Integral C — A Practical Environment for C Programming

Graham Ross

CASE Division
Tektronix, Inc.

ABSTRACT

Integral C[1] is an industrial grade integrated programming environment for C programming on an engineering workstation. A single interactive tool replaces a syntax checking editor, a compiler, and a source-level debugger. Its multi-window user interface allows program editing and animated source level debugging, tailored to the needs of a C programmer. The compiler accepts standard C code and reacts to editing changes with function-level incremental compilation. Compilation is done without prompting to maintain the client program in a ready-to-run state. Emitted code is instrumented to catch run-time errors and to permit fine grained debugging. Debugging support code is written in C in a 'workspace', which grants it direct access to a local scope while keeping it separate from the client program.

Introduction

Writing in C is hazardous duty. The language is lexically terse, syntactically sparse, and semantically loose. Each of these characteristics makes programming in C more of a puzzle. Traditional C tools carry this sense of simplicity as far as they reasonably can, which gives the language a reputation as a 'high-level assembly language'. Some of this reputation is well-deserved; it is easy to write confusing and unmaintainable C programs. Likewise, it is easy for an experienced C programmer to see clearly 'through the compiler' anticipating the machine code his program will become. This feature of the language often yields better code. Obviously it can be exploited to an unhealthy extreme.

To preserve C's transparency and frugality, the rules say that a compiler and run-time environment can leave many programming errors undetected. In the draft proposed ANSI standard's parlance, "the behavior is undefined." A separate program, *lint*, is used in many C systems to diagnose the errors the compiler didn't bother to look at. Some programmers use *lint* as a weekly chore, others as a last resort, with the hope that their bugs will turn out to have been detectable at compile time.

The trend in Unix-based[2] C tools has been toward improved interaction models for source-level debugging [Ada86], but without significant debugging support from compilers (thus preserving the high-level assembler) and without the direct aid of integrating the compiler with the debugger. Run-time automated aids to debugging have been implemented with preprocessors that transform a program in various ways to provide animation or run-time checks, such as in *Safe C*[3]. Version control [Tic85] and dependency tracking [Fel79] are the province of still other tools.

Integral C diverges abruptly from this trend. It provides a single tool integrating the typical functions of an editor, compiler, and debugger. Dependency tracking is handled implicitly by Integral C. Integrating the compiling and debugging functions into one tool reduces I/O overhead, allows the debugger to make use of the compiler's structures, and ensures a uniform user

---

1. Integral C and ∫c are trademarks of Tektronix, Inc.

2. Unix is a trademark of AT&T Bell Laboratories.

3. Safe C is a trademark of Catalytix Corp.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

©1986 ACM 0-89791-212-8/86/0012/042 75¢

interface.

Integral C places the greatest emphasis on detection and correction of programming errors. Whenever practical, Integral C diagnoses undefined C language constructs as errors. Most checks done by lint are applied continuously under Integral C. In addition, a large number of run-time errors are identified and diagnosed.

Integral C was inspired by Magpie [DeL84], a prototype Pascal environment developed by Tektronix's Computer Research Labs. The bulk of Magpie's user interface ideas have been brought to Integral C with only minor changes. Where Integral C deviates most significantly is in applying these mechanisms to the C language.

Overview of Integral C

Integral C runs under Ultrix[4] on a Vaxstation[4] equipped with a bitmapped display. It is invoked from the Unix shell. A *project browser* and an *execution window* appear when it is invoked. The execution window is a simple terminal emulator in which the client program does its input and output unless other arrangements are made. The project browser displays a tabular list of source code modules. Several other types of windows are created using commands available within the project browser. Each window provides commands through a pop-up menu. The following sections describe the interactions that take place within each window.

Code Browser

Program text is entered in an unstructured fashion through the keyboard in a *code browser.* Integral C 'figures it out.' We prefer free-form entry using a conventional visual editor instead of template editors or other structured mechanisms [Tei81]. Learning a new editor is never a pleasing prospect, and our editor recognizes a command language similar to the standard Unix editor *vi*. (The editor's internal design permits additional command models to be added with little effort.) C's macro processing features make a template-driven approach more of a nuisance than it might be with a lexically simpler language. Our approach allows our environment to handle any legitimate C code.

4. Ultrix and Vaxstation are trademarks of Digital Equipment Corp.

Reports of syntax and static-semantic errors are available with a single mouse click, so the feedback on static errors is quick. Magpie, which also permitted free-form input, diagnosed syntactic errors immediately by recasting the portion of the program fragment to the right of the error in another font. Semantic errors required a slight delay, but they also were highlighted without a specific request from the user. This unsolicited feedback on errors was seen as 'too busy'.

The check command finds errors that belong to a particular module. The build command does a more global job, finding errors in all modules. It also finds and diagnoses type conflicts between any two modules and between any one module and the standard library. Error messages appear next to the corresponding errors in the program text and commands are provided to help rummage through them.

The code browser demonstrates the smooth integration of separate functions into a single tool. Besides its function as a program editor, it is a primary instrument for static and dynamic debugging. Commands in the code browser can: report syntax errors; look up the definition of an identifier; start, abort, and single-step execution; and set breakpoints and workspace traps (v.i.). The statement-trace option animates execution by highlighting statements in the source code.

Header File Browser

Header files (objects for inclusion using C's `#include` directive) are edited in a distinct *header file browser* because of the ambiguous nature of header files that are included more than once. If the header file has been imported from a standard system directory, the browser provides a read-only editor.

Data Browser

One entry in the project browser is named 'Data'. Creating a window for it yields a *data browser*, which gives a hierarchical view of the data being manipulated by the program. Its three columns contain respectively the name, the type, and the value of a data object, each depicted in high-level notation, as they would appear in C source. The hierarchy is broken at its root into two halves. The 'modules' domain shows a symbol hierarchy that parallels the symbol table. Statically allocated data objects appear here. The 'stack' domain shows a list of activation records on the stack. It displays values for automatically

allocated data objects, which may appear recursively.

Generic 'zoom-in' and 'zoom-out' commands probe deeper into nested scopes in the symbol table and also, by expanding arrays and structures and by dereferencing pointers, probe deeper into structured data objects. The zoom commands work recursively (to explore linked structures) and provide a simple but effective elision mechanism. The ability to use the zoom-in command to dereference pointers makes objects obtained from `malloc` accessible.

Figure 1 shows the effect of several zoom-in commands. The stack displays the names of active functions. Automatic variables belonging to the function `paintPicture` are visible. One of them, `displayList`, has been dereferenced and expanded to show still finer detail.

```
Data Browser                                  Trace on
stack:
 main              int(int,char**)
 paintPicture:     int ()
  i                int                  123
  displayList:     struct disp *        0xcfa0
   *:                                   ...
    .link          struct disp *        NULL
    .origin        double [3]           21,31.4,-6.2
    .rotation      double               0.05432198
    .magnification double               12.34567
```

Fig. 1: Using the zoom-in command

With the stack zoomed in, calls and returns in the client program have the effect of adding and deleting activation records from the list displayed in the window.

With an activation record in the stack display selected, the step-here command executes the client program until a statement boundary is crossed in the selected activation. The browse command opens a code browser on the selected function. The where-am-I command locates the statement at which execution has been suspended in the selected frame.

A variable-trace option causes the displayed values of variables to be updated dynamically as they change during program execution.

The data browser provides a complete but simplistic method for viewing data objects. For more complex jobs, a special purpose workspace can be built.

Other Windows

The *workspace browser* is an editor for code that supports debugging. It is fully described in a later section.

The *command window* provides a scrolling terminal emulator for commands that don't fit the static formula of the browsers. These commands include certain mode-switching commands (like 'workspace off', which disables all workspace-event connections without forgetting them) and certain commands that display tabular data (like 'show errors'). It is intended also to serve as a field service tool, allowing a remote technician to inspect internal structures and provide detailed information on failures to maintenance engineers.

Urgent error messages, and ones that fit nowhere else, appear in *notify windows*. Notify windows can be moved, resized, and deleted.

Building a Program

Integral C keeps track of all dependency information. Relationships are maintained between each source file and the header files it includes, and between each module or workspace and the workspaces that are tied to its events.

Three general mechanisms are used to achieve quick turnaround after a program change: incremental program construction, lazy translation, and throw-away code generation. They are dealt with in order. Magpie was the inspiration for lazy translation and throw-away code generation. Integral C's incremental parsing scheme appears to be new.

Incremental Compilation

When a change is made to program source, a module known as the builder is notified of the change and asked to rebuild the client program. The affected source code is first repreprocessed. The preprocessor results are compared with the previous ones for differences. The preprocessing increment is typically a small number of lines—on the order of ten to a hundred. Only the increment containing the change need be repreprocessed. For a change to a header file this is done once for each inclusion.

In many cases, such as a change to a comment, there are no differences after preprocessing and the builder completes without doing any more work. However, if the change affected the preprocessor's symbol table, for instance by adding a definition for a new macro, preprocess-

ing continues through the rest of the file (or files, for a change in a multiply-included header file). Still, in some images the change might have no effect, for instance if a particular file doesn't use the newly defined macro.

For files in which a change in preprocessed results was seen, the 'virtual parser' is invoked to make up-to-date any program fragments that might have been invalidated by the change. The virtual parser is 'virtual' in the sense that it might not need to do any work: it simply ensures that the requested syntactic and semantic analysis has been done or that an error precluding the analysis has been found and diagnosed. The virtual parser's unit of incrementality—a *fragment*—is one 'external definition' (a syntactic entity in C), except that because of the vagaries of the preprocessor, a fragment is constrained to contain only whole lines. A freak program, in which every newline lies within an external definition, contains only one fragment. (In real life, it turns out that a fragment is approximately one function.)

The virtual parser maintains import-export relationships among fragments. The rules that determine whether recompilation is necessary are comparable to those given by Tichy in [Tic86], but applied with finer granularity and clouded by two features of the C language that make its import/export characteristics unusually complicated. First, each of several declarations of the same variable can lay claim to both importing and exporting it:

```
extern int array[];
extern int array[10];
int array[10];
int array[10] = { 1 };
```

The first declaration gives `array` a type. The second gives it a size. The third appears to be a 'defining declaration', actually setting aside space for `array`. However, when we see the fourth declaration, we reinterpret the third as being synonymous with the second. The fourth declaration finally defines the array and initializes its zeroth element to 1.

Second, tentative definitions (such as the third line in the example) complicate the effects of subsequent changes to the program. For instance, notice that even though the fourth declaration is the defining one, deleting the fourth declaration in the example doesn't leave `array` undefined!

It is worth noting that while Tichy's theory is largely applicable, his examples are not. The import/export notions here are those between fragments within a module, not between modules, so ordering is significant and generally limits the set of fragments a change can affect. Furthermore, Integral C treats inclusions of header files textually, permitting, for example, a macro invocation whose meaning changes from one inclusion to another.

An important criterion in the design of Integral C was to give free rein to programming styles that make use of C's remarkable lexical freedom. One reason for this was that we wanted to be able to handle existing C code that had already taken advantage of that freedom. A second motivation was our belief that C programmers appreciate and demand that freedom. Rather than prescribe fragment boundaries and require the user to edit within them (as was done for Pascal by Magpie) the virtual parser computes fragment boundaries as it compiles the code. Each module begins as one fragment and is subdivided when a natural boundary is identified.

While analyzing a fragment, the parser generates code in an intermediate form. The intermediate code is subsequently compiled into assembly code, assembled, and linked. Each of these 'downstream' compiler phases processes one fragment at a time.

Incremental linking minimizes the effect of program changes on an executing image. When a change is made in code that is not active, execution can proceed from where it left off after the change is made. If the code was active, affected frames are removed from the stack and execution continues by redoing the first call to a function that was replaced.

The ability to continue execution after a change is particularly useful in the case of an undefined function. The function can simply be defined, perhaps as a stub, and execution resumed.

Lazy Translation

Lazy translation permits the program to begin execution before it is completely built. A 'load-me stub' is created by the run-time monitor for each exported entry point and linked into the program as though it were the real fragment. When invoked, a load-me stub sends a message to the builder, which responds by completing the downstream compilation phases for the fragment.

Then the load-me stub jumps to the new code and execution resumes.

Fragments exporting names of statically allocated variables cannot be stubbed in this fashion.

Thrown-Away Code Generation

When the user is idle for a few seconds, the builder begins to do throw-away code generation. Modules are selected in least recently changed order in an attempt to reduce wasted compilations. The expectation is that though the code might be thrown away, any work that might make execution happen sooner is welcome on an idle workstation. For workstations that are being used by more than one programmer, throw-away code generation can be disabled via the command window.

Smart Pointers

When a C program computes a pointer value, the computation invariably hints at the boundaries of the region in which the pointer is valid. Integral C takes the hint and prevents accesses outside the implied region. This is done by recording three addresses in every pointer. 'Current' is the actual pointer value and is modified in the usual ways by pointer arithmetic. 'Base' is the lowest valid address for the pointer. 'Bound' is the lowest invalid address above the valid region. Base and bound are initially set by the `&` operator and affected only by certain so-called primary operators, such as selecting a member of a structure. Before a pointer is used in making a memory reference, the values of these three fields are measured against each other and against the size of the referent. If the region mentioned in the reference is not wholly within the pointer's region of capability, a run-time error is diagnosed. A similar mechanism prevents invalid array indexing. The pointer is actually a five-word quantity and contains two more fields defining the referent's static and dynamic identities. These will be discussed along with workspaces.

What's valid and what's not in pointer operations is a matter for debate. We made some choices that a few programmers might take issue with. When an operator computes the effective address of a member of a structure, Integral C's compiler reduces the capability of the resulting pointer value so it has access only to that member. Some programs make assumptions about the packing of adjacent struct members; we decided those assumptions are non-portable. Maybe we were too strict. When an integer is cast to a pointer, we permit the pointer to access one object of its referent-type at the specified address. Perhaps this is a little too lenient.

The standard memory allocator generates smart pointers, protecting its otherwise vulnerable data structures. Similarly, the system call interface ensures that I/O operations and other 'magical' memory accesses will not violate pointer bounds. Certain library routines, notably `printf`, have been enriched with special diagnostics when pointers are violated in particular ways.

It is important to note that only actual memory references are flagged. It is not illegal for a C program simply to compute an invalid pointer value. It is commonplace in such constructions as

```
for (p = a; p < &a[N]; p++)
  /*loop body*/;
if (p == &a[N])
  /*normal termination*/
```

where `N` is the declared array dimension and `&a[N]` is not inside the bounds of `a`. If the loop terminates normally, `p` holds an address that is invalid and calculations, such as the equality test shown, can reasonably involve that address. Integral C doesn't diagnose this code as erroneous since no out-of-bounds memory reference was made.

Workspaces

A *workspace* is a brace-enclosed block of C statements (syntactically a C compound statement), usually valid in one or more contexts within the main program. A workspace appears in a workspace browser, which isolates the text of a workspace in its own window. The window is split into two panes, side by side. The left pane looks like the left column of the data browser. It is used to select a compilation and execution context for a workspace. Toward this end, it offers the data browser's zoom commands. The right pane looks like a code browser and offers most of the code browser's commands.

To help debug workspaces, the right pane of the workspace provides a trap command (like the code browser) and a stepwise do-it command.

Events

Integral C identifies certain junctures during program execution as run-time *events*. Entering a particular statement or modifying a particular variable are typical events. A short sequence of instructions, known as an *event hook*, is emitted into the object code stream at the point corresponding to each event. The behavior of an event hook is modified by the debugger to implement a debugging function requested by a browser (such as statement tracing) or by the user directly (such as a workspace trap).

Events are selected in the code browser by pointing to the related location in the program text. A workspace can be attached to an event using the code browser's trap command, so that whenever the event occurs during the execution of the client program the hook causes the workspace to execute. When the trap command is given, an icon appears in the program source text at the pointed-to place. The workspace is compiled in the context of the event and, when the event occurs, executed in that context as well.

Because of smart pointers and event hooks, an assignment to an interesting variable is reliably noted even when it happens through a pointer. To accomplish this, the pointer remembers the identity of the variable to which it points (something equivalent to a symbol table pointer) and also the dynamic context that defined the variable (its activation record's address). The data browser uses these pieces of information to identify which displayed value was modified. If a workspace is invoked by the hook it gets the dynamic context as a static link and thus has complete access to the context of the modified variable.

The invocation of the workspace when the event occurs is only slightly slower than a conventional function call, so workspaces can be put to work where one would normally use an assertion (typically a macro), a loop precondition or invariant (typically a passive comment or macro), or a data invariant (typically a forgotten comment and a fervent hope).

More sophisticated schemes might use workspaces to do coverage testing or to identify 'hot spots'.

Immediate Execution

When execution is suspended at a breakpoint, a workspace can be executed with the do-it command in any context in which it is valid. The context is selected in the workspace's left pane, as shown in figure 2.

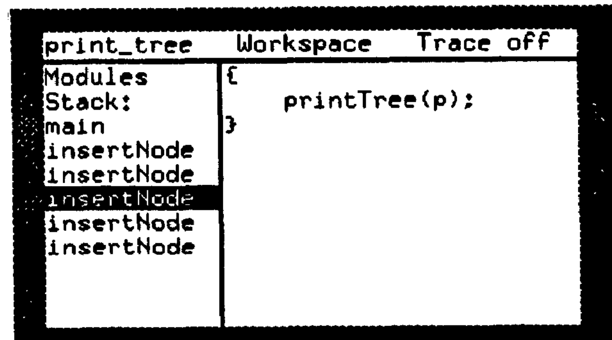

Fig. 2: Immediate execution of a workspace

Selecting an activation record in the left pane of the workspace browser implicitly chooses the active statement in that suspended activation. It also chooses a dynamic context (equal to the address of the stack activation record). The statement determines specific contexts for both the preprocessor and the parser. The do-it command quickly preprocesses and compiles the workspace in the identified static contexts, and links it into the suspended image. The workspace is then immediately executed in the identified dynamic context.

Workspaces are implemented as functions with inaccessible names. They make reference to variables in 'enclosing' scopes through a static link list which is passed as the first parameter to a workspace. For example, `insertNode` in figure 2 is called recursively to insert a node in a binary tree. The pointer `p` is an automatic variable defined in `insertNode`. It points to the root of the subtree visible in an activation of `insertNode`. When a particular activation record has been selected (as shown), the do-it command calls `printTree` to display that activation's subtree.

A do-it workspace can execute a `return` statement, which has the effect of forcing a return from the function in whose context it is running.

A do-it in a static context (selected from the 'modules' domain in the left pane) can be done even when the scope selected is not active on the stack. In this case, explicit `return` and `goto` statements and references to automatic variables are diagnosed by the compiler as static-semantic errors.

Workspaces in Multiple Contexts

Like a header file, a workspace can be used in any number of contexts; it is preprocessed and compiled independently in each one. This is a departure from Magpie's demons, which were usable only in one context at a time. It is a convenience: the same workspace might be useful at several points in the client program. Curiously, the meaning of one use can be different from that of the others. For instance, a workspace containing

```
{
    x = 0;
}
```

can be used in one context where `x` is an `int` and in another where `x` is a `double`. This is a necessary result, but not typically a useful one.

Summary

Integrated environments for procedural languages have been in existence for several years. Cedar [Swi85] and the Cornell Program Synthesizer [Tei81] are good examples. They defined their own programming languages. Environments for existing, popular languages, implemented in a manner faithful to the definition of the language, are rare but not unknown. Magpie is a good example. Integral C is the first environment we have seen that is faithful to the C language. Details of the language dominated its design.

Integral C's approach to incremental compilation is novel. The scheme works well for a highly stream-oriented language like C.

Smart pointers are effective in detecting the most frequent C run-time errors. They also provide a convenient mechanism for implementing some of the other debugging functions of the environment.

Workspaces integrate smoothly into both the environment and the C language. They provide direct access to hidden data without affecting the client program. The multiple-context capability of workspaces makes them substantially easier to use. Instrumenting the code allows a breakpoint to be placed reliably on the modification of a particular variable. Unlike the 'action on breakpoint' of typical source level debuggers, the use of workspaces causes no substantial loss in speed.

Acknowledgements

To Mayer Schwartz and Norm DeLisle for cheerful reviews of a grim rough draft and to the Integral C design team for giving me something to write about.

References

[Ada86]
Adams, E. and Muchnick, S. S. Dbxtool: A window-based symbolic debugger for Sun workstations. *Softw. Pract. Exper. 16,* 7 (July 1986), 653–669.

[DeL84]
DeLisle, N. M. et al. Viewing a programming environment as a single tool. *ACM SIGPLAN Not. 19,* 5 (April 1984), 49–56.

[Fel79]
Feldman, S. I. Make—a program for maintaining computer programs. *Softw. Pract. Exper. 9,* 3 (March 1979), 255–265.

[Swi85]
Swinehart, D. et al. The structure of Cedar. *ACM SIGPLAN Not. 20,* 7 (July 1985), 230–244.

[Tei81]
Teitelbaum, T. and Reps, T. The Cornell Program Synthesizer: a syntax-directed programming environment. *CACM 24,* 9 (September 1981), 563–573.

[Tic86]
Tichy, W. F. Smart recompilation. *ACM TOPLAS 8,* 3 (July 1986), 273–291.

[Tic85]
Tichy, W. F. RCS—a system for version control. *Softw. Pract. Exper. 15,* 7 (July 1985), 637–654.