*For my **Lord and Savior**, **Jesus Christ**.*
And for my own son, Alex.

RDC &
CAlive

Rapid Development Compiler
-- *and* --
CAlive Programming Language

by ***Rick C. Hodgin***
of ***Liberty Software Foundation***

Public **B**enefit **L**icense -- `Oct.09.2014`
by **Lib**erty **S**oftware **F**oundation

[http://www.libsf.org/**licenses**](http://www.libsf.org/licenses)

First Edition: **`Nov.15.2015`**
Printed in Phoenix, AZ
United States of America
ISBN: **`No ISBN for This Document`**

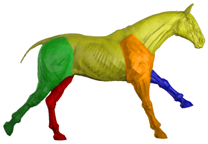
Table of Contents

Preface

This book describes two related yet separate entities: ***RDC*** and ***CAlive***.

RDC is a comprehensive ***compiler framework architecture*** designed to support the creation of multiple programming languages. It introduces and exposes a host of facilities which allow the creation of programming languages through descriptive syntax and grammar rules. It does so graphically, and with an all-points debug ability. This leverages the *RDC* engine as a powerful developer tool.

CAlive, on the other hand, is a ***programming language*** with traditional C language support. In extends beyond C to include the *basic class*, *exception handling*, and other features (*introduced later in this book*).



RDC was created as a ***workhorse compiler engine***. It was designed to ***the*** core compiler for ***all*** of *LibSF's* software languages, complete with ***full edit-and-continue***. *RDC* was constructed in parallel with ***CAlive*** and two other languages, ***VXB*** *(an XBASE language)*, and ***LASM*** *(a multi-ISA assembly language)*. *RDC* was designed to contain a host of facilities comprehensive enough to allow these disparate languages to be designed and completed entirely within the same framework. No additional specialized or external coding was required.

By following the guide used for these languages, developers will be able to port existing programming languages into the *RDC* framework, or create new ones, all being done quick and easy.

Overall design

The *RDC* compiler architecture is comprised of several steps and multiple sub-components. Each sub-component is designed to allow for multi-language support within the common framework:

- ***SourceLight*** – *An external tool for language syntax design*
- ***Front-end*** – *Translates text into high level operations*
- ***Compiler*** – *Translates high level operations into steps*
- ***Generator*** – *Translates steps into assembly code*
- ***Assembler*** – *Translates assembly code into binary code*
- ***Linker*** – *Translates binary code into OS programs*
- ***LiveCode*** – *The ABI used, allows for on-the-fly editing during runtime*

Sourcelight

RDC provides a host of common tools to make adding new languages and compiler extensions easy. The primary tool is called: ***SourceLight***

SourceLight is a two-part system. The first part is (1) a designer which allows command, function, flow, and operator definitions to be created in a GUI, stored in a text-based *.slss* file, which is then (2) compiled into a database which is used by the *RDC* program to parse and process the language.

SourceLight has a built-in compiler which generates a compiler module suitable for use in the *RDC''s* front-end module as a plugin DLL for the language. In addition, it also generates a separate database to be used by *SourceLight*-aware editors. It aids in providing syntax-assist features to developers while editing source code, including token lookups, documentation, and contextual / related links.

Front-end

The *front-end* is the tool which takes raw text file inputs, and the *SourceLight* database, and parses the text files through instructions in the database in stages to generate an internal structure used by the the next step.

The *front-end* supports a wide array of tools which support the full range of compiler abilities available in *RDC*, including the ability to call custom DLL modules at any time. As every programming language is different, the translation steps between raw source code and a form suitable for processing in the *compiler* module mandates that the *RDC* compiler design be extremely flexible, with the necessary fundamental abilities seen in programming languages exposed through a common API.

This flexible relationship in the front-end processing, use of a *SourceLight* database to process the raw text files, is one of *RDC*'s greatest assets. It is what gives the *RDC* framework its flexibility.

Compiler

The *compiler* module receives a pre-defined structure generated by the *front-end*. It begins looking at the actual high level steps encoded at that point, and then translates them into a sequence of fundamental steps to carry out the workload.

> ***Note:*** *Externally, there is no user-visible distinction between the front-end module, and the compiler module, for both of these are included in the same binary. However, internally they are two separate and distinct modules.*

SourceLight allows for some high-level plugin definition to be used at this point as well. For example, various optimizations could be applied at this stage. However, the purpose of the RDC is that it is a ***rapid*** *development compiler*, so more attention is paid to getting the source code changes quickly through the compiler and into the binary (which may be actively running, see *LiveCode*).

> ***Note:*** *Over time more attention will be paid to optimizations. The RDC compiler architecture is robust and extensible enough to support full optimization.*

Generator

The *generator* module is specific to a particular ISA. Support is currently targeted for these ISAs: ***32-bit*** *x86 and ARM,* ***64-bit*** *x86 and ARM, and* ***32-bit*** *and* ***40-bit*** *li386-x40.*

The *generator* module's only job is to take the fundamental steps created in the *compiler* module, and generate the appropriate assembly source code output. This creates a clean break in the compilation process from the previous modules to anything beyond.

> ***Note:*** *While the translation in this module is usually very close to a 1:1 ratio between fundamental steps and assembly source code mnemonics, some optimizations may also take place at this level.*

The *generator* module tags the generated source code with annotations indicating to the assembler which ISA or ISAs are being targeted (as *RDC* can generate output to multiple ISAs in a single compile).

All assembly source code created by the generator module is ***position independent*** and can be migrated to any location in memory with only a few pointers needing updated when moved outside of a *LiveCode* update. However, anything which contains data on the stack should only be updated using *LiveCode*.

> ***Note:*** *The generator module physically writes out* ***annotated assembly source code****, and not binary code. This allows its text-based output to be passed to other tools which can analyze or alter the resulting source code before it goes into the assembler module as a separate step.*
>
> ***Note:*** *This multi-stage source code level was done to allow LiveCode to track program flow from original source code and its high-level operations, down to the corresponding assembly source code, and finally down into raw binary code. This allows debugging to occur at any of those levels, as well as LiveCode fixups to be performed at any of those levels.*

Assembler

The *assembler* module is essentially another complete compiler, but one which accesses some different internal functionality to compute the binary output. It translates assembly source code into a binary object file suitable for later linking.

The generator stage also automatically provides `/|auto|\` markups which carry forward into the output program database, allowing the correlation of every high-level operation in source code to be visualizable down to the assembly level, providing full *horizontal debugging* abilities.

Linker

The *linker* module is also essentially another complete compiler, but one which also accesses some different internal functionality to receive the binary input, process it through to additional binary output.

It allows one or more binary object files generated by the *assembler* module to be used, and creates an OS binary targeted for the specified OS and ABI.

The *linker* module is capable of creating multiple binaries targeted at different operating systems from the same binary object file, as well as all required fixup code specific to the OS.

LiveCode

LiveCode is the name given to the ABI (application binary interface) *RDC* employs. It was created by LibSF and is designed to allow for rapid changes to a binary image running in memory. It provides full support to allow every aspect of the program to be edited on-the-fly while the program is running.

LiveCode uses a BXML file (Binary XML) to hold the various modules which are present (depending on compiler settings):

- *Full original source code, or links to disk-based source code*
- *Generated assembler source code*
- *Machine code*
- Compiler, and Debugger settings and environment
- Meta data

LiveCode introduces a new ability to the compilation process: *the ability to run* ***errant code***.

If source code is malformed, it does not necessarily prevent the *RDC* compiler from ultimately generating a binary which will run. However, there are settings in the *SourceLight* language definition which may do exactly that. But if those stoppers are not enabled, then *RDC* can generate a binary that contains errant code. This will result in the execution of as much of the binary as is possible until the line with errant code is reached. At that point it will trigger a new *LiveCode* feature, called an an ***inquiry***, which allows the binary to be suspended into a debugger where live changes to the code can be made to correct the errant condition.

> ***Note:*** *An **inquiry** is a new running program state where it suspends to a debugger, allowing the source code or runtime environment to be corrected so the program can continue. If no resolution is possible, then the program can be unengaged. In addition, if no debugger is available, a special LiveCode function can be called which will handle the errant condition programmatically.*
>
> ***Note:*** *Everything about RDC (**R**apid **D**evelopment **C**ompiler) is targeted at writing code faster, and being more productive as a developer during the authoring and debugging experience. Less attention is paid to the runtime execution speed as these are typically more than adequate on modern hardware.*

Philosophy

LibSF

The *Liberty Software Foundation* maintains a fundamental philosophy, one which is derived from the relationship each of us shares here on this planet as comparable citizens, placed here by the wisdom and guidance of Jesus Christ.

We remember always that He had a plan for us as a community of fellows, in service to Him (and therefore to each other) when He placed us here. As such, the *Liberty Software Foundation* philosophy is summed up generally in this brief saying:

In God's sight we've come together...
We've come together to help each other...
Let's grow this project up … together!
In service and Love to the Lord, forever!

At all points in creation, Jesus Christ orchestrated a magnificent, harmonious, and inspiring creation of awe and wonder. If we gaze at the Heavens, peer into mathematics, dive into physics, or investigate the variety and diversity of life here on this planet … at all points it pushes our minds to their absolute limits, and yet we are still unable to comprehend the vastness of it all, the complexity of it all. We can only gaze truly at the beauty of it all.

Every sign that we have been given demonstrates, with fullest clarity, that we are part of a community of man here upon this planet, that our lives are integral pieces of the lives of those around us (both near and far). Not one of us is here by accident. And all of us are here with a purpose unto God, and unto one another.

Those of us who author content for the *Liberty Software Foundation* recognize and acknowledge this relationship. We acknowledge that it was God who first gave all of us every unique and special ability and talent we possess, that the opportunities we've had for schooling, learning in general, and everything else associated with the arrival at our vocation and position in this world, that all of it came from Him.

Without God's guiding hand often pulling us along as we go, we would not be where we are today, nor would we be the people we are today. Those of us at the *Liberty Software Foundation* bring this realization ***front and center***, desiring to ***give back unto God*** the ***fullest fruits*** of that which He ***first gave us***: *everything we possess.*

In Application

The Liberty Software Foundation content is all released into the Public Domain with a single caveat: ***that each person receiving it employ self-accountability unto God that the content will remain open***.

This is, after all, the model God gave mankind with regards to the Earth and everything in it. It was His intention that we receive His gift with gladness, and also receive His guidance for using that gift with gladness, and then employ His gift of guidance upon the gift of the Earth and everything in it, enjoying it all to the fullest capacity (as per His correct and desirable guidance and instruction).

However, God is not the only entity at work in our lives. A prideful and arrogant enemy of God arose and approached man, tempting him to seek guidance from a source apart from God. And when this was accomplished, the world we see around us today was manifested.

The ways of this world today are not the ways this world was intended to be for man. It is this way today because people have stopped listening to God, following instead after the enemy of God through his temptations of sin upon our sinful nature. Our bodies exist in a fallen state (they age, get sick), and we are spiritually dead (unable to discern matters of the spirit), yet we are and remain three-part beings made up of distinct components: *soul, spirit, body.* Our soul is the true us. Our spirit is dead, so we have no life within us. Instead, God gives us a spirit which connects our soul to our body, and we manifest in that way here in this world. Once the spirit departs the body, the body is no longer quickened, and it ceases to function. This does not change our soul, however, as our soul truly is eternal and we will continue on after death, either in Heaven with God, or in Hell with the flames of torment.

In this world today, it is only through the redemptive blood of Jesus Christ that any of us can come to faith and be saved, to come to Jesus Christ, ask forgiveness for our sins, and literally have those sins taken away from us bringing us again into a right standing with God so that we are then alive again spiritually, able to discern and comprehend the things of God gain (the “born again” nature).

And for those of us at the Liberty Software Foundation who have come to Jesus Christ and received the rebirth, our attentions are now focused back on that which God would've had us do from the beginning (had we not fallen into temptation, and died because of sin).

In Focus

The Liberty Software Foundation employs a larger philosophy in everything that we do than that which would apply to the mere authoring of some digital content. That philosophy ties back to what is called the ***Village Freedom Project***, and it is part of a philosophy whereby this very type of assertive and affirmative action for God, purposed upon Him with the fullness of our lives, is given by us in all areas of our lives.

To put it simply: *We are His prayerful servants in this world, living our lives for Him, and brining knowledge of Him to the people of this world in everything we do.*

RDC and CAlive

It is the general belief of those at LibSF that computers should aid people in their work. They should provide features and tools which allow them to conduct their tasks more efficiently. There have been a lot of advancements in presentation graphics made since the late 1990s and early 2000s. These have culminated into a host of new, nearly 3D user interfaces. These concepts should be incorporated not just into our screens, but into the way people think, work, and plan their software and other computer-based products.

RDC and CAlive were designed from inception to be part of an integrated environment. That system is expected to be working in direct conjunction with a GUI editor/ IDE, and an always-present, always-on debugger.

The GUI editor provides code facilities which operate in the background in real-time. These bring source code to life using the features of LibSF's *SourceLight*, for example, which is the term given to a set of features which explore that which is being developed in source code, showing references, usages, as well as full documentation for those things being constructed.

It is the ongoing hope of those at LibSF that these GUI features will only expand and increase over time. LibSF seeks to provide a robust framework allowing easy, custom, add-on extensions to be created and shared by/with other developers. Each extension accesses and utilizes the same API that core-*RDC* and core-*CAlive* utilize, giving full insight into all features and abilities known to the native products.

Invitation

We at the LibSF would like to invite you to come participate in writing software for Jesus Christ, and for our fellow man. To use the unique and special talents, abilities, interests, and fields of expertise, to serve the Lord in this way.

Won't you come and join the community of man who reaches deep within themselves and offers up that which the Lord first gave us?

CAlive

The Language

CAlive was designed to create an extended C-like programming language, one which adds some modern programming features including the ***class***, ***exception handling***, and other features not found in C or C++.

It was given the name ***CAlive*** because it the hope of LibSF to bring new life to our old workhorse ... C.

CAlive is pronounced ***"sea alive,"*** and is abbreviated to the letters ***CA*** (as in *C*, *C++*, *C#*, and *CA*), and the source files for *CAlive* use `.ca` for their file extension.

From the ***C*** *perspective*:	*CAlive* brings some notably absent features into the core language, jump-starting it with a simple implementation of well-proven features that are in wide use today in many systems.
From the **C++** *perspective:*	*CAlive* distills some of the best features down to a few features, jettisoning the excessive C++ baggage in favor of source code simplicity, better eyesight, increased sanity, and ease of code maintenance and understanding.
Bottom line:	*CAlive adds to C what should've been added by C99.*

C and C++ developers

If you already know C or C++ and their shared general syntax, then you already know 98% of everything you need to know to program in *CAlive* today. In fact, you don't need to use any of *CAlive's* new features at all. By using the ***-c90*** or ***-c99*** command line switches, most of your C code (and simple C++ code) will compile today without changes.

There are a few features of C, and many in C++, which are not supported in *CAlive*. And some of *CAlive's* new features provide a much welcomed relaxation of stricter syntax in C designed to make source code easier to read, document, and maintain. Still some new features relate to some new concepts to C/C++ developers (or applied implementations of existing concepts).

Engage, Unengage, and Binary

LibSF uses the terms ***engage*** and ***unengage*** to describe starting and stopping something. A program, proces, or thread is *engaged*, and later *unengaged*, as part of its normal life cycle.

Compiled software goes through cycles from (1) ***source code***, (2) ***intermediate code***, (3) ***assembly code***, (4) ***object code***, which ultimately produce a (5) ***binary*** in its machine code form.

Member Access with `.` or `->`

CAlive has made the decision to merge the *dot* `.` and *pointer* `->` access to members. This means that use of either `parent.member` or `parent->member` will work in all cases, on all variables, and with equal weight and assessment. There is no difference in syntax if an object was declared locally, or is accessed through a pointer. *CAlive* auto-resolves this for the developer by context, removing another painful hurdle so often seen in C-like development.

LiveCode

CAlive uses a new ABI (Application Binary Interface) called ***LiveCode***. *LiveCode* is a full *edit-and-continue* (or *fix-and-continue*) ABI. It allows changes to be made to all code and data at both development time and runtime.

Because of this ability, error conditions do not typically generate errors. Instead, they signal something called an *inquiry*. *Inquirys* are triggered to indicate something unexpected was encountered. They suspend the program, and wait for a developer fix the condition so the program can continue.

LiveCode provides a full trace of every source code to machine code, including additional tags provided for by *side coding* `\|code|/` casks (explained later).

> ***Note:*** *There may be multiple levels of source code conversion, such as expanded macros, and pre- or post-processing by third party tools.*

> ***Note:*** *All such levels of processing are included as well.*

LiveCode's step-by-step tracing allows a type of diff to be created at various levels, determining what exactly needs to be changed in the running *binary.* And GUI editor refactoring features aid in that change by assigning new names to existing tokens, or by refactoring members from one location to another.

Program Inquirys

As previously mentioned, *CAlive* introduces a runtime suspension state called an ***inquiry***.

Inquirys are entered into as the result of an unhandled exception, or b use of the new ***inquiry keyword***. *Inquirys* exist in lieu of exceptions and errors, and allow developer intervention to correct the errant state, and resume the running process, without losing anything active in memory.

Inquirys can also be introduced directly by the compiler or linker when some pieces required to complete the *binary* perfectly are not present. Rather than failing to create a *binary* in such a case, *RDC and CAlive* simply wrap the missing components in *inquiry* signals, which allow the missing bits to be fixed when encountered.

Once an *inquiry* is entered into, the program is suspended and developer is able to examine the running state and determine what needs to be corrected to allow the program to continue. Or, the developer may choose to *unengage* the program if the condition is non-recoverable.

`livecode_precompile()` and `livecode_update()`

CAlive introduces the ability to control compilation by use of functions that are compiled and evaluated at compile-time. These are: **`livecode_precompile()`**, and **`livecode_update()`**.

> ***Note:*** ***livecode_precompile()*** *is called only on a full recompile. CAlive calls the* ***livecode_update()*** *with data for all livecode updates.*

livecode_precompile()
This function is used to allow controllable pre-compilation edits and a full examination of all source code, #include files, macro expansions made in iteration, and more.

After each main source file has been loaded, all their #include files have been loaded, and macros have been expanded, if it was found, *CAlive* will compile and execute the **`livecode_precompile()`** code before compiling the rest of the source code.

```
// Code to handle custom compilation steps goes here
bool livecode_precompile(CCompile* context)
{
    CCompileLine* cl, clNew;
    CcompileFile* cf;


    // Compiler switches can be set or retrieved
    compile->switches.set(name, value);
    if (compile->switches.get(name).equals(value))
        // Action here


||||||||||
|||| Lists of files can be examined:
|||
|||  for (cf = compile->file.first(); cf; cf = cf->next())
|||  {
|||      // Obtain information on files here.
|||      // Find our their source code blocks
|||      // Load new files
|||      // Delete files completely
|||      // Swap out their code based on compiler environment settings
|||      // Et cetera
|||  }
|||
||||||


||||||||||
|||| Source code lines can be traversed:
|||
|||  for (cl = compile->sourceCode.first(); cl; cl = cl->next())
|||  {
|||      // Source code lines can be inserted:
|||      clNew = cl->insert("// new source code line", /*before?*/false);
|||
|||     // Marked "already compiled"
|||      clNew->flags.set(completed, true);
|||
|||      // Deleted:
|||      cl = cl->delete();
|||
|||     // Altered:
|||     cl->update("// new source code line content");
|||
|||     // And generally examine and process as necessary
|||  }
|||
||||||

}
```

The `livecode_precompile()` function is called on compilation steps during livecode updates.

livecode_inquiry()

The *inquiry* solution in use by *RDC and CAlive* is powerful for live maintenance, but it does require manual intervention by the developer. It is recognized that this type of immediate, hands-on attention may not always be possible. As such, an automated facility to handle *inquirys* is also provided for in the framework.

The automated facility is available to handle conditions when a debugger and/or developer is not immediately available. It's handled through a new function called **`livecode_inquiry()`**, which exists as a function definition at some point in the file. This function can only be called by the OS, and is called whenever an *inquiry* is encountered and a debugger instance is not available.

The **`livecode_inquiry()`** function is given context information about the *inquiry*, where it occurred, what type of *inquiry* it was, etc., This allows a determinable action to be *taken*. It may include a *memory dump*, some type of *user interaction*, a transfer of control to *another function*, or even reporting the *inquiry* as a legitimate error condition, and subsequently *unengaging* the program.

Here's an example of the implementation of the **`livecode_inquiry()`** function:

```
function livecode_inquiry
|params livecode liveCode
{
 switch (liveCode.inquiry.code)
 {
 case _INQUIRY_DOUBLE_INQUIRY:
 // An inquiry was encountered in main_inquiry()
 break;

 case _INQUIRY_UNHANDLED_INQUIRY:
 // An unspecified inquiry was encountered
 break;

 // et cetera...
 }
}
```

any and *allow*

CAlive introduces two new types called ***any*** and ***allow***. These are designed to (1) make the transfer of pointer data easier (without casting), (2) to provide a mechanism to allow unnamed parameters to be passed to a function, and (3) to provide a method of accounting for and accessing the receipt of variable parameters within a function.

any can be used as in lieu of a void* to indicate that the poitner can be automatically translated to any other type of pointer without casting. It can also be used as an unnamed parameter declaration.

allow is only used in function declarations, and in function body *param* listings. It provides a logical array which accesses the types and data passed to the function.

```
// An example of any* used in lieu of void*
struct SAbc
{
 void*  v;
```

```
    any*   a;           // Identical to void*, except it does not
                        // require casting to another type
};

SAbc abc;
char* cv;
char* ca;

// Initialize code would be here

cv = (char*)abc.v;      // Requires cast on void* to another type
ca = abc.a;             // Does not require cast on any* to another type


||||||||||
|||| See jump f(); for another example using any to hold unnamed params
||||||||||


// Declaration and prototype
int my_function(any);


// Body definition
function my_function
| returns int r
| params any
{
    // Code goes here, parameters accessed manually on the stack
}
```

The passed parameters can be accessed by their expected incoming types:

```
// Declaration and prototype
int my_function(any);


// Usage in code:
my_function(1,2,3,4);   // Note that the 4th parameter is NOT accessible
 // directly in my_function()'s body


// Body definition
function my_function
| returns int r
| params int a, int b, int c     // No 4th parameter
{
 // Code goes here, the first three parameters accessed by name
 // The 4th parameter is accessible manually on the stack.
 // Had this function been declared to match the
}
```

Use *allow* in lieu of `...` to indicate there are variadic parameters. It conveys a name through which additional parameters can be accessed at runtime, without requiring a *va_arg* solution.

> ***Note:*** *va_arg is still supported.*

To access each element in an *allow*, use its `name[n]` reference beginning at `n=1`. `n=0` is reserved as an int type holding the *allow* parameter count (as in `int count = name[0]`). In addition, *dot tags* are available which provide *type information* for each parameter. Use of `name[n].type` returns an integer which follows back to a litany of `_LIVECODE_ALLOW_TYPE_*` constants, which resolve to back to their fundamental types (including `any*` for types of *class** or *struct**).

> **Note:** *This ability is designed to be used in functions which are called using allow parameters.*

> **Note:** *The allow parameter can be subsequently passed to additional functions. They will inherit the exact parameters the original received as a* ***direct reference*** *to those values. To make an* ***explicit copy*** *of the allow parameters, use the* ***copyof()*** *macro.*

```
// Use of allow parameters
function my_function
| returns int r
| params allow p
{
    int i;

    // Dump passed parameters if the count is > 0
    if (p[0] > 0)
    {
        // At least one parameter
        printf("Param count: %d\n", p[0]);
        for (i = 1; i < p[0]; i++)
            printf(" -- p[%d] is %d, %s\n", i, p[i].type,
                                       iLiveCode_allow_getTypeName(p[i].type));

        // Call another function with these same parameters:
        other_function(p);    // Receives a direct pointer to the data

        // Call another function with a copy of the p parameters
        other_function(copyof(p));

    } else {
        // No parameters
        printf("No parameters passed.\n");
    }
}
```

> **Note:** *allow parameters can be augmented at any point using the built-in LiveCode functions:*

```
    // Allows deletion of ann allow param
    iLiveCode_allow_delete(name, index);


    // Parameters can be inserted
    iLiveCode_allow_insertAfter(name, index, newParam);
    // Note:  To insert at the start of the list, use index = 0
```

```
// Multiple parameters can be appended in one command
iLiveCode_allow_append(name, ...);
```

> **Note:** *There is a minor performance hit in accessing and using allow parameters, but their typing advantages make it well worth the hit in most cases.*

> **Note:** *To minimize the hit on heavy use, copy the* `name[N]` *parameter to a local variable one time, and then use the local variable from that point forward. If the parameter needs updated, then store it one time at the end of normal processing.*

thisCode and thisCodeOf(x())->

CAlive introduces two concepts which relate to a particular location in source code. They are the *thisCode* identifier, and the *thisCodeOf()* function.

thisCode is a context-relative identifier which can be passed to other functions using the ***thiscode*** type. *thisCode* allows things which are visible to its home location to then be exposed and accessed through the pointer. It conveys explicit source information, including the namespace, function, and line number it came originated at, allowing for the comparison of two *thiscode* variables.

thisCodeOf() is a compile-time feature which allows for accesses up the call stack. They can be nested, but each one must have either the known function name they're accessing, or a *thiscode* variable conveyed within. This ability is made available in *CAlive* to make documentation clearer, and to reduce the need to pass parameters which may or may not be needed.

When used as a traditional pointer on a *thiscode* type, *thisCodeOf()* exhibits a performance hit because it must validate the named token and type on each reference and use.

```
// Usage examples of thisCode and thisCodeOf()
function main
| returns int r
| params int argc, char* argv[]
{
    first();
    slow(thisCode);
    fast(thisCode);
}


// Accesses main()'s argc integer
function first
```

```
{
    printf("%d parameters\n", thisCodeOf(main())->argc);
    second();
}


// Accesses up the call stack, to first(), then to main()
function second
{
    // Traverse call stack if you know it, up to first(), then main()
    printf("%s is the first user param\n",
                      thisCodeOf(first())->thisCodeOf(main())->argv[1]);
}


// Uses a generic livecode reference that must be validated on use
function slow
| params livecode lc
{
    // Must validate lc, lookup the named token, validate its type usage
    printf("%d parameters, % is the first user param\n",
                                                lc->argc, lc->argv[1]);
}


// Uses a known livecode reference, no performance hit
function fast
| params livecode(main()) lc_main
{
    // Validation is not required. It knows exactly what lc_main is
    printf("%d parameters, % is the first user param\n",
                                    lc_main->argc, lc_main->argv[1]);
}
```

Note: *Use of either of these methods requires intimate knowledge of the machine state and/or call hierarchy as it uses information determined at both compile-time or runtime to obtain the data's location on the stack. Unless the parameter definitions match up exactly, the code would not work properly and would access some unknown bit of data, possibly signaling an inquiry.*

Note: *The thisCode context provides a way to reference to the context by which the current instruction pointer is executing. There are two ways to use thisCode -- (1) The slow method, which does validation, runtime token name lookup, and type validation on each use, and (2) the fast method, which knows at compile-time the context the LiveCode variable relates to, allowing direct access and type validation at that time.*

`jump f();`

CAlive introduces the ability to jump to a function, bypassing its *prologue* code. This feature can easily result in an inquiry, and should only be used when the conditions of execution are well known.

In this example, parameters are passed into a kickoff function using an *any*. This hides the parameters, though they still exist on the stack.

```
// Passes parameters received by an any
function main
| returns int r
| params int argc, char* argv[]
{
    // Only calls kickoff(), and then kickoff() does the jump
    kickoff(2, 1, 2);
    kickoff(3, 1, 2, 3);
}


||||||||||
|||| Note:  The extra passed parameters are hidden in the anonymous any.
||||        They're not automatically accessible here in kickoff(), but
||||        they do still exist, and when the jump is executed below, the
||||        functions which receive them will be able to access them by
||||        their own declaration, one which aligns with the stack.
||||||||||
function kickoff
| params int param_count, any
{
    // Note:  All parameters are inherited by kickoff_2_params()
         if (param_count == 2)     jump kickoff_2_params();
    else if (param_count == 3)     jump kickoff_3_params();
    // Note:  No params were passed to kickoff_N_params().
}


// This function exposes two params from kickoff()'s any declaration
function kickoff_2_params
| params int, int a, int b            // Both are accessible
{
    // Code is entered here, receiving all params kickoff() received.
    // The first parameter has no name so it not directly accessible,
    // but remains only as a placeholder to align the stack.
}


// This function exposes three params from kickoff()'s any declaration
function kickoff_3_params
| params int, int a, int b, int c     // All three are accessible
{
    // Code is entered here, receiving all params kickoff() received.
    // The first parameter has no name so it not directly accessible,
    // but remains only as a placeholder to align the stack.
}
```

Code Groupings

CAlive introduces *groupings*. *Groupings* allow for related information to be conveyed in a visible manner. They are divided into two general types, both of which make use of the pipe sign | character:

Header groupings
These are used in normal code blocks, and convey that, which in standard C syntax, would be something like an expression or parameters.

> **User groupings**
> These are used ad hoc as needed throughout the source file, and allow an obvious visual grouping of related content.

Groupings can occur only as the first non-whitespace token *at the start of a source code line*.

Header groupings are introduced to provide a new way to define things like **`functions`**, **`adhocs`**, **`flowofs`**, **`do while { }`** blocks, **`for { }`** blocks, and *member functions* in classes and structs.

> ***Note:*** *Header groupings use **one pipe sign character**, and follow immediately after the definition line before its opening **{** character.*

Example:

```
// Single pipe-sign for header groupings
function sample
|returns int r                  // Header grouping (leading pipe sign)
|params int a, int b            // Header grouping (leading pipe sign)
{
    // Code goes here
}
```

User groupings, on the other hand, are inserted anywhere needed by the developer. They are interpreted by the compiler as either ***whitespaces*** or ***line comments***, depending on their token length:

> ***Note:*** `|||` *(3) character sequences are interpreted as **whitespaces**.*
>
> ***Note:*** `||||` *(4+) character sequences are interpreted as **line comments**.*

User groupings was added not only to give a visible block cue which catches the eye quickly and easily, but also to introduce a new debugger feature called ***group single step***. *This* allows source code to be stepped over at the *user grouping* level, and to collapse source code where in use, thereby leaving only the associated line comments visible in the source code, making it easier to see the big picture.

Expanded example:

```
||||||||||
|||| Print five
|||
|||  int i;
|||  i = 5;
|||  printf("%d\n", i);
|||
||||||
```

In this example, lines `01`, `02`, and `08` are interpreted as ***line comments***. Lines `03` thru `07` are all interpreted as ***whitespaces***.

Collapsed examples:

```
// Collapse display       -- All line comments in the block are shown
01:      ||||||||||
02: [+]  |||| Print five
08:      ||||||

// Full collapse display  -- Only non-blank line comments are shown
02: [+]  |||| Print five
```

In each case, the `[+]` indicates the code is collapsed, and can be expanded by clicking on it.

N| Line Renumbering

CAlive introduces the ability to reorder lines for visual purposes. To do this, another type of grouping is employed, one where the pipe sign is prefixed with only the local line number order.

```
// Normal code:
function name
|returns s32 r
|params int a, int b
{
    a = get_value();
    r = some_function(a, b);
    b = set_value(b);
}

// Reordered code.
function name
|returns s32 r
|params int a, int b
{
   // It's compiled & engaged in the order 1,2,3, but shown in the 1,3,2
   // order in the GUI editor:
   1|  a = get_value();
   3|  b = set_value(b);
   2|  r = some_function(a, b);
}
```

Why would anyone do this? Good question. :-)

There have been times in programming where I've wanted to see the differences between two source code lines, but because there was a line or two of source code between them it made it difficult to do a visual compare because the content was interrupted by the intervening line (or lines). This resulted in a type of visual noise which made it difficult to accomplish the task.

My solution at that time was to reorder the lines temporarily, perform the comparison, and then put them back, as by normal source code editing. However, each time I came back I had to repeat the steps. I concluded that if there were a way to leave them that way, yet signify visually that they're out of order, it would make it easier to see the minor differences.

By re-ordering the lines, the eye automatically groups things which are similar, making it easy to see those subtle differences without affecting compilation order or program flow. The leading `N|` characters indicate visually that they are out of order in the GUI editor.

> ***Note:*** *The GUI editor has built-in features which also re-order them into proper order visually when the feature is enabled.*
>
> ***Note:*** *Groupings can be of any length, but most be contiguous. This would include any comment lines that are within. They must all be* `N|` *numbered.*

Double `{{` and `}}`

CAlive introduces `{{` and `}}` characters, referred to as ***double left brace*** and ***double right brace***.

Their purpose is directly related to their forerunners `{` and `}`, and serve to mark off block which may include improperly mated standard block characters, `{` and `}`.

They can be used anywhere, but are typically used only in macro definitions and flowof blocks *(explained later)*.

Multi-line `#define` Macros

CAlive introduces a simple way to define multi-line macros through the use of the double left-brace {{ and double right-brace }} combinations:

```
// Traditional macro definition
#define multiLineMacro \
        line1 \
        line2 \
        line3


// CAlive allows this syntax:
#define multiLineMacro {{
        line1
        line2
        line3
}}
```

It is slightly longer, but the content between `{{` and `}}` is easier to read.

> **Note:** *By using the **double left-brace** `{{` and **double right-brace** `}}`, the ability exists in the multi-line macro definition to have unmatched `{` and `}` combinations.*
>
> **Note:** *The **{** and **}** characters can be escaped within the **{{...}}** macro body using `\{` and `\{` as needed. Embedded **{{** and **}}** characters can also be escaped using `\{\{` and `\}\}` as needed.*

Fundamental Data Types

CAlive supports standard C and C++ fundamental data types, including:

```
bool                                -- 8-bit boolean

int8_t,   char                      -- 8-bit signed
int16_t,  short                     -- 16-bit signed
int32_t,  int                       -- 32-bit signed
int64_t,  long                      -- 64-bit signed
long long                           -- 64-bit signed

uint8_t,  uchar,   unsigned char    -- 8-bit unsigned
uint16_t, ushort,  unsigned short   -- 16-bit unsigned
uint32_t, uint,    unsigned int     -- 32-bit unsigned
uint64_t, ulong,   nsigned long     -- 64-bit unsigned
unsigned long long                  -- 64-bit unsigned

float, real4                        -- 32-bit floating point IEEE-754
double, real8                       -- 64-bit floating point IEEE-754
long double, real10                 -- 80-bit floating point IEEE-754
```

Overrides

CAlive supports standard variable-level overrides including ***volatile*** (also called ***live*** in *CAlive), **const***, ***extern***, ***static***, and also introduces three new ones:

```
ro      -- store in read-only memory
rw      -- store in read-write memory (default), and
rwp     -- read-write protocol, store in read-write memory
```

The **`rwp`** *(read-write protocol)* is an override protecting against data changes unless the **`(|rwp|)`** cask is included immediately before the write. If the **`(|rwp|)`** cask is not included, write accesses result in a compilation error and if encountered in a runtime, an *inquiry*.

> ***Note:*** *Read accesses on **rwp** variables are **always** allowed.*

Example:

```
ro  int _COUNT = 5;    // Read-only memory, behaves like const
rw  int _LOOPS = 5;    // Read-write memory, standard access, default
rwp int _ITERS = 0;    // Read-write protected, only¹ written to when
                       // immediately prefixed with a (|rwp|) cask.

// ¹Note:  All compile-time initializations are honored, but changes
//         made to protected variables must follow protocol.
```

Endianness

CAlive data is stored in memory as ***little endian*** *by default*, but can be overridden for ***big endian***.

To explicitly set the endianness using the command-line, the switches ***-be*** for big endian and ***-le*** for little endian are provided. In addition, every variable can also have its own explicit override which prefixes the variable. Use a prefix of **`be`** for big endian, or **`le`** for little endian will override command line switches.

Example:

```
be int ab;      // ab will be stored in memory as big endian
le int al;      // al will be stored in memory as little endian

// Note:  All references are automatically converted as needed by the CPU
//        for computational use, and re-converted as needed for storage.
```

New Types

CAlive introduces several new fundamental types using a general pattern of signed, unsigned, and floating point designators. These are the single digits **`s`**, **`u`**, and **`f`**. In addition, there are new fundamental structure types which are added to make certain types of low-level software development less library dependent, and more capable out of the box. Note that you can still use any libraries you choose.

```
s8, u8         -- 8-bit signed and unsigned
s16,u16        -- 16-bit signed and unsigned
s32,u32        -- 32-bit signed and unsigned
s64,u64        -- 64-bit signed and unsigned
f32,f64,f80    -- 32-bit, 64-bit, and 80-bit floating point
```

```
mint          -- Machine size integer (32-bit or 64-bit)
muint         -- Machine size unsigned integer (32-bit or 64-bit)
```

Auto-sizing

CAlive introduces the ability to have non-standard variables, which are ***auto-sizing*** on use. These auto-size to 32-bits (prefixed with `e`) or 64-bits (prefixed with `r`) for use in computations. All auto-sizing data types will sign saturate upon storage back to their native form, and can also be used in conjunction with endianness overrides:

> ***Note:*** *These* ***e*** *and* ***r*** *prefixes are taken from the x86 prefixes used for* ***32-bit*** `e--` *and 64-bit* `r--` *registers.*

```
// Expand to 32-bit prefixed with "e"
es8, eu8      -- 8-bit signed and unsigned
es16,eu16     -- 16-bit signed and unsigned
es24,eu24     -- 24-bit signed and unsigned

// Expand to 64-bit prefixed with "r"
rs8, ru8      -- 8-bit signed and unsigned
rs16,ru16     -- 16-bit signed and unsigned
rs24,ru24     -- 24-bit signed and unsigned
rs32,ru32     -- 32-bit signed and unsigned
rs40,ru40     -- 40-bit signed and unsigned
rs48,ru48     -- 48-bit signed and unsigned
rs56,ru56     -- 56-bit signed and unsigned
```

Built-in Extensions

CAlive introduces several built-in extensions which can aid in native data processing without external library support.

> ***Note:*** *These are* ***not part*** *of a "CAlive Standard Library," but are native language features built-in to the compiler and language. As such, these can also be incorporated directly into "standard library" code.*

Built-in Struct and Class Types

```
// The following are available in CAlive as fundamental types, but
// internally they are publicly exposed struct or class definitions
var            -- A weakly typed variable (determined by use and access)
bi             -- A big integer (arbitrary precision)
bfp            -- A big floating point (arbitrary precision)
builder        -- An accumulator allocated in chunks as needed
datum          -- An allocated data block with length, and populated length
date           -- A date with year, month, day
datetime       -- A date plus hour, minute, second, millisecond
process        -- A process control variable
thread         -- A thread control variable

// Note:  Once Visual FreePro, Jr. is completed, four new types will also
//        be introduced:  database, index, record, and field.  These will
//        allow full access to a dBASE-like database engine.
```

Keywords

```
function        -- A function definition
adhoc           -- An ad hoc function definition
flowof          -- A flowof definition
inquiry         -- Signal an explicit inquiry

flow            -- A flow { } block for flow control
flowof          -- A flowof { } block
flowin          -- Restarts a flow { } block at the top
flowout         -- Leaves a flow { } block, can also use break or exit
inquiry         -- An inquiry { } block

purpose         -- An externally visible local function
port            -- An externally visible port of entry into a function

with            -- A with (x) { } block to simplify source code

marker          -- [|marker|name|], a target for [|unwindto|]
unwindto        -- Unwinds back up the stack to a named [|marker|]

label           -- A local label
goto            -- Moves to the named label

if              -- A standard if { } block with early-out
else            -- A standard else { } block for if, lif, and fif
lif             -- A line-if with early-out
fif             -- A full-if which always tests every condition
do              -- A standard loop, either [do] while { }, or do { } while
for             -- A standard for { } loop
break, exit     -- Exits a block
continue, loop  -- Moves to next iteration in a standard flow control block

self            -- Refers to current function, adhoc, or member function
this            -- Refers to current object
thisCode        -- References CAlive-specific objects

only            -- For purposes and ports, to override returns or params
pause           -- Pauses a thread
resume          -- Resumes a thread
engage          -- Launches a new remote process
unengage        -- Closes and shuts down a thread or remote process

meta            -- A meta relationship to a true result of a logic test
mefa            -- A meta relationship to a false result of a logic test
meia            -- A meta relationship to an inquiry
mema            -- A meta relationship to a return message
```

Compile-time Functions and Constants

CAlive offers compile-time functions to aid in documentation, conditional compilation, and sizing.

```
// Variable inquiry
sizeof                   -- size in bytes
countof                  -- count of units
offsetof                 -- offset within its parent
addressof                -- address in memory

// Constant values
true, yes, on, up        -- assigns -1 to their target (all bits set to 1)
false, no, off, down     -- assigns  0 to their target (all bits set to 0)
null, NULL               -- constant value of 0, 0.0f, or 0.0, by context

// Logic operators:
and                      -- verbose form of the && operator
or                       -- verbose form of the || operator

// Bitwise operators:
bcmp                     -- verbose form of the bitwise ~ operator
bnot, bflip              -- verbose form of the bitwise ! operator
bxor                     -- verbose form of the bitwise ^ operator
band                     -- verbose form of the bitwise & operator
bor                      -- verbose form of the bitwise | operator
bshl                     -- verbose form of the bitwise << operator
bshr                     -- verbose form of the bitwise >> operator
```

Casks

CAlive introduces the ***cask***. *Casks* are drop-ins that can be inserted anywhere in code in an otherwise syntactically correct expression.

The basic *cask* is made up of a matching left-side and right-side form, with the cask name in the middle. They come in six types: *five **user types**, and one **system type***

`~| utility |~`	for arbitrary code (*multi-line* support is given via `~|{ … }|~`).
`<| logic |>`	conditional logic test response
`(| reference |)`	references something which already exists
`[| definition |]`	defines something, or alters an existing/prior definition
`| code |/`	source code line delineation for *Side Coding*
`/| auto |\`	auto-inserted by the compiler *(not used manually)*

Graphically the casks appear to be a solid unit in a GUI editor, and always collapsed unless explicitly opened. In text form they take on the form shown above.

Sides and Slots

Casks naturally have a ***left-side***, ***middle***, and a ***right-side*** in their format. In the `(|reference|)` cask used above, these would equate to **(1)** left-side: `(|`, **(2)** middle: reference, and **(3)** right-side: `|)`. However, that is only part of their physical construction.

Casks can also hold data in two extra *slots* called the ***left-slot***, and ***right-slot***. This is in addition to the implicit ***middle-slot***.

To introduce data into each slot location, use two `||` near the symbol on its side, as in:

```
(||left | middle |)                        Note:  (||..|...|)
       (| middle | right||)                Note:  (|...|..||)
(||left | middle | right||)                Note:  (||..|..||)
```

This results in the various forms appearing like this (shown with the ***left-*** and ***right-slots*** highlighted):

```
~|| left |  utility    | right ||~         Note:  ~||..|..||~
<|| left |    logic    | right ||>         Note:  <||..|..||>
(|| left | reference   | right ||)         Note:  (||..|..||)
[|| left | definition  | right ||]         Note:  [||..|..||]
```

> ***Note:*** `\|code|/` *casks do not have any meaning assigned to their left- or right-slots, however, if they did, the general format would be the same:*
>
> `\|| left | code | right ||/`

Specialized Roles

All casks were not created equal. Each of them has a specialized role in *CAlive*, and each of them are integral and necessary within the scope of those roles.

`~|utility|~`

`~| utility |~` casks were created to allow an arbitrary injection of code at any location in the program. As such, it was recognized that they may need include multiple lines of code, causing their definition to exceed a single encapsulated `~| statement |~`.

Once populated, the `~| utility |~` cask can be reduced to a short form with a developer-given name, which represents a visible placeholder for the more complex underlying expression. This frees up screen real-estate in the GUI editor, aids the eyes in arranging things, and eases the mind in gaining an understanding of the algorithm (rather than being bogged down by the unwieldy mechanics of the algorithm).

To create a **~| utility |~** cask with multiple lines, use a **{ }** block:

> ***Note:*** *Either* ***{..}*** *or* ***{{….}}*** *can be used and perform the identical operation, they just need to be mated.*

```
// Code can be bunched together
~|{ printf("Entered\n"); second_thing(); third_thing(); /*Et cetera*/ }|~

// Or code can be spaced out visually as any other block
~|{{

    // Based on the where we are, do one of two things:
    if (some_test)
    {
        // Do something here
    } else {
        // Something else here
    }

}}|~
```

<|logic|>

<|logic|> casks were created to allow engagement on the result of conditional test. There are two default **<|logic|>** casks provided:

<| meta |> -- *Called when the result of a logic test is true*
<| mefa |> -- *Called when the result of a logic test is false*

Custom logic tests treat the **<|logic|>** cask like the trinary operator ((condition) ? if_true : if_false). For the **<|logic|>** cask use this syntax:

<|| condition | if_true | if_false ||>

Each of the if_true or if_false slots can contain standard code to call or embedded casks.

(|Reference|)

Reference casks are used to access something that already exists, and *adhoc* or *function* for example.

To ***receive*** return values, insert them in the ***left-slot***.
To ***pass*** parameters, insert them in the ***right-slot***.

For example:

`(| myprint | "Hello, world!" ||)` passes the "Hellow, world!" parameter to myprint(). It's essentially the same syntax as: **`myprint("Hello, world!");`**

`(|| result | compute |)` receives a return parameter from compute(). It's essentially the same syntax as: **`result = compute();`**

`(|| result | compute | a,b,c ||)` sends parameters and receives a return parameter from compute(). It's essentially the same as: **`result = compute(a,b,c);`**

```
function main
| returns int r
| params int argc, char* argv[]
{
    int x = 0;

    adhoc myprint
    | params s8* format, int value
    {
        printf(format, value);
    }


    // Stand-alone example:
    (|myprint|"%d\n", 5||)
    // Displays: 5


    // Used before and after an assignment:
    (|myprint|"Before: %d\n", x||)  x = 9  (|myprint|"After: %d\n", x||);
    // Displays:  Before: 0
    //            After: 9
}
```

[|Definition|]

`[|definition|]` casks were created to alter the way things are defined. They appear after the name of the thing they are modifying, and they convey the change. **`[|definition|]`** cask modifications come in three forms:

Augment -- *alters attributes of a named token and/or its type*
Definition -- *alters the definition of a class, angel, or struct*
Usage -- *alters a single instance use of a definition*

Augment modifications apply attributes which override the token name's existing attributes. These typically are used to alter things like its *const* or *read-only* nature, whether or not it's *volatile* in this context, etc. These can be coded in the traditional way of course, but when they are modified using **`[|definition|]`** casks, they will appear visibly augmented in the GUI editor in their standard form, with the actual definitions not normally appearing unless a *reveal casks* feature is enabled.

```
// Broken out separately:
int value [|const|] [|volatile|];

// Or combined:
int value [|const volatile|];

     // In the GUI editor, a definition augment alters its appearance:
        [|int value|];    // Is shown blocked off and colorized
```

Definition modifications are used to alter the name of the token, or its type. Changing a token's name requires only that it be unique, but changing its type requires that it be *the same or lesser size*. They use one of two syntaxes:

`[| rename name to new_name |]` -- *renames a member*

`[| retype name to new_type |]` -- *retypes a member*

> ***Note:*** *When alternate definitions are applied, they occupy the same space in the structure as the unaltered definition, so if the altered definition were cast back to the unaltered form, they would be accessible by their original names. However, in cases where a double was re-typed to a float, for example, the value stored in the double token's slot would contain invalid information because the two types aren't compatible.*

```
struct SGeneral
{
    void*  p;
    int    num;
};

struct SCensus
{
    SGeneral* populace [|rename num to count|]
                       [|rename p to people|]
                       [|retype p to SPeople*|];
};

function my_function
{
    SCensus* cen;

    // Members which ARE available:
    //      cen->count;
    //      cen->people;

    // Members which ARE NOT available:
    //      cen->p;
    //      cen->num;
}
```

> **Note:** *In this case, the original member name is **always** used to signify which member is being altered, which is why `p` is used after the first rename.*

Usage modifications do not affect the actual definition of the thing being used, but only the one use in a particular case. This can come in handy when a member is defined as **`void* p`**, for example, and in context it is known that the **`void*`** member won't be **`void*`**, but rather is of some other known form.

```
struct SGeneral
{
    void*  p;
    int    num;
};

function my_function
{
    SGeneral* gen [|rename num to count|]
                  [|rename p to people|]
                  [|retype p to SPeople*|];
}
```

\|Code|/

CAlive introduced `\|code|/` casks for the purpose of *side coding*, a text file augment for enhancing source code explained later in this book.

/|Auto|\

CAlive will auto-insert **`/|auto|\`** casks as needed to provide hints from the compiler about things which may be questionable, or if there was some ambiguity to convey the choice the compiler made.

> **Note:** *This type of immediate feedback from the compiler allows code to edited in-the-moment to fix an incorrect assumption made by the compiler based on need.*

Drop-ins

Casks can be dropped into code at any point. Based on their form they will have different effects on the binary that is produced, or they may have no affect at all.

```
// Standard function using its syntax
function main
| returns int r
| params int argc, char* argv[]
```

```
{
    printf("%d\n", argc);
}


// To apply a cask, insert it anywhere in code.  Note that in the
// examples below, the extra spacing is added to make it easier to
// see.  However, in real source code the spacing is not required.


// Example 1:
function main
| returns int r
| params int argc, char* argv[]
{
    printf((|cask|) "%d\n", argc);
}
// In this case, the (|cask|) is inserted before the "%d\n" parameter


// Example 2:
function main
| returns int r
| params int argc, char* argv[]
{
    printf("%d\n", argc) (|cask|);
}
// In this case, the (|cask|) is inserted after the entire function


// Example 3:
function main
| returns int r
| params int argc, char* argv[]
{
    printf("%d\n", argc (|cask|));
}
// In this case, the (|cask|) is inserted after the argc parameter
```

In each of these examples, the insertion of the cask did not alter the syntax of the expression. The cask is simply viewed as a drop-in inserted at that point in the instruction decoding.

Based on where it's added, the results can be different. If, for example, casks are inserted before and after an assignment expression, the value that will be visible

[|initialize|] and [|sinitialize|]

CAlive will introduce the ability to initialize memory variables to values other than the default 0 by use of a **[|initialize|value|]** cask for numeric initialization, and **[|sinitialize|"init string"|]** for string initialization.

```
// Initialize as though:
//      memset_u8(&x, 0xff, sizeof(x));
int i [|initialize|0xff|];
int i [|initialize|u8 0xff|];
int i [|initialize|u8 -1|];
```

```
// Initialize as though:
//      memset_u16(&x, 1234, sizeof(x));
int i [|initialize|u16 1234||];

// Initialize as though:
//      memset_u32(&x, 1234, sizeof(x));
int i [|initialize|u32 1234||];

// Initialize as though:
//      char c[] = "UninitializedUninitializedUninitializedU"
char c[40] [|initialize|"Uninitialized"||];

// Initialize as though:
//      char c[] = "UninitializedUninitializedUninitialized\0"
char c[40] [|sinitialize|"Uninitialized"||];
```

#Pragmas

CAlive allows for various **#pragmas** to aid in compilation.

```
// Definitions and augments
#align N                -- aligns code/data at indicated byte size boundary
#define X(...)          -- defines a macro
#undefine X             -- un-defines a macro, also #undef
#typedef                -- aliases one type to another name

// For source-code defined debugger assistance
#hide                   -- indicates the following members should not be
                           shown in the debugger
#show                   -- indicates the following members should be shown
                           in the debugger
#order N                -- assigns a new order to the following member

#optimize N [enforce]-- provides a hint to the compiler on what
                           optimization level to use from this point forward.
                           If enforce is used, then it overrides any command
                           line switches.

// Conditional compilation
#if X                   -- a conditional compilation block test
#else                   -- optional conditional compilation block
#elseif, #elif          -- optional conditional compilation block test
#endif                  -- termination for a conditional compilation block

// Compilation context blocks
#push [X]               -- pushes X (if specified) to the stack, or all
                           #pragma settings
#pop                    -- pops the most recent #push back off the #pragma
                           stack

// Source-code controllable compilation
#text                   -- Displays text at compile time
```

```
#warning [N] [text]   -- issues a warning with optional number and text
#error [N] [text]     -- issues an error with optional number and text
#stop [text]          -- stops compilation with some optional text
```

Structs

CAlive supports structures, and adds several features to give more direct developer-time control over how the compiler processes structures, and how those members are displayed in a debugger.

The general forms are:

> ***Note:*** *CAlive's naming convention is to prefix struct definitions with an* **`S`**.

```
struct SNormalStruct
{
    // Members go here
};

struct SBitStruct
{
    // Note:  be and le overrides can be used here, or in variables,
    //        classes, or structs, which use this struct.
    s32   value1    : 5;    // 5-bit variable
                    : 0;    // Skip forward to next byte
    s32   value2    : 3;    // 3-bit variable
                    : 5;    // Ignore these five bits
};
```

Because *CAlive* supports the class, the struct can be thought of as a type of class with all its members public. In this way, structs can also have member functions, including constructors and destructors.

The keyword **`hide`** and **`#order`** pragma can be used to alter how member show up in standard debugger displays. Use of *hide* prevents a member from appearing by default. **`#order`** is typically used to bump an item up to the top of member display lists while debugging that part of the program:

```
struct SExample
{
    s32          id;
#order 1                    // Bump "data" up to show as first member
    datum        data;

    hide void*   p;         // Hides "p" by default, now shown in debugger
};
```

> ***Note:*** *CAlive* ***does not require*** *that the prefix "struct" be used anywhere except*

> *definitions. Once defined, the struct becomes its own type which can be used as needed.*

Renaming Members

CAlive allows for structure members to be renamed on instance use. While this does not change the underlying data storage size, the name used to reference the member can be altered. This allows instances of classes or structs to have members renamed when their forms take on specific meanings in contexts.

```
struct SOriginal
{
    void*  p;
    void*  p2;
};


// Instance use renaming:
SOriginal myStruct rename p to data;

// Right now, myStruct.p      -- would result in an error
//            myStruct.data   -- would is the member's name


// More than one rename:
SOriginal myStruct
    rename p  to data,
    rename p2 to data2;
```

Classes

CAlive supports simple classes, including ***constructors***, ***destructors***, ***operator overrides***, ***default parameters***, ***public*** and ***private*** members, ***multiple inheritance***, ***virtual methods***, ***function overloading***, and a keyword designator for `accessor` functions (which both documents functionality, and simplifies syntax usage in expressions concatenated expressions).

Basic class form (also supports bulk of C++ syntax):

> ***Note:*** *CAlive's naming convention is to to prefix class definitions with a* `C`.
>
> ***Note:*** *CAlive* ***does not support*** *protected members. The keyword protected can still be used, but it will be* ***interpreted as*** `public`.
>
> ***Note:*** *CAlive* ***does not support*** `const` *member functions, but only* `const` *data members and member function parameters. CAlive classes can still have their members defined as const for backward compatibility. However, the statement*

is ignored.

```
class CName
|extends inherit1, inherit2
|extends inherit3
|rename memberName to newMemberName
|rename inherit2.memberName to newMemberName
{
public:

    ||||||||||
    |||| Class functions
    |||
    |||  function CName
    |||  |init memberName1 to value
    |||  |init memberName2 to value
    |||  {
    |||      // Init code goes here
    |||  }
    |||
    |||  function ~CName;
    |||
    ||||||


operator =
|returns CName& r          // "r" used for [r]eturns, can be anything
|params const CName& p     // "p" used for [p]aram, can be anything
{
    // Code goes here
}


||||||||||
|||| Accessors are used for shortened syntax
|||
|||  accessor s32 GetValue()     { return(m_value); }
|||  accessor SetValue(s32 v)    { m_value = v; }
|||
|||  //////////
|||  //
|||  // Allows syntax usages:
|||  //
|||  //    s32   x;
|||  //    CName cn;
|||  //
|||  //    x = cn.GetValue;     // rather than cn.GetValue();
|||  //
|||  //
|||  // If multiple accessors existed through pointers:
|||  //    x = cn.level1.level2.level3.GetValue;
|||  //
|||  //
|||  // Instead of repeated () requirements:
|||  //    x = cn.level1().level2().level3().getValue();
|||  //
|||  //////
|||
||||||


private:
    s32 iSetValue2(s32 iv)
    | returns s32 r
    {
        r = m_value;
        m_value = iv * 2;
```

```
    }

private:
    s32 m_value;
};
```

Angels

CAlive supports a special type of class called an `angel`. The `angel` is identical to a class in every way, except that they are explicitly designed for one-shot usage. They are automatically constructed, used, and deconstructed by *CAlive* without the mechanics of those operations being seen / required by the developer.

Angels also **anonymous reuse** *as needed.*

Two forms of an `angel` can be created:

> ***angel classes***
> *Where a class-like object is defined using the keyword* `angel` *in place of* `class`. *This allows the class definition to automatically be used as an angel when encountered. The only exception is when instances are explicitly created using the* `new` *or* `lnew` *keywords, in which case they will be created as classes, following all class rules for construction and deconstruction.*
>
> ***angel overrides***
> *Angel overrides allow a regular class to also be used as an angel by prefixing a reference with the keyword* `angel`.

```
// Explicit angel definition:
angel ANote
{
public:
    ANote(datum d) : m_d(d) { }
    ~Anote()                { datumfree(m_d); }

    accessor datum d()      { return(m_d); }

private:
    datum m_d = NULL;
};


function name
{
    ANote n;

    printf("%\d, %\d\n", n("Hi!").d,
                         n("John").d);
}
```

In this case, the **`angel ANote`** with the name **`n`** was used twice in one expression. The accessor **`d`** was used to access the datum for the **`printf()`** statement's needs.

If it had been a regular class definition instead of an angel class definition, the syntax changes only slightly:

```
// Explicit angel definition:
class CNote
{
public:
    CNote(datum d) : m_d(d) { }
    ~Cnote()                { datumfree(m_d); }

    accessor datum d()      { return(m_d); }

private:
    datum m_d = NULL;
};


function name
{
    angel CNote n;

    printf("%\d, %\d\n", n("Hi!").d,
                         n("John").d);
}
```

In this case, prefixing the **`CNote n;`** definition with the keyword **`angel`** allows that regular class instance to be used as an angel class through the token name: **`n`**

Multiple angels can be created as needed, each with different names. Names should be used as a mechanism to document usage appropriately, such as using the same **`angel`** capable of storing a message in the contexts of: ***note***, ***warning***, and ***error***. In these cases, use in individual statements conveys more information than simply using something like the name: **`n`**

Structs Are Public Classes

CAlive allows everything which exists in **`class`** or **`angel`** definitions to also also exist in *struct* definitions. The only difference is that in *struct* definitions, everything is always ***public***.

> ***Note:*** *Struct members cannot be overridden to include be private members. They are **always and only public**.*

Conveyable Member References

CAlive introduces the concept of ***conveyable member references***.

Conveyable member references are used when you know the member reference within a *class*, *struct*, or *angel*, but you don't yet have the physical object or pointer to which the member will apply, so you are unable to obtain the direct reference or address.

The definition syntax is similar to that of pointers, but uses the `?` *question mark* in lieu of the `*` *asterisk*.

For example:

```
struct SData;

struct SContainer
{
    SData*  d1;
    SData*  d2;
    SData*  d3;
    SData*  d4;
    SData*  d5;
    SData*  d6;
    SData*  d7;
    SData*  d8;
};


// Create a conveyable member reference
SContainer? t;


// Assign it the d5 member
t = d5;


// Create a new structure instance
SContainer* c = new Scontainer;
if (c)
{
    // Populate by member reference
    c->?t  = NULL;    // With auto type checking
    c->??t = NULL;    // Bypass auto-type checking for speedup
}
```

> ***Note:*** *Type checking can also be globally enabled or disabled using the command line switches **-member-ref-safe**, and **-member-ref-fast**.*

Volatile, Static, Extern, Successive, and Literal

CAlive gives the developer some explicit control over optimizations. These include the traditional ***volatile*** override, which can also be called ***live*** in *CAlive*, and ***static*** and ***extern***, along with two new ones: ***successive*** and ***literal*** overrides.

`volatile` or **`live`** override specifies that the data can change at any time, and should be re-read from memory continually during use.

`static` forces the variable to become a singleton.

`extern` tells the compiler the variable will be defined or initialized in an external module that will be linked in later. ***Note:*** *Full definitions with initializers can be declared extern while still be used locally for determining all references.*

`successive` override specifies that anything within the { .. } block needs to be in successive order as indicated. This causes all data within to be byte aligned, with zero padding provided.

`literal` override specifies that anything within the { .. } block needs to be taken as it is, and cannot be altered in any form for optimizations.

> ***Note:*** *The CAlive compiler allows for command line switches **-volatile**, **-successive**, and **-literal** to be used.*

```
struct SExample
{
 volatile  s32     type;
 datum   data;

 literal           // Prevent altering any type to any other type
 {
 successive    // Prevent padding
 {
 u16    fileId;
 u8     blockCount;
 }
 }
};
```

[|initialize|] and [|sinitialize|]

CAlive introduces a new ***initialize*** ability to override the default initialization value of 0 for unspecified initializers. The initialize content can be given in unsigned integers, or a string of characters to repeat for **`sizeof(x)`**. Use ***sinitialize*** for string to initialize to **`sizeof(x) - 1`**.

> ***Note:*** *The **`[|initialize|]`** you see is called a **cask**, and is described later in this book. This kind of cask is used to aid in defining something.*

```
// Initialized 8-bits:   memset(&x, (u8)0xff, sizeof(x));
int x [|initialize|0xff||];
int x [|initialize|u8 0xff||];
int x [|initialize|u8 -1||];

// Initialized 16-bits:   memset(&x, (u16)1234, sizeof(x));
int x [|initialize|u16 1234||];

// Initialized 32-bits:   memset(&x, (u32)1234, sizeof(x));
int x [|initialize|u32 1234||];

// Initialized:   char c[] = "UninitializedUninitializedUninitializedU"
char c[40] [|initialize|"Uninitialized"||];

// Initialized:   char c[] = "UninitializedUninitializedUninitialized\0"
char c[40] [|sinitialize|"Uninitialized"||];
```

Markers and Unwindtos

CAlive introduces a special definition cask called a **[| marker |]**.

[| marker |] casks are used to create an anchor-point in source code which allow a multi-called function descent in normal program flow to politely unwind and return to the named location, which is higher up on the call stack.

To return a a previously defined **[| marker |]**, use the **[| unwindto | name ||]** cask:

```
function main
| params int argc, char* argv[]
{
    int i = 0;

    [|marker|top||]

    // Exit after 40 iterations
    if (++i > 40)
        exit(0);

    f1();
}


function f1
{
    f2();
}

function f2
{
    f3();
}

function f3
```

```
{
    [|unwindto|top||]
}
```

Auto-Resolves

CAlive introduces a feature to aid in structure and class member access, called *auto-resolve*. Its syntax is two dots either ***within***, or ***trailing***, a parent object.

> ***Within***
> *The compiler will traverse child members to find the matching name at the outer-most edge of all children.*
>
> ***Trailing***
> *The compiler will attempt to auto-resolve the type based on usage. If it finds a single matching member of that type, it will use it. If multiple types are found, then it must report an ambiguity error and request that an explicit member name be used.*

Example of a ***within*** usage:

```
struct SLevel2
{
    int   i;
    float f;
};

struct SLevel1
{
    SLevel1 level2;
};

struct SMyStruct
{
    SLevel2 level1;
};

SMyStruct ms;


// Standard access is given as:
ms.level1.level2.i = 5;
ms.level1.level2.f = 5.0f;


// With auto-resolves, access skips the middle links:
ms..i = 5;
ms..f = 5.0f;

printf(“%d, %f\n”, ms..i, ms..f);

// Same as:  printf(“%d, %f\n”, ms.level1.level2.i, ms.level1.level2.f);
```

Example of a ***trailing*** usage:

```
struct SAbc
{
    int   i = 5;
    float f = 5.0f;
};

SAbc abc;         // Auto-initializes to:  i=5, f=5.0f;


// Trailing references auto-populate within the parent
abc.. = 2;        // Will populate into i
abc.. = 1.0f;     // Will populate into f


printf("%d, %f\n", abc.., abc..);

// Same as:  printf("%d, %f\n", abc.i, abc.f);
```

In this case, the only possible direct population values for `2` and `1.0f` are the members `i`, and `f`, respectively. Were here other members which could also be populated naturally using those values, *CAlive* would report an ambiguity error and require the full usage.

In cases where the only ambiguity may be something like a *short* and an *int* reference both being able to receive the value `2`, then casting the value directly to `(short)2` or `(int)2` would override that ambiguity, and allow auto-resolve to work correctly.

Process Control

CAlive introduces direct process control through a new type called `process`. Two new keywords provide direct support: `engage`, and `unengage`, along with message parsing:

```
process externalApp;    // Initialized to null process

engage      externalApp
    as      "myOtherApp.bin"
    handler msgHandler;

unengage externalApp
    with nExitCode;

if (externalApp)
    send externalApp n1, n2, data;


// Handler prototype for incoming messages
void msgHandler(process p, s32 n1, s32 n2, void* data);
```

Thread control

CAlive introduces direct thread control through a new type called **thread**. Threads are launched using the **in** keyword, with multiple simultaneous thread launches using the **andin** keyword.

Example:

```
thread t1, t2;

in t1 {
    // Code to execute in thread 1 goes here
    // This can be a dispatch to a function, or it can be
    // arbitrary code

    // To leave a thread manually, issue the out command.
    if (!data_is_ready)
        out with nExitCode;

    // When control reaches the closing }, it will issue
    // an out automatically, with an exit code of 0.
    // Or, you can explicitly exit with or without an
    // exit code.  If one is not specified, it uses 0.
    out;

} andin t2 {
    // Code to execute in thread 2 goes here

} inquiry name {
    // If a thread failed to start, this code is signaled
}


//////////
// Re-join when threads t1 and t2 complete,
// Auto-timeout in 2000 milliseconds
//////
    join t1, t2 timeout 2000 handler timeoutHandler(...);


//////////
// Control threads manually
//////
    pause t1;                          // Unconditional pause
    resume t1;                         // Unconditional resume
    pause t1 until sSemaphore;         // Pause to sSemaphore change
    unengage t1 with nExitCode;


//////////
// Handler prototype
//////
    void timeoutHandler(thread t, s32 milliseconds, ...);
```

Purposes and Ports

CAlive introduces the concept of the **purpose** and **port**.

These features were added to give additional documentation and encapsulation to *CAlive* developers at the function level.

> ***Purposes***
> Provides an externally visible encapsulation of sub-abilities which are exposed relative to the main function. An example might be a documentation of the function contained directly within the function which handles the normal compute. An example might be a ***function.help()*** purpose.
>
> ***Ports***
> Provides an externally visible port or destination where processing should begin. Once completed, program flow naturally enters a common code base shared by all ports. An example might be the ability to ***add***, ***edit***, or ***delete***, from within the same function. These would use the ***function.add()***, ***function.edit()***, and ***function.delete()*** port names in code, which then translate to *port { }* blocks within the function itself.

Purposes

Purposes were added explicitly to aid in documentation and encapsulation of their very nature. It allows a single function to be created, and to expose purposes externally which are related, but offer different abilities based on context and need.

Purposes could be thought of as externally visible local functions. They are entered, and they exit, as by normal calling protocols. And there are no mechanisms to directly move to another purpose, except the standard calling convention whereby you call **`self.other_purpose();`** for example.

The original intent was for self-documenting code where the function is defined, and then within its own definition help information could be added. This would encourage the developer to update the help section fro any changes made to the function. :-)

Here we see an example showing a retrieval of help text, in one case, but also the ability to format that text for the GUI help in **`SObject`** structures.

We also see use of the keyword **`only`**, which overrides the function's stated **`returns`** or **`params`**, indicating those of the purpose are to be used in their place. Without use of the keyword **`only`**, parameters are added to those defined in the function, allowing for standard information to be defined in the function, and purpose-specific information to be defined in each **`purpose`**.

```
function process
|returns s32 r
|params s32 a
{
```

```
    purpose help_text
    |params datum helpText
    {
        // The function process.help_text() requires two parameters:
        //     a          -- defined in process() above
        //     helpText   -- defined here
    }


    purpose help_gui
    |only params SObject** guiObjects, s32** tnCount
    {
        // The function process.help_gui() requires two parameters:
        //     guiObjects  -- defined here
        //     tnCount     -- defined here
        //
        // Note:  This purpose still shares the returns value because
        //        because it wasn't overridden with an "only" keyword.
    }


    // Native code for process() goes here
}
```

In this example, calls to **`process.help_text()`**, and **`process.help_gui()`** would call the *purpose* code, rather than the native **`process()`** code for the function.

The **`purpose`** definitions can be used for any program need, and essentially introduce a layer of encapsulation to source code, bringing within a single function related functions.

Function Overloading

The **`purpose`** also introduces the ability to use overloading through the **`operator()`**.

```
function report_error
{
    purpose operator()
    | params int number
    {
        // Reports an error based on number
        // Accessed using:  report_error(5);
    }

    purpose operator()
    |params datum general
    {
        // Reports based on the error text information in general
        // Accessed using:  report_error(datum("General failure"));
    }
}
```

Ports

The **port** introduce the ability to directly enter a **function** or **adhoc** at a point other than its top source code line. This concept of the **port** was originally created to be an optimization feature only, however it was later decided to expose the ability natively in source code because it it is reasonable to expect that the information it keys upon will be known to the developer, and by using it some additional documentation abilities are afforded.

```
function process
|returns s32 r
|params s32 a, s32 b
{
    // Each port is exposed external to this function, being visible
    port add {
        // Code for add operation goes here
        // Accessed as:  r = process.add(a, b);
    }

    port edit {
        // Code for edit operation goes here
        // Accessed as:  r = process.edit(a, b);
    }

    noport {
        // Code to enter when no port was specified
        // If noport is not specified, then it begins
        // at the common code block below
        // Accessed as:  r = process(a, b);
    }

    // Common code begins here
}


function main
{
    s32 x;

    x = process.add(1, 2);
    x = process.edit(3, 4);
}
```

Adhocs

CAlive introduces the **adhoc**. *Adhocs operate the same as* functions, and can be declared at any point in source code. They can be referenced by anything they're declared within. They do not affect normal flow control when they appear in the middle of other code, but are created to help document and simplify source code.

> ***Note:*** *Think of them as local functions, designed to provide documentation and use within a specific scope or context.*

```
function sample
| returns int r
```

```
| params int a, int b
{
    |||||||||||
    |||| Initialize
    |||
    |||   r = 0;
    |||
    ||||||


    |||||||||||
    |||| Quick sum to determine range
    |||
    |||    adhoc quicksum
    |||    | returns total
    |||    | params int value1, value2
    |||    {
    |||        total = value1 + value;
    |||    }
    |||
    ||||||


    // Based on the total, process differently
    if (quicksum(a, b) > 50)
    {
        // Some larger values have sales tax
        if (quicksum(a, b) > 90)
            r += (int)((float)quicksum(a, b) * 0.05f);

        // Additional larger value processing here

    } else {
        // Smaller value have no sales tax
        // Smaller value processing here
    }
}
```

Flowofs

CAlive introduces the **`flowof`**. *CAlive* adds the **`flowof`** to help break out and self-document source code into more manageable components.

Consider this logic:

```
function process
| returns int r
| params int a, int b
{
    if (a >= 0)
    {
        if (a <= 5)
        {
            // a is 0..5
            // Code goes here

        } else if (a <= 10) {
            // a is 6..10
            // Code goes here
```

```
        } else {
            // a > 10
            // Code goes here
        }

    } else {
        // a is negative
        // Report error
    }
}
```

It's not particularly difficult to follow, but what if it could be broken out to be easier to see and understand without having to do any logic refactoring, just a little reordering?

Consider this example broken out using the `flowof`:

```
// Linear flowof block (no parameters, no return value)
function process
| returns int r
| params int a, int b
{
    if (a >= 0)
    {
        ..low_risk;      // 0..5 low-risk
        ..medium_risk;   // 6..10 medium risk
        ..high_risk;     // 11+ high risk

    } else {
        // a is negative
        // Report error
    }
}


flowof process::low_risk
{{
    if (a <= 5)
    {
        // a is 0..5
        // Code goes here
}}


flowof process::medium_risk
{{
    } else if (a <= 10) {
        // a is 6..10
        // Code goes here
}}


flowof process::high_risk
{{
    } else {
        // a > 10
        // Code goes here
    }
}}
```

In this example the code is longer vertically, but it gains notable simplicity in understanding the **`process()`** function logic, and the **`flowof`** blocks below are able to move the highly indented code in the original code to a more manageable level.

All parameters, returns, and local variables are inherited. And, new ones can be added within the **`flowof`** as needed, visible only there. To reference the **`flowof`** in source code, precede the function name with an *auto-resolve* to instruct the compiler on how to find it. In the alternative, use the full-form style, as in: **`flowof.low_risk`** or **`process.low_risk`**.

Each **`flowof`** definition can take on one of two forms, and all **`flowof`** blocks can be used more than once in source code:

> ***linear flowof***
> Used for blocks containing incomplete flow blocks, syntax that would normally be incomplete in the form which appears in the *linear flowof* block. The *linear flowof* bocks can only be referenced by name. They cannot receive parameters or return parameters. Their sole purpose is to take code that's not easily separated from a more complex algorithm, and allow separation in a structured manner.
>
> ***function flowof***
> These are encapsulated functions associated only with the *flow { }* block.

A ***linear*** **`flowof`** is code that is inserted linearly, as though you reached in to the original source code with your fingers, pinched and extracted the portion you needed, and inserted it directly into the **`flowof`**. When program flow reaches the **`flowof`** block, it will execute top-down the code that was extracted, and the debugger will enter into the **`flowof`** block showing the program flow.

> ***Note:*** *The previous* **`flowof`** *example demonstrated a* ***linear*** **`flowof`** *block.*

A ***function*** **`flowof`** is one that can receive parameters and send back return values. This makes them useful in that they still inherit the original context from which they're called, but can process variably within based on a parameter rather than a fixed local variable.

```
// flowof function example
function process
| returns int r
| params int a, int b
{
    a = ..compute(a * 5);
    b = ..compute(b * 3);
    r = ..total;
}


flowof process::compute
```

```
|returns int r
|params int x
{
    // Process the input in some way
    r = some_computation(x);
}


flowof process::total
|returns int r
{
    // a and b are inherited
    r = a + b;
}
```

In this example, there are parameters involved with each **`flowof`**, **`compute`** and **`total`**. The **`flowof`** **`ompute`** receives an input parameter, and has a return a value. And the **`flowof total`** does not receive any input parameters, but returns its own return parameter.

> ***Note:*** *The returns parameter in in both the compute and total are called **r**. This local* **`flowof`** *definition overrides the scope of the returns parameter in the* **`function process()`***, and demonstrates how* **`flowof`** *variables are scoped.*

The concept of the **`flowof`** may seem very similar to that of a **`function`** or **`adhoc`**. However, they differ in that these **`flowof`** blocks are defined outside of the function

Flow Blocks

CAlive introduces a new type of flow control block, one which is closely related to the **`flowof`**, from the previous section. It is called **`flow`**. The **`flow`** block is actually a more comprehensive and encapsulated form of the **`flowof`**, allowing for both ***function*** **`flow`** ***control blocks***, and ***linear*** **`flow`** ***control blocks*** to be created within.

The general format of the **`flow`** block is as follows:

> ***Note:*** *Flow block names are **[optional]**, but convey relative context to operations.*

```
flow [name1]
{
    // Local variable declarations are legal

    // Nested flow blocks are legal
    flow [name2]
    {
        // Nested flow { } block code goes here
        flowout;          // Flows out of this flow
        flowout name1;    // Flows out of the parent flow
```

```
    }

    // Normal flow control block goes here
    flowin;   // Restarts the flow { } block from the top
    flowout;  // Exits the flow { } block

    // When flow reaches the end, it naturally flows out

} always before {
    // Code runs before the flow { } block is entered

} always after {
    // Code runs as the flow { } block is exited

} flowof name {
    // A flowof substitution by name


// Supports C-style declarations:
} return_type name(input_parameters) {
    // An encapsulated function specific to this flow


// Supports Calive-style declarations:
} flowof name
  | returns return_type
  | params input_parameters
{
    // An encapsulated function specific to this flow


// Inquiry interception in code is possible
} inquiry [name] {  // Handles any inquirys which arise
    return;         // Returns to the next line after the line which
                    // caused the inquiry.

    retry;          // Return to the line which caused the inquiry
                    // and retry it.

[|unwindto|name||]; // Unwinds to the previous [|marker|name||]

    flowout;        // Flows out of the flow { } block
}
```

The *flow* block names can be overloaded. Nested **`flow { }`** blocks are allowed:

> ***Note:*** *Unmatched* **`{`** *and* **`}`** **`flow { }`** *blocks can be created. In such a case use* **`{{`** *and* **`}}`** *for the start and end.*

```
flow
{
    if_cond;
    else_cond;

} flowof if_cond {{
    if (cond)
    {
        // Code for the if condition

}} flowof else_cond {{
    } else {
        // else code goes here
```

```
    }
}}
```

In this example, the `flowof if_cond` has only the `if (cond) {` portion of the code block. That is unmatched within the `flowof if_cond`. Likewise, the `flowof else_cond` has the `} else { … }` block which is unmatched within the `flowof else_cond`. These are legal through use of the `{{` and `}}` for the block markers.

For a practical example, consider the previous example as it would be applied using `flow` blocks instead of `flowof`:

```
//////////
// flow block example using a standard C-like syntax
//////
    function process
    | returns int r
    | params int a, int b
    {
        flow
        {
            a = compute(a * 5);
            b = compute(b * 3);
            r = total;

        } int compute(int x) {
            // Process the input in some way
            return(some_computation(x));

        } int total() {
            // a and b are inherited
            return(a + b);
        }
    }


//////////
// flow block example using CAlive syntax
//////
    function process
    | returns int r
    | params int a, int b
    {
        flow
        {
            a = compute(a * 5);
            b = compute(b * 3);
            r = total;

        }

        flowof compute
        |returns int r
        |params int x
        {
            // Process the input in some way
            r = some_computation(x);

        }

        flowof total
        |returns int r
```

```
            {
                // a and b are inherited
                r = a + b;
            }
        }
```

In these examples, the **`flow`** block itself encapsulates the logic required to make it work making the leading *auto-resolves* no longer required, while also breaking out the source code into manageable portions.

The **`flowof`** blocks provide a self-documenting framework which would explain the algorithm more clearly in human words, while allowing the GUI editor the ability to re-assemble the source code into a top-down form without **`flowof`** blocks should it need to be inspected directly for logic.

Note also that standard ***linear*** **`flow`** ***blocks*** can also be created to extract source code, documenting the extraction, without changing logic or program flow:

```
function process
| returns int r
| params int a, int b
{
    flow
    {
        if (a >= 0)
        {
            low_risk;      // 0..5 low-risk
            medium_risk;   // 6..10 medium risk
            high_risk;     // 11+ high risk

        } else {
            // a is negative
            // Report error
        }

    } flowof low_risk {{
        if (a <= 5)
        {
            // a is 0..5
            // Code goes here

    }} flowof medium_risk {{
        } else if (a <= 10) {
            // a is 6..10
            // Code goes here

    }} flowof high_risk {{
        } else {
            // a > 10
            // Code goes here
        }
    }}
}
```

Note that in this case the *auto-resolves* are not required. However, because there are blocks of code which contain unmated **`{`** and **`}`** combinations, the use of the double left-brace **`{{`** and double right-brace **`}}`** are required.

And again, using the GUI editor features, these `flowof` blocks can be rendered to be *inline*, making the original source code flow much easier to visualize and edit in traditional programming logic, but for long-term maintenance and understanding, they afford a much simpler way to truly get a feel for what's going on within the algorithm, before approaching the underlying mechanics involved.

Side Coding

CAlive will introduce a new mechanism for augmenting source code beyond basic program instructions. It allows additional pieces of information to be tagged onto each line using the `\|code|/` cask. The word "code" is replaced with a unique identifier for the tag, which will appear as a horizontal pane in the source code window.

Two standard forms of `\|code|/` casks are defined:

`\|c|/` — *Comment*
`\|sc|/` — *Source code*

Additional user-defined types are possible, such as `\|review notes|/`, `\|bug number|/`, or any other named type … whatever is desirable for your needs.

If *CAlive* sees that a source code line begins with a `\|code|/` cask, it will convey all of the \|code|/ cask identifiers through the compilation process, but will only process source code between `\|sc|/` (if any) up to any other `\|code|/` casks which appear on the line. This allows an unlimited number of `\|code|/` casks to be introduced per line, yet without affecting the actual source code. And from withing the GUI editor the individual panes will be available for display and editing, with new panes being able to be added at any time.

Consider this snippet of source code in a ***myprog.ca*** file:

```
\|c|/Load file\|sc|/numread = read_file(fh, length, &ptr);\|review|/ptr
init?
if (numread != length)
{
\|sc|/     printf("Error reading %d\n", length);\|review|/no return?
}

\|c|/Lowercase\|sc|/for (i = 0; i < length; i++)
    lowercase(&ptr[i]);
```

When viewed through the GUI editor, the source code yields three panes or columns, which are identified by their names. The end result of the above source file is a display which looks like this:

```
     Comments                       Source Code                       Review
+------------+---------------------------------------------+------------+
| Load file  |   numread = read_file(fh, length, &ptr);     | ptr init?  |
|            |   if (numread != length)                     |            |
|            |   {                                          |            |
|            |       printf("Error reading %d\n", length);  | no return? |
|            |   }                                          |            |
|            |                                              |            |
| Lowercase  |   for (i = 0; i < length; i++)               |            |
|            |       lowercase(&ptr[i]);                    |            |
+------------+---------------------------------------------+------------+
```