, perform a single, well-defined task. Function definitions follow one another. Do not embed one function definition within another function definition. Execution starts at the beginning of the function name.

Functions with arguments have the arguments declared after the function name and enclosed within parenthesis. The general form of function is shown in the figure below:

	Return Type	Function Name	Formal Parameter List
Function Header	`float`	`AddTwoNumbers`	`(float _Number1,float _Number2)`
Function Body	`{ float sum; sum=_Number1 + _Number2; return(sum); }`		

```
float AddTwoNumbers(float _Number1,float _Number2)
{
  float sum;

  sum=_Number1 + _Number2;

  return(sum);
}
```

Functions can be grouped into two categories: functions that don't have return values and functions that do have return values. Functions without return values are termed type void functions and have the following general form:

```
void FunctionName(parameterlist)
{
  statement or statement block
  return;            // optional
}
```

A function with a return value produces a value that it returns to the function that called it. In other words, if the function returns the square root of 9 (`sqrt(9)`), then this becomes replaced by the value 3. Such a function is declared as having the same type as the value it returns. The general form of this function is:

```
typename FunctionName(parameterlist)
{
  statement or statement block
  return(value);    // value is of type typename
}
```

Functions with return values require that you use a return statement so that the value is returned to the calling statement. The value returned can be a constant, a variable, or a more general expression.

Program Flow Control

Relational Expressions

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Relational expressions are used to make decisions and control the way a computer executes its instructions. C/C++ provides six relational operators for comparing numbers. Because characters are represented by their ASCII code, you can use these operators with characters, too. But they don't work with character strings. Each relational expression reduces to the value *1* if the comparison is true and to *0* if the comparison is false, so they are well suited for use in loop test expression. The following table summarizes these operators.

Operator	Meaning
<	Is less than
<=	Is less than or equal to
==	Is equal to
>	Is greater than
>=	Is greater than or equal to
!=	Is not equal to

```
x < 5             // x is less than 5 ?
x >= 2 && x < 5   // is x greater than and equal to 2 and less than 5
x == 6 && y != 10 // is x equal to 6 or y is not equal to 10
```

The relational operators have a lower precedence than the arithmetic operators. That means the following expression

```
x + 3 > y - 2     // expression 1
```

corresponds to

```
(x + 3) > (y - 2) // expression 2
```

and not the following

```
x + (3 > y) - 2   // expression 3
```

Because the expression (3 > y) is either 1 or 0, expressions 2 and 3 both are valid. But most of us would want expression 1 to mean expression 2, and that is what C++ does.

❑ A common pitfall

Usually more times than not the equality operator(==) is substituted with the assignment operator(=). For example the following expression compares X with 7 and returns 1 if it is true otherwise returns 0 if it is false.

```
X == 7            // Is X equal to 7 ?
```

However inadvertently you might use an equals(=) sign by itself rather than the equality operator.

```
X = 7             // This inadvertent assignment results in true always
```

What happens is that X is assigned the value of seven and the whole expression has the value of 7 because that's the value of the left-hand-side. Because the value of the left hand side of the expression is non zero it

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

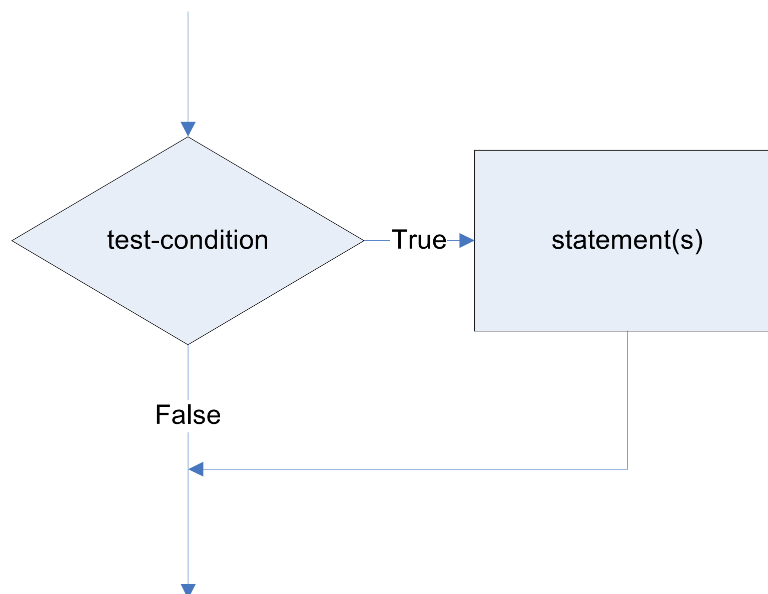
is always true from a logical perspective. If this expression was used, as a test condition in a loop then the loop would never exit and run forever or until the computer was switched off.

The if Statement

When a C program must choose whether or not to take a particular action, you can use the **if** statement. The **if** statement comes in two forms: **if** and **if…else**. The **if** statement directs a program to execute a statement or statement block if a test condition is true and to skip that statement or block if the condition is false. The syntax is as follows:

```
if (test-condition)
  statement or statement block
```

Diagrammatically the **if** statement can be represented by the following flowchart:



A non-zero *test-condition* (true) causes the program to execute statement, which can be a single statement or a block. A zero *test-condition* (false) causes the program to skip *statement*. The entire **if** construction counts as a single statement. By the way, the *statement* portion can't be a single declaration statement; declarations have to be placed where they can't be skipped. However a statement block enclosed with braces can contain declarations used as temporary variables. Most often, test-condition will be a relational expression like those used to control loops.

An example of an **if** statement is:

```
if(GetDate(i) > 20081103)
  Result[i]=lastvalue;
```

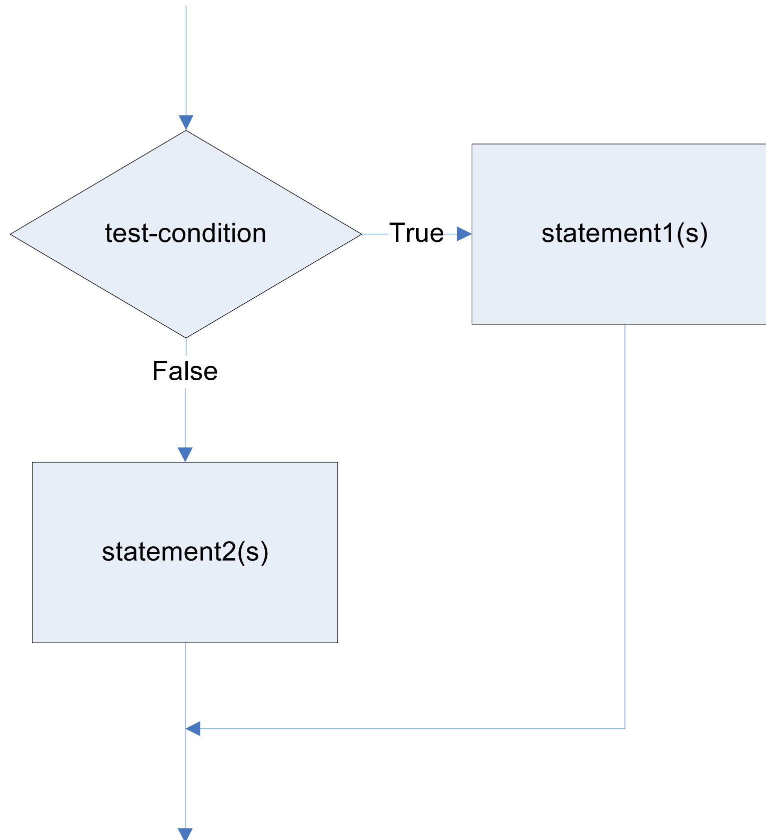
The if…else Statement

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The **if** statement lets a program decide whether a particular statement or block is executed. The **if…else** statement lets a program decide which of two statements or blocks is executed. It is an invaluable statement for creating alternative courses of action. The if else statement has the general form:

```
if (test-condition)
   1st statement or statement block
else
   2nd statement or statement block
```

Diagrammatically the **if…else** statement can be represented by the following flowchart:



If *test-condition* is nonzero(true), the program executes *statement1* and skips over *statement2*. Otherwise, when *test-condition* is zero (false), the program skips *statement1* and executes *statement2* instead.

An example of an **if…else** statement is:

```
if(GetDate(i) > 20081103)
  Result[i]=6;
else
  Result[i]=Close[i];
```

The if…else…if…else Statement

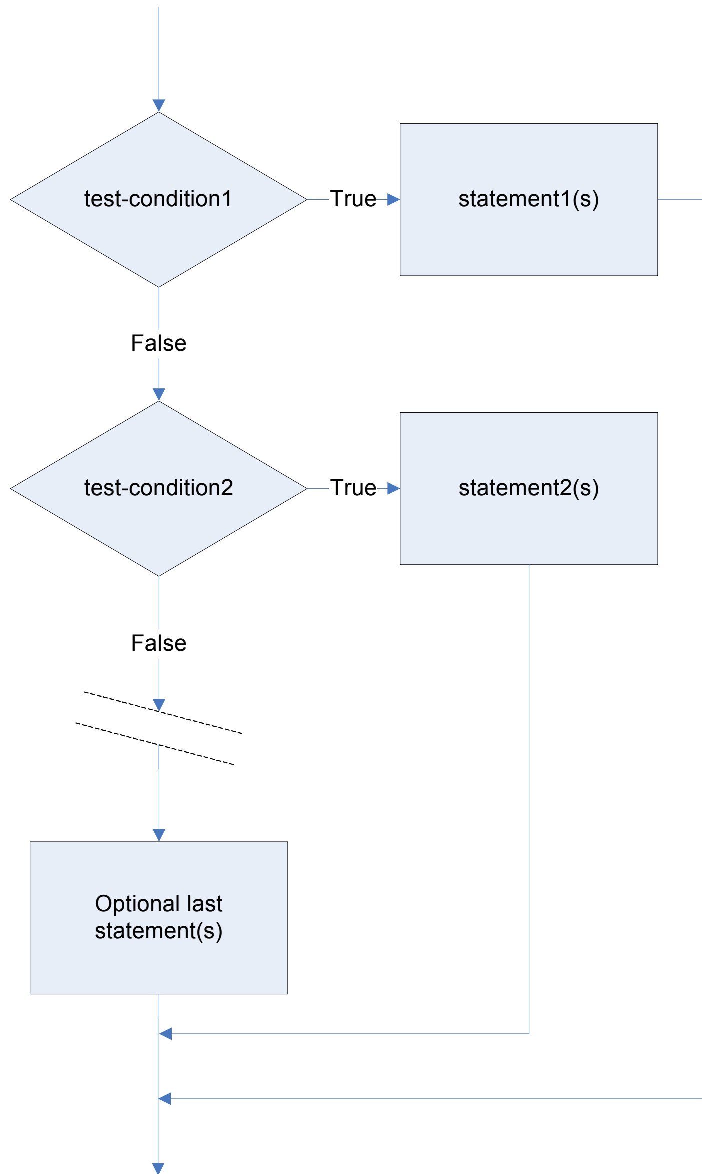
The **if…else…if** statement lets a program decide which of many statements or blocks is executed. It is an invaluable statement for creating many courses of action. The **if…else…if…else** statement has the general form:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
if (test-condition1)
   1st statement or statement block
else if (test-condition2)
   2nd statement or statement block
……….. 
else (optional-test-condition)
   last statement or statement block
```

It is important to note that the last **else** statement does not have an **if** statement proceeding it.

Diagrammatically the **if…else…if…else** statement can be represented by the following flowchart:



- **Example 1**

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
if(date > 20081103)
  Result[i]=6;

  Result[i]=Close[i];
```

- **Example 2**

```
if(date > 20081103)
  Result[i]=6;

  Result[i]=Close[i];
else
  Result[i]=(High[i]+Low[i]+Open[i]+Close[i])/4;
```

The ?: Operator

C++ has an operator that can often be used instead of the **if…else** statement. This operator is called the conditional operator, written ?:, and is the only C++ operator that requires three operands. It is what is called a ternary operator as opposed to binary and unary operators. The general form looks like this:

```
expression1 ? expression2 : expression3
```

If *expression1* is true, then the value of the whole conditional expression is the value of *expression2*. Otherwise, the value of the whole expression is the value of *expression3*.

We used the conditional operator in an earlier chapter when we ported some AFL code across to run under Vitamin-C. In this case we use the conditional operator to replace the AFL Max(x,y) function used to return the maximum of two values. For example, to calculate the maximum of two values use the following:

```
Maximum=(x > y) ? x : y;
```

The switch Statement

You can extend an **if..else…if…else** sequence to handle multiple alternatives, but the switch statement handles selecting a choice from an extended list more easily. Here's the general from for a **switch** statement:

```
switch(integer-expression)
{
  case label1 : statement(s)
  case label2 : statement(s)
  case label3 : statement(s)
  ...
  default: statement(s)
}
```

A **switch** statement acts as a routing device, re-directing the computer to execute a particular statement or block of statements based on the value of the *integer-expression*. For example, if *integer-expression* has the value 4, the program will go to the line where the label matches the integer-expression, which in this case is the **case 4:** label. The value *integer-expression*, as the name suggests, must be an expression that reduces to an integer value. Also, each label must be an integer constant expression. It cannot be a variable of any kind. Most often labels are simple **int** or **char** constants such as 1 or 'q'. If *integer-expression* doesn't match any of the labels, the program jumps to the line labeled **default.** The **default** label is optional. If you omit it and there is no match, the program jumps to the next statement following the **switch**.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

It is important to note that a potential trap can arise because the **case** label functions only as a line label, not as a boundary between choices. That is, once a program jumps to a particular line in a **switch**, it then sequentially executes *all* the statements following that line in the **switch** *unless* you explicitly direct it otherwise. Execution does NOT automatically stop at the next **case**. To make execution stop at the end of a particular group of statements, you must use the **break** statement. This causes execution to jump to the statement following the **switch**.

- **Example 1**

```
for(int i=0;i<count;i++)                    // loop for all bars
{
  Result(i)=0;                              // initialize result for each bar

  switch(_method)
  {
    case SIMPLE:                            // simple moving average
      sum+=User1(i)-User1(i-_period);
      Result(i)=sum/_period;
      break;
    case EXP:                               // exponential moving average
      Result(i)=User1(i)*K+Result(i-1)*(1-K);
      break;
  }
}
```

- **Example 2**

Their maybe situations where you want the same code to be executed for a range of labels rather than a particular label. In this case you just list all of the labels in sequence followed by the respective code and break statement as in the following example:

```
switch(inchar)
{
  case 'A':
  case 'B':
  case 'C':
    alpha=true;
    break;
  case '0':
  case '1':
  case '2':
    numeric=true;
    break;
}
```

Logical Expressions

Quite often you will need to test for more than one condition. For instance you may want an entry price to be between the low of the day and the opening price in order to get filled in the market. To meet these kind of needs the C language provides three logical operators to combine or modify existing expressions. The operators are the logical OR, written as ||, logical AND, written as &&, and logical NOT, written as !. We will now describe these in more detail.

The Logical OR Operator ||

This operator combines two expressions into one. If either, or both, of the original expressions is nonzero (true), the resulting expression has the value 1 (true).

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

The Truth Table for Expr1 || Expr2		
	Expr1 == true	Expr1 == false
Expr2 == true	1 (true)	1 (true)
Expr2 == false	1 (true)	0 (false)

Here are some examples:

```
6==6 || 7==9   // true because the first expression is true
7>9 || 16<20   // true because the second expression is true
3==3 || 11==11 // true because both expressions are true
5!=5 || 7!=7   // false because both expressions are false
```

Because the || has a lower precedence than the relational operators, we don't need to use parentheses in these expressions.

The Logical AND Operator &&

The logical AND operator, written &&, also combines two expressions into one. The resulting expression has the value 1(true) only if both of the original expressions are true.

The Truth Table for Expr1 && Expr2		
	Expr1 == true	Expr1 == false
Expr2 == true	1 (true)	1 (false)
Expr2 == false	1 (false)	0 (false)

Here are some examples:

```
6==6 && 7==9   // false because the second expression is false
7>9 && 16<20   // false because the first expression is false
3==3 && 11==11 // true because both expressions are true
5!=5 && 7!=7   // false because both expressions are false
```

Because the && has a lower precedence than the relational operators, we don't need to use parentheses in these expressions.

The Logical NOT Operator !

The ! operator negates, or reverses the truth value of, the expression following it. That is if *expression* is true, then *!expression* is false, and vice versa. More precisely, if *expression* is nonzero, then *!expression* is zero. And if *expression* is zero, then *!expression* is 1.

The Truth Table for !Expr		
	Expr == true	Expr == false
!Expr	0 (false)	1 (true)

Loops

Loops allow you to repeat tasks a multiple number of times. These tasks can be simple or complex. There are essentially three types of loop constructs in C++. These are the for-loop, while, and do-while loop. Each will be discussed in detail in the next few sections.

The for-loop

The for-loop provides a step-by-step framework for performing repeated actions. It is the most commonly used form of the looping constructs in C++. The usual parts of a for loop handle these steps:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

- Setting an initial value
- Performing a test to see if the loop should continue
- Executing the loop actions
- Updating value(s) used for the test

The initialization, test, and update actions constitute a three-part control section enclosed in parentheses. Each part is an expression, and semicolons are used to separate the expressions from each other. The statement following the control section is called the *body* of the loop, and is executed as long as the test-expression remains true:

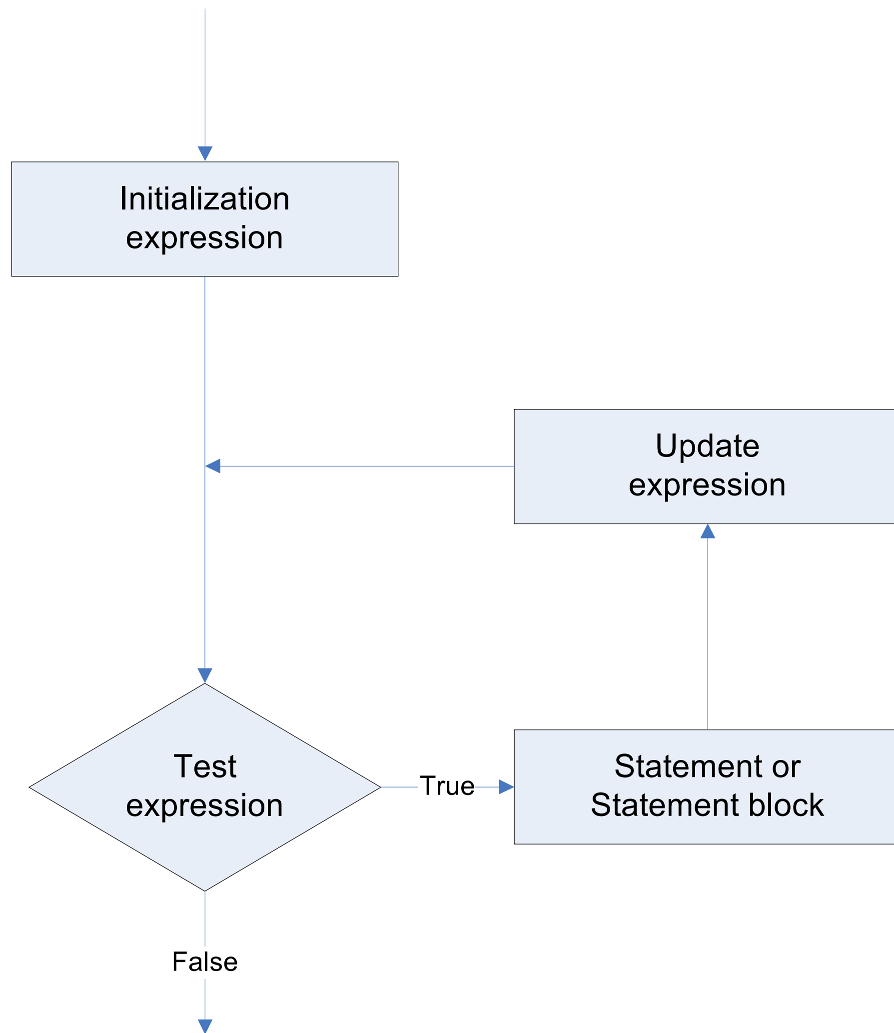
```
for(initialization; test-expression; update-expression)
  body
```

C++ syntax counts a complete for statement as a single statement, even though it may incorporate one or more statements in the *body* portion.

The loop evaluates *initialization* just once. Typically, programs use this expression to set a variable to a starting value, then use the variable to count loop cycles.

The *test-expression* determines whether the loop body gets executed. Typically, this expression is a relational expression, that is, one that compares two values. If the comparison is true, then the program executes the loop body. Actually, C++ doesn't limit *test-expression* to true or false comparisons. You can use any expression. If the expression evaluates to zero, the loop terminates. If the expression evaluates to nonzero, the loop continues. The following flowchart illustrates the way a for-loop operates.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



- **Example**

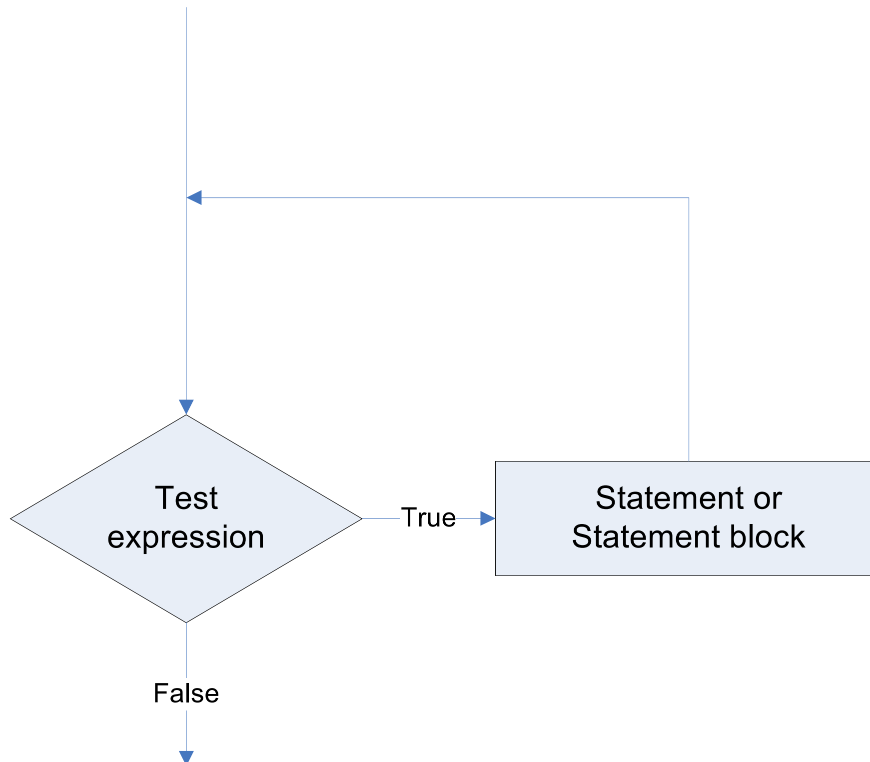
```
for(int i=0;i<BarCount;i++)
  Result[i]=(Close[i]+Open[i])/2;
```

The while loop

The while loop is a for-loop with the initialization and update parts removed. It has the following syntax and flowchart forms:

```
while(test-expression)
  body
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



Firstly a program evaluates the *test-condition* expression. If the expression evaluates to a nonzero value(true), the program executes the statement(s) in the *body*. As with a for loop, the *body* consists of a single statement or of a block defined by paired braces. After finishing with the *body*, the program returns to the test-condition and reevaluates it, If the condition is nonzero, the program executes the *body* again. This cycle of testing and execution continues until the *test-condition* evaluates to 0(false).

If you want the loop to terminate eventually, something with the loop *body* must do something to affect the *test-condition* expression other the loop will repeat forever. For example, the loop could increment a variable used in the test condition, which will eventually reach a certain value for which the *test-condition* will evaluate as false and the loop will terminate.

Like the for loop, the while loop is an entry-condition loop. Thus if *test-condition* evaluates as false to begin with, the program never executes the *body* of the loop.

- **Example**

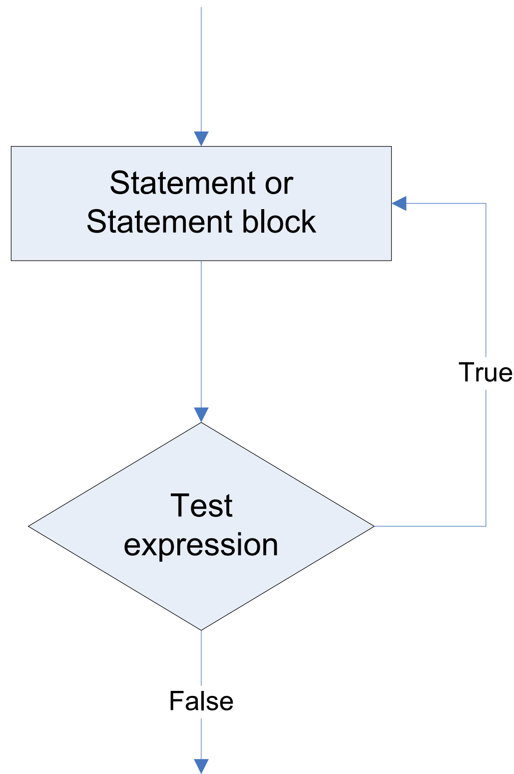
```
int i=0;
while(i<BarCount)
{
  Result[i]=(Close[i]+Open[i])/2;
  i++;
}
```

The do…while loop

The do while loop is different to the other two loop constructs because it is an exit-condition loop. This means that the *body* of the loop is always executed at least once because the test expression is performed at the end of the loop. If the condition evaluates to false, the loop terminates; otherwise it repeats itself again before the test expression is evaluated again. It has the following syntax and flow chart forms:

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
do
  body
while(test-expression)
```



If you want the loop to terminate eventually, something with the loop *body* must do something to affect the *test-condition* expression other the loop will repeat forever. For example, the loop could increment a variable used in the test condition, which will eventually reach a certain value for which the *test-condition* will evaluate as false and the loop will terminate.

- **Example**
  Read from keyboard and echo it to the consol until the user presses the carriage return.

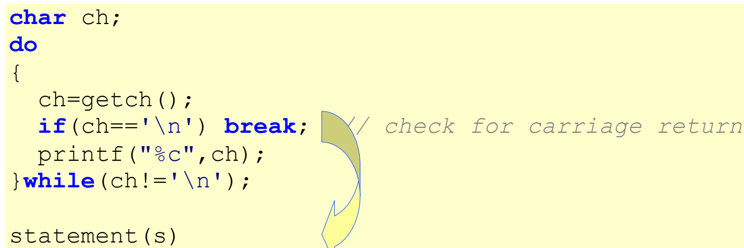
```
char ch;
do
{
  ch=getch();
  printf("%c",ch);
}while(ch!='\n');
```

The break statement

You were briefly introduced to the break statement in an earlier section as a means of jumping or breaking out of a switch statement. Break can also be used to break out of a loop typically when a condition is met. Typically you may have a while or for loop but you may want to break out of the loop before the test condition is tested.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

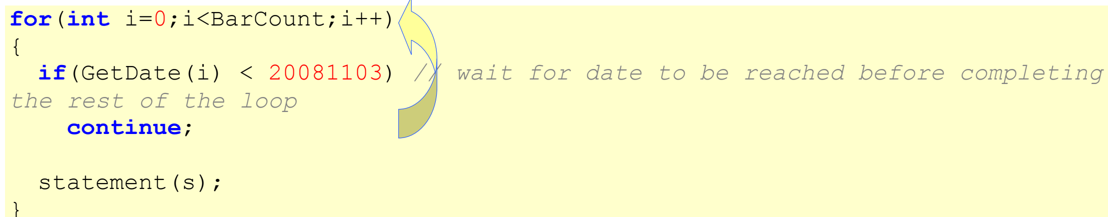
- **Example**
  Read from keyboard and echo it to the consol until the user presses the carriage return. If the user presses the carriage return first then the program aborts and does not print the carriage return.



```
char ch;
do
{
  ch=getch();
  if(ch=='\n') break; // check for carriage return
  printf("%c",ch);
}while(ch!='\n');

statement(s)
```

The continue statement

The continue statement is used in loops and causes a program to skip the body of the loop and continue back up to the start of the loop. You should make sure there is a means of exiting the loop such as incrementing a loop iterator otherwise the loop may not terminate and hang your program.



```
for(int i=0;i<BarCount;i++)
{
  if(GetDate(i) < 20081103) // wait for date to be reached before completing
the rest of the loop
    continue;

  statement(s);
}
```

Appendix B

General Forms of CallScript

The different forms of the CallScript function only vary by the number of data array arguments. Use the function call, which is appropriate for your Vitamin-C script.

CallScript (0 user array arguments)

```
ExtFml( "VitaminC.CallScript1",    { Name of external function }
        "SCRIPT_FILENAME",         { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",     { 2. Function Name and arguments }
);
```

CallScript1 (1 user array arguments)

```
ExtFml( "VitaminC.CallScript1",    { Name of external function }
        "SCRIPT_FILENAME",         { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",     { 2. Function Name and arguments }
        UserArrayArg1              { 3. User Supplied Array }
);
```

CallScript2 (2 user array arguments)

```
ExtFml( "VitaminC.CallScript2",    { Name of external function }
        "SCRIPT_FILENAME",         { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",     { 2. Function Name and arguments }
        UserArrayArg1              { 3. User Supplied Array }
        UserArrayArg2              { 4. User Supplied Array }
);
```

CallScript3 (3 user array arguments)

```
ExtFml( "VitaminC.CallScript3",    { Name of external function }
        "SCRIPT_FILENAME",         { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",     { 2. Function Name and arguments }
        UserArrayArg1              { 3. User Supplied Array }
        UserArrayArg2              { 4. User Supplied Array }
        UserArrayArg3              { 5. User Supplied Array }
);
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

CallScript4 (4 user array arguments)

```
ExtFml( "VitaminC.CallScript4",   { Name of external function }
        "SCRIPT_FILENAME",        { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",    { 2. Function Name and arguments }
        UserArrayArg1             { 3. User Supplied Array }
        UserArrayArg2             { 4. User Supplied Array }
        UserArrayArg3             { 5. User Supplied Array }
        UserArrayArg4             { 6. User Supplied Array }
);
```

CallScript5 (5 user array arguments)

```
ExtFml( "VitaminC.CallScript5",   { Name of external function }
        "SCRIPT_FILENAME",        { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",    { 2. Function Name and arguments }
        UserArrayArg1             { 3. User Supplied Array }
        UserArrayArg2             { 4. User Supplied Array }
        UserArrayArg3             { 5. User Supplied Array }
        UserArrayArg4             { 6. User Supplied Array }
        UserArrayArg5             { 7. User Supplied Array }
);
```

CallScript6 (6 user array arguments)

```
ExtFml( "VitaminC.CallScript6",   { Name of external function }
        "SCRIPT_FILENAME",        { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",    { 2. Function Name and arguments }
        UserArrayArg1             { 3. User Supplied Array }
        UserArrayArg2             { 4. User Supplied Array }
        UserArrayArg3             { 5. User Supplied Array }
        UserArrayArg4             { 6. User Supplied Array }
        UserArrayArg5             { 7. User Supplied Array }
        UserArrayArg6             { 8. User Supplied Array }
);
```

CallScript7 (7 user array arguments)

```
ExtFml( "VitaminC.CallScript6",   { Name of external function }
        "SCRIPT_FILENAME",        { 1. Vitamin-C script filename including path}
        "FUNC_NAME_ARGUMENTS",    { 2. Function Name and arguments }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
        UserArrayArg1            { 3. User Supplied Array }
        UserArrayArg2            { 4. User Supplied Array }
        UserArrayArg3            { 5. User Supplied Array }
        UserArrayArg4            { 6. User Supplied Array }
        UserArrayArg5            { 7. User Supplied Array }
        UserArrayArg6            { 8. User Supplied Array }
        UserArrayArg7            { 9. User Supplied Array }
)
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Appendix C

The Array Class Type

The Array class is a special class type used to define new arrays of floating point numbers or to encapsulate existing arrays of floating point numbers such as the closing, opening, low, high, volume floating point price series in MetaStock so that they are easily accessible and can be manipulated within the Vitamin-C environment, and without having to worry about the idiosyncrasies of accessing this information at the lower level using the MetaStock Developers Kit.

The Array class allows encapsulation of existing MetaStock price and data arrays for an intuitive and streamlined program interface. Both array indexing and array manipulation are available to the programmer and can be used independently or together to get the most flexibility. As well some member and friend functions have hand optimized machine code (Ultraboost), which takes advantage of the on-chip data caches as well as the CPU and Numeric Processor instruction set allows super fast array processing and manipulation !

Available Operators

The following table shows the available operators that are built into the Array class structure. All overloaded operators have been written in assembly language for maximum speed and performance by taking advantage of the CPU and Numeric Processor Instruction set. The following definitions are used:

A = Array object,
B = Array object,
C = Array object,
K = Floating point or Integer Constant,
i = integer used for Array element indexing

Array Operators					
Unary	Operator	Syntax	Ultraboost	Examples	Equivalent MSFL
Negation	-	-A	Yes	-Close	-
Inversion (NOT)	!	!A	Yes	!EntryTrigger	N/A
Binary	Operator	Syntax	Ultraboost	Examples	Equivalent MSFL
Addition	+	A+B	Yes	Open+Close	+
	+	A+K	Yes	Open+2	+
	+	K+A	Yes	2+Open	+
Subtraction	-	A-B	Yes	Open-Close	-
	-	A-K	Yes	Open-2	-
	-	K-A	Yes	2-Open	-
Division	/	A/B	Yes	Open/Close	/
	/	A/K	Yes	Open/2	/
	/	K/A	Yes	2/Open	/
Multiplication	*	A*B	Yes	Open*Close	*
	*	A*K	Yes	Open*2	*
	*	K*A	Yes	2*Open	*
Greater Than	>	A>B	Yes	Open>Close	>
	>	A>K	Yes	Open>2	>
	>	K>A	Yes	2>Open	>

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Greater Than or Equal to	>=	A>=B	Yes	Open>=Close	>=
	>=	A>=K	Yes	Open>=2	>=
	>=	K>=A	Yes	2>=Open	>=
Less Than	<	A<B	Yes	Open<Close	<
	<	A<K	Yes	Open<2	<
	<	K<A	Yes	2<Open	<
Less Than or Equal to	<=	A<=B	Yes	Open<=Close	<=
	<=	A<=K	Yes	Open<=2	<=
	<=	K<=A	Yes	2<=Open	<=
Equal To	==	A==B	Yes	Close==Open	=
	==	A==K	Yes	Close==2	=
	==	K==A	Yes	2==Close	=
Not Equal To	!=	A!=B	Yes	Close!=Open	<>
	!=	A!=K	Yes	Close!=2	<>
	!=	K!=A	Yes	2!=Close	<>
Compound	Operator	Syntax	Ultraboost	Examples	Equivalent MSFL
Addition	+=	A+=K	Yes	Result+=2	N/A
	+=	A+=B	Yes	Result+=Close	N/A
Subtraction	-=	A-=K	Yes	Result-=2	N/A
	-=	A-=B	Yes	Result-=Close	N/A
Division	/=	A/=K	Yes	Result/=2	N/A
	/=	A/=B	Yes	Result/=Close	N/A
Multiplication	*=	A*=K	Yes	Result*=2	N/A
	*=	A*=B	Yes	Result*=Close	N/A
Other	Operator	Syntax	Ultraboost	Examples	Equivalent MSFL
Subscripting	[ ]	A[i]	-	Result[5]	N/A

Array Member Functions

Array Member Functions		
Function	Description	Example
Array::size()	Returns the number of elements in the array	A.size()
int Array::GetFirstIndex(void)	Gets the first valid bar in the array.	A.GetFirstIndex()
int Array::GetLastIndex(void)	Gets the last valid bar in the array.	A.GetLastIndex()
void Array::SetFirstIndex(const int N)	Sets the first valid bar N bars from the start of the array.	A.SetFirstIndex(4)
void Array::SetLastIndex(const int N)	Sets the last valid bar N bars from the start of the array.	A.SetLastIndex(230)

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Array Friend Functions

Array Friend Functions			
Function/Description	Ultra boost	Example	Equivalent MSFL
Array ref(const Array A,const int N)			
References a previous or subsequent element in a DATA ARRAY(A). A positive PERIOD references "N" periods in the future; a negative PERIOD references "N" periods ago.	Yes	ref(Close,-1)	ref(DATA ARRAY, PERIODS)
Array IF(const Array C,const Array A1,const Array A2)			
A conditional function that returns the second parameter (THEN) if the conditional expression defined by the first parameter, C, is true; otherwise, the third parameter is returned (ELSE).	Yes	IF(Close>Open,Close,Open)	if(DATA ARRAY > >= < <= <> = DATA ARRAY, THEN DATA ARRAY, ELSE DATA ARRAY)
Array sum(const Array A,const int N)			
Calculates a cumulative sum of the DATA ARRAY(A) for the specified number of lookback PERIODs(N) (including today).	Yes	Sum(close,5)	sum(DATA ARRAY, PERIODS)
Array valuewhen(int N,const Array C,const Array A)			
Returns the value of the DATA ARRAY, A, when the EXPRESSION, C, was true on the Nth most recent occurrence. This includes all data loaded in the chart.	No	valuewhen(2,Close==Open,Close)	valuewhen (Nth, EXPRESSION, DATA ARRAY)
Array lowest(const Array A)			
Calculates the lowest value in the DATA ARRAY since the first day loaded in the chart.	No	lowest(Close)	lowest(DATA ARRAY)
Array lowestbars(const Array A)			
Calculates the number of periods that have passed since the DATA ARRAY's lowest value. This includes all data loaded in the chart.	No	lowestbars(Close)	lowestbars(DATA ARRAY)
Array llv(const Array A,const int N)			
Calculates the lowest value in the DATA ARRAY over the preceding PERIODS (PERIODS includes the current day).	No	llv(Close,14)	llv(DATA ARRAY, PERIODS)
Array highest(const Array A)			
Calculates the highest value in the DATA ARRAY since the first day loaded in the chart.	No	highest(Close)	highest(DATA ARRAY)
Array highestbars(const Array A)			
Calculates the number of periods that have passed since the DATA ARRAY's highest value. This includes all data loaded in the chart.	No	highestbars(Close)	highestbars(DATA ARRAY)
Array hhv(const Array _A,const int N)			
Calculates the highest value in the DATA ARRAY over the preceding PERIODS (PERIODS includes the current day).	No	hhv(Close,14)	hhv(DATA ARRAY, PERIODS)
Array cum(const Array _A);			
Calculates a cumulative sum of the DATA ARRAY from the first period in the chart.	Yes	cum(Close)	cum(DATA ARRAY)

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Appendix D

Installing the Free Borland C++ Compiler on your system

To use an external C++ compiler to parse the Vitamin-C script file and report back the errors to the IDE requires that you have installed the available Borland C++ compiler Version 5.5 compiler, which is freely available from:

http://www.codegear.com/downloads/free/cppbuilder

EMBARCADERO TECHNOLOGIES

Search | Location | English | Log On

Home Products Solutions Support Developer Network Education Downloads How to Buy About Us

Downloads
Trial downloads, free product downloads and registered users downloads

Downloads (keys where required)

You can purchase the product editions listed on this page, and the other C++Builder editions that are not listed on our shop site.

Download Help

Name	Platform	Version	Release Date	Size	Notes
C++Builder 2010 Architect Trial	Windows	2010	08/25/2009	Varies	Free trial edition of C++Builder 2010 full IDE including our latest C++ compiler with early support for C++0x standards, editor, debugger, visual development tools, new database modeling and design capabilities, Unicode support and much more. C++Builder 2010 product information
Embarcadero RAD Studio 2010 Trial	Windows	2010	08/25/2009	Varies	Free 30 day trial includes C++Builder 2010, Delphi 2010 and Delphi Prism 2010 for .NET. RAD Studio product information.
Borland C++ Compiler	Windows	5.5	08/24/2000	8.7 Mb	Free Borland C++ Compiler download. Please see the file bcb5tool.hlp in the Help directory for complete instructions on using the C++Builder Compiler and Command Line Tools. Some items referred to in the Command-line Tools help (bcb5tools.hlp) are not included in the free C++Builder Compiler package. More Information: What is Included; Supplementary Information; Using C++Builder Compiler

Downloads are no longer available for C++Builder products earlier than version 2007, Turbo C++ or Borland C++.

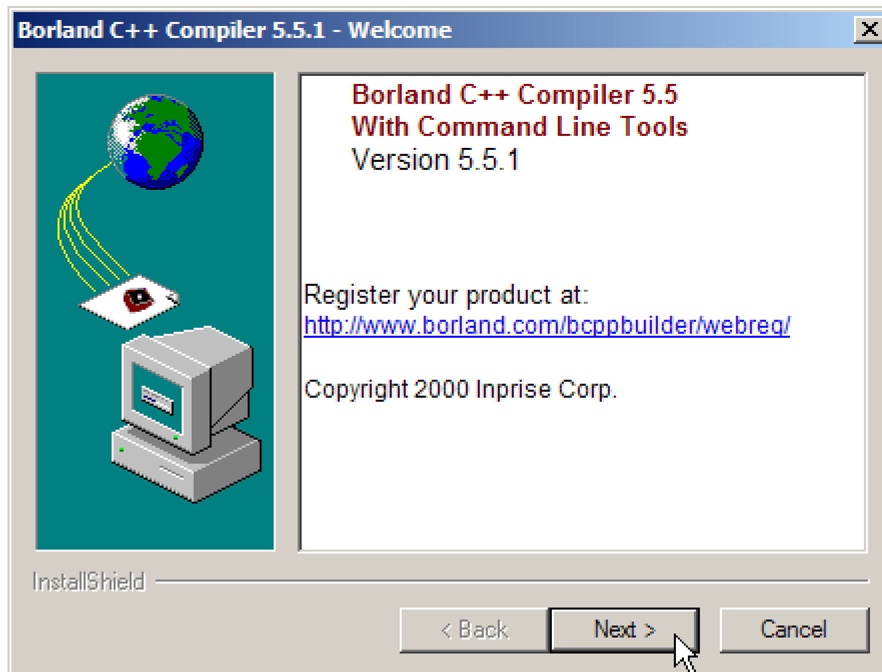
CONTACT US SITE MAP LEGAL NOTICES PRIVACY POLICY REPORT SOFTWARE PIRACY Copyright© 1994 - 2009 Embarcadero Technologies, Inc. All rights reserved.

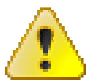
If you already have C++ Builder 6 or below installed on your system then you can skip this step. You can easily check if you have C++ Builder and the compiler is installed on your system by running a command prompt and typing bcc32 at the command prompt. You should see something similar to the following:

VitaminC

```
C:\>bcc32
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
Syntax is: BCC32 [ options ] file[s]     * = default; -x- = turn switch x off
  -3    * 80386 Instructions        -4      80486 Instructions
  -5      Pentium Instructions      -6      Pentium Pro Instructions
  -Ax     Disable extensions        -B      Compile via assembly
  -C      Allow nested comments     -Dxxx   Define macro
  -Exxx   Alternate Assembler name  -Hxxx   Use pre-compiled headers
  -Ixxx   Include files directory   -K      Default char is unsigned
  -Lxxx   Libraries directory       -M      Generate link map
  -N      Check stack overflow      -Ox     Optimizations
  -P      Force C++ compile         -R      Produce browser info
  -RT   * Generate RTTI             -S      Produce assembly output
  -Txxx   Set assembler option      -Uxxx   Undefine macro
  -Vx     Virtual table control     -X      Suppress autodep. output
  -aN     Align on N bytes          -b    * Treat enums as integers
  -c      Compile only              -d      Merge duplicate strings
  -exxx   Executable file name      -fxx    Floating point options
  -gN     Stop after N warnings     -iN     Max. identifier length
  -jN     Stop after N errors       -k    * Standard stack frame
  -lx     Set linker option         -nxxx   Output file directory
  -oxxx   Object file name          -p      Pascal calls
  -tWxxx  Create Windows app        -u    * Underscores on externs
  -v      Source level debugging    -wxxx   Warning control
  -xxxx   Exception handling        -y      Produce line number info
  -zxxx   Set segment names

C:\>
```

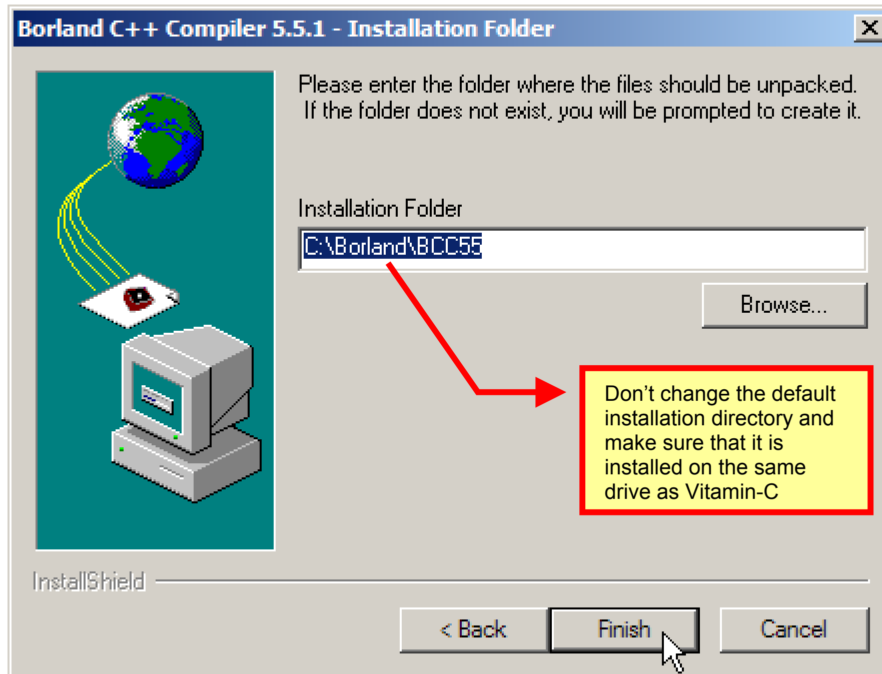
Otherwise if it is not installed on your system download the setup file to your computer and then run it and follow the prompts as shown below:



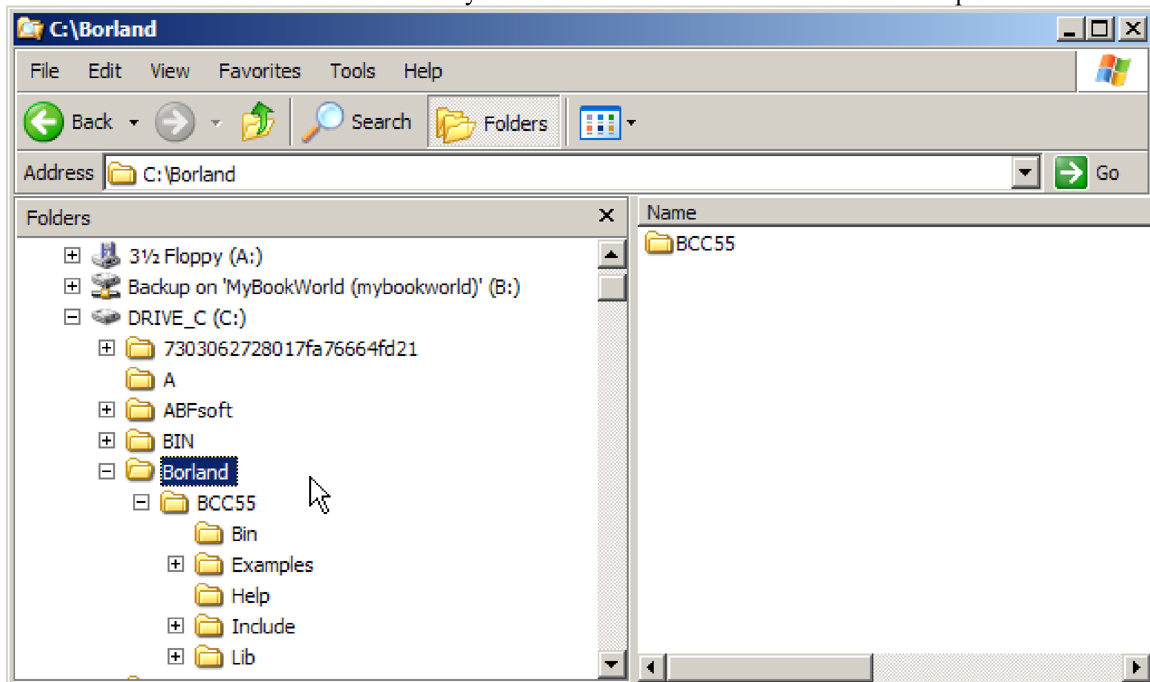


When installing the free Borland compiler make sure it is installed in its default installation directory ie X:\Borland\BCC55 where X is the same logical drive where Vitamin-C is installed on.

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au



If the installation has been successful then you should be able to see the files with file explorer.



Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Appendix E

MetaStock Sample Indicator code

The following MSFL code is used for creating Indicators that call the sample Vitamin-C scripts that are installed as part of the set-up file to the Vitamin-C script directory (c:\VitaminCScript\) upon installation. To use them you will need to create new indicators in MetaStock and copy and paste the code below into each indicator.

Adaptive Moving Average

Vitamin-C - AMA

```
fast := 2/(2+1);
slow := 2/(30+1);
dir:=Abs(CLOSE-Ref(CLOSE,-10));
volatility:=Sum(Abs(CLOSE-Ref(CLOSE,-1)),10);
ER:=dir/volatility;
sc:=Power(ER*(fast-slow)+slow,2);

ExtFml( "VitaminC.CallScript2",
"AMA.c",        { name of script file }
"AMA()",        { function name and parameter }
CLOSE,          { User1 data array }
sc);            { User2 data array }
```

Exponential Moving Average

Vitamin-C - EMA Test

```
ExtFml( "VitaminC.CallScript1",
"EMA.c",        { name of script file }
"EMA(10)",      { function name and parameter }
CLOSE);         { data array }
```

Simple Moving Average

Vitamin-C - SMA Test

```
ExtFml( "VitaminC.CallScript1",
"SMA.c",        { name of script file }
"SMA(10)",      { function name and parameter }
CLOSE);         { data array }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Guppy CBL

Vitamin-C - Guppy CBL Test

```
ExtFml( "VitaminC.CallScript",
"GuppyCBL.c",           { name of script file }
"CBLTrailingStop(3)"); { function name and parameter }
```

2D-Array

Vitamin-C - 2D-Array

```
ExtFml( "VitaminC.CallScript",
"2DArray.c",           { name of script file }
"Test2Darray()");      { function name and parameter }
```

3D-Array

Vitamin-C - 3D-Array

```
ExtFml( "VitaminC.CallScript",
"3DArray.c",           { name of script file }
"Test3Darray()");      { function name and parameter }
```

STL string

Vitamin-C - STL string Test

```
ExtFml( "VitaminC.CallScript",
"STL_string.c",        { name of script file }
"Test()");             { function name and parameter }
```

Sinewave

Vitamin-C - Sinewave

```
ExtFml( "VitaminC.CallScript",
"Sinewave.c",        { name of script file }
"Sinewave()");       { function name and parameter }
```

If()

Vitamin-C - Test If(...)

```
ExtFml( "VitaminC.CallScript",
"IF.c",            { name of script file }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
"TestIF()");        { function name and parameter }
```

Simple Moving Average

Vitamin-C - SMA Test

```
ExtFml( "VitaminC.CallScript1",
"SMA.c",       { name of script file }
"SMA(10)",     { function name and parameter }
CLOSE);        { data array }
```

Trailing Stop

Vitamin-C - Trailing Stop Test

```
ExtFml("VitaminC.CallScript3",
"TrailingStop.c",
"TrailingStop(BAND,LONG)",
3*ATR(10),
CLOSE,
LOW);

ExtFml("VitaminC.CallScript3",
"TrailingStop.c",
"TrailingStop(BAND,SHORT)",
3*ATR(10),
CLOSE,
HIGH);
```

Time Stop

Vitamin-C - Time Stop

```
EntryTrigger:=(DayOfWeek()=1);


EncodedTrigger:=ExtFml( "VitaminC.CallScript1",
"TimeStop.c",         { name of script file }
"TimeStop(30)",       { function name and parameter }
EntryTrigger);


ActualEntryTrigger:=(EncodedTrigger=1) OR (EncodedTrigger=3);
ActualExitTrigger:=(EncodedTrigger=2) OR (EncodedTrigger=3);


ActualEntryTrigger;  { display entry trigger on chart }
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

```
ActualExitTrigger;   { display exit trigger on chart }
```

Profit Stop

Vitamin-C - Profit Stop

```
EntryTrigger:=(DayOfWeek()=1);

EntryPrice:=OPEN;  { entry or reference price }

ExitPrice:=CLOSE;  { exit or threshold price }


EncodedTrigger:=ExtFml( "VitaminC.CallScript3",

"ProfitStop.c",

"ProfitStop(2)",

EntryTrigger,

EntryPrice,

ExitPrice);


EncodedTrigger;
```

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Reference Literature

The following references are ones that provide relevant background material for using TradeSim.

1) Equis - ***Metastock User Manual*** for Windows 95/98 & NT. This is the user manual that comes with Metastock Version 7,8,9 or 10 and is a prerequisite for using Vitamin-C.
2) Equis - ***MetaStock Developers Kit User Manual***

General Reading and References

The following references are ones that the author has read and recommend for general reading. This list is by no means exhaustive.

C/C++ Programming and Language Reference

3) Stephen Prata - ***C++ Primer Plus***
4) Jesse Liberty, Bradley Jones – ***Sams Teach Yourself C++ in 21 days***.
5) Herbert Schildt – ***Teach Yourself C++***
6) Stephen R. Davis – ***C++ for Dummies***
7) John Hubbard – ***Programming with C++***
8) Microsoft Publication – ***C for yourself***

Online C/C++ References

cplusplus.com - http://www.cplusplus.com/

Cited TradeSim Documents

Note: These documents should be available if you have TradeSim installed on your system but if any of these articles are missing they can be downloaded from the following website location http://www.compuvision.com.au/Articles.htm

9) ***TradeSim User Manual***
10) ***AN2 - Implementing Volatility Trailing Stops the Simple Way***
11) ***TB2 - The Universal Text Trade Database File Format***

General References on Trading

12) Alexander Elder – ***Trading for a Living***

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au

Copyright © 2000-2009 by Compuvision Australia Pty Ltd. All Rights Reserved
http://www.compuvision.com.au