

CV Users Guide

Version 0.1

Peter A. Buhr, Glen Ditchfield, David Till, C. R. Zarnke ©*1992, 1994, 1998, 2001
Dept. of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

October 2, 2003

*Permission is granted to make copies for personal or educational use

Contents

1 Overview	1
2 Motivation: Why fix C?	1
3 ANSI C versus C\forall	1
4 How To Use C\forall	2
5 Underscores in Constants	2
6 Declarations	2
7 Type Operators	3
8 Routine Definition	4
8.1 Returning Values	4
9 Routine Prototype	5
10 Routine Pointers	5
11 Named and Default Arguments	5
12 Tuples	8
12.1 Tuple Coercions	9
13 Mass Assignment	10
14 Multiple Assignment	10
15 Cascade Assignment	11
16 Record Field Tuples	11
17 Labelled Break/Continue	12
18 Switch Statement	13
19 Case Clause	17
20 Unnamed Structure Fields	17
21 Attribute	18
22 New Arrays	18
23 Exception Handling	19
24 Parametric Polymorphism	19
25 Concurrency	19
26 Example C\forall Program	19
References	19

1 Overview

The current C \forall ¹ project is a combination of two C programming language projects. Both projects extended C in a similar fashion to C++, i.e., building new features and capabilities into the language; in a sense, building a newer language on top of the older one. The work started with K-W C [Til89, BTZ94], which extended C with simpler declaration syntax, multiple return values from routines, and extended assignment capabilities using the notion of tuples. (See [WC96] for some similar work, but for C++.) The original C \forall project [Dit92] extended the C type system with parametric polymorphism and overloading for reuse purposes, as opposed to the C++ approach of object-oriented extensions to the C type system. The work done in these two projects is being merged and augmented to form the current version of C \forall .

2 Motivation: Why fix C?

Even with all its problems, C is a very popular programming language because it allows writing software at virtually any level in a computer system without restriction. For systems programming, where direct access to hardware and dealing with real-time issues is a requirement, C is usually the language of choice. As well, there are millions of lines of C legacy code, forming the base for many software development projects (especially on UNIX systems). Furthermore, for every line of C++ or Java code written, there are probably still 10 lines of C code being written, indicating the continuing popularity of the language. Finally, love it or hate it, C has been an important and influential part of computer science for 30 years and will continue to be for another 10-20 years (albeit, a risky prediction). Unfortunately, C has too many problems and omissions to make it an acceptable programming language for modern needs.

The goal of this project is to engineer modern language features into C in an evolutionary rather than revolutionary way. C++ [Str97] is an example of a similar project; however, it largely extended the language, and did not address existing problems.² Java [GJS96] is an example of the revolutionary approach for modernizing C/C++, resulting in a new language rather than an extension of its descendents. Fortran 95 [For97], Ada 95 [Int95], and Cobol 9X [?] are examples of the evolutionary approach where modern language features are added and problems fixed within the framework of the existing language.

The result of this project is a language that is largely backwards compatible with ANSI'99 C [Int99], but containing many modern language features and fixing some of the well known C problems. Without continued development of the language, C will be unable to cope with the needs of modern programming problems and programmers; as a result, it will fade into disuse. Considering the large body of existing C code and programmers, there is significant impetus to ensure C is transformed into a modern programming language. While ANSI'99 C made a few simple extensions to the language, nothing was added to address existing problems in the language or to augment the language with modern language features. While some may argue that modern language features would make C complex and inefficient, it is clear a language without capabilities like exception handling, concurrency, and modules is insufficient for the complex programming problems existing today.

3 ANSI C versus C \forall

Why not develop C \forall in conjunction with the ANSI C standards committee? The purpose of the ANSI C committee is to legislate rather than innovate. If Bjarne Stroustrup had approached the ANSI C committee with C++, he would not have made progress, and there would be no C++. Without C++, object-oriented programming would not have flourish in the 1980s, and developed into a standard programming paradigm, even though there existed other object-oriented programming languages at that time (e.g., Simula [BDMN80], SmallTalk [GR83]). Similarly, the innovations proposed by C \forall are not yet proven technologies, and hence, too radical for a standards committee to adopt whole-sale. Therefore, it is necessary to develop these ideas first to determine if they are viable before approaching a standards committee.

¹pronounced "C-for-all", and written C \forall , CFA, or Cforall

²Two important existing problems addressed were changing the type of character literals from `int` to `char` and making the type of an enumerator the type of its enumeration.

4 How To Use C \forall

The command `cfa` is used to compile C \forall program(s). This command works just like the GNU `gcc` command for compiling C \forall programs, for example:

```
cfa [C options] yourprogram.c [assembler and loader files]
```

The following additional option is available:

-CFA Only the C preprocessor and the C \forall translator steps are performed and the transformed program is written to standard output, which makes it possible to examine the code generated by the C \forall translator.

5 Underscores in Constants

Numeric constants are extended to allow underscores within the body of the constant, e.g.:

```
2_147_483_648      /* decimal constant */
56_ul              /* decimal unsigned long constant */
0_377              /* octal constant */
0_x_ff_ff;        /* hexadecimal constant */
0x_ef3d_aa5c      /* hexadecimal constant */
3.141_592_654     /* floating point constant */
10_e_+1_00        /* floating point constant */
0x_ff.ff;         /* hexadecimal floating point */
0x_1.ffff_ffff_p_128_l /* hexadecimal floating point long constant */
```

The rules for placement of underscores is as follows:

1. A sequence of underscores is disallowed, e.g., `12__34` is invalid.
2. Underscores may only appear *within* a sequence of digits (regardless of the digit radix). In other words, an underscore cannot start or end a sequence of digits, e.g., `_1`, `1_` and `_1_` are invalid (actually, the 1st and 3rd examples are identifier names).
3. A numeric prefix may end with an underscore; a numeric infix may begin and/or end with an underscore; a numeric suffix may begin with an underscore. For example, the octal `0` or hexadecimal `0x` prefix may end with an underscore `0_377` or `0x_ff`; the exponent infix `E` may start or end with an underscore `1.0_E10`, `1.0E_10` or `1.0_E_10`; the type suffixes `U`, `L`, etc. may start with an underscore `1_U`, `1_ll` or `1.0E10_f`.

It is significantly easier to read and type long constants when they are broken up into smaller groupings (most cultures use comma or period among digits for the same purpose). This extension is backwards compatible, matches with the use of underscore in variable names, and appears in Ada.

6 Declarations

C declaration syntax is notoriously confusing and error prone. For example, many C programmers are confused by a declaration as simple as:

```
int *x[10]
```

Is this a pointer to an array of 10 integers or an array of 10 pointers to integers? Another example of confusion results from the fact that a routine name and its parameters are embedded within the return type, mimicking the way the return value is used at the routine's call site. For example, a routine returning a pointer to an array of integers, is defined and used in the following way:

```
int (*f())[10] { ... };
... (*f())[3] += 1;          /* definition mimics usage */
```

Essentially, the return type is wrapped around the routine name in successive layers (like an onion). While attempting to make the two contexts consistent was a laudable goal, it has not worked out in practice.

C \forall provides its own type, variable and routine declarations, using a slightly different syntax. The new declarations place modifiers to the left of the base type, while C declarations place modifiers to the right of the base type. The only exception is bit field specification, which always appear to the right of the base type. ANSI C and the new C \forall

declarations may appear together in the same program block, but cannot be mixed within a specific declaration. Not supported are K&R C declarations where the base type defaults to `int` if no type is specified³, e.g.:

```
x;                /* int x */
*y;              /* int *y */
f( p1, p2 );     /* int f( int p1, int p2 ); */
f( p1, p2 ) { }  /* int f( int p1, int p2 ) { } */
```

In C \forall declarations, the same tokens are used as in C: the character `*` is used to indicate a pointer, square brackets `[]` are used to represent an array, and parentheses `()` are used to indicate a routine parameter. However, unlike C, C \forall type declaration tokens are specified from left to right and the entire type specification is distributed across *all* variables in the declaration list (as in Pascal). For instance, variables `x` and `y` of type pointer to integer are defined in C \forall as follows:

C \forall	C
<code>* int x, y;</code>	<code>int *x, *y;</code>

Other examples are:

C \forall	C	
<code>[10] int z;</code>	<code>int z[10];</code>	<i>/* array of 10 integers */</i>
<code>[10] * char w;</code>	<code>char *w[10];</code>	<i>/* array of 10 pointers to char */</i>
<code>* [10] double v;</code>	<code>double (*v)[10];</code>	<i>/* pointer to array of 10 doubles */</i>
<code>struct s {</code>	<code>struct s {</code>	
<code>int f0:3;</code>	<code>int f0:3;</code>	<i>/* common bit field syntax */</i>
<code>* int f1;</code>	<code>int *f1;</code>	
<code>[10] * int f2;</code>	<code>int *f2[10]</code>	
<code>};</code>	<code>};</code>	

As stated above, the two styles of declaration may appear together in the same block. Therefore, a programmer has the option of either continuing to use traditional C declarations or take advantage of the new style. Clearly, both styles need to be supported for some time due to existing ANSI style header files, particularly for UNIX systems. In general, mixing declaration styles in a routine or even a translation unit is not recommended, as it makes a program more difficult to read. Therefore, it is suggested that an entire translation unit be written in one declaration style or the other.

All type qualifiers, i.e., **const** and **volatile**, are used in the normal way with the new declarations but appear left to right, e.g.:

C \forall	C	
<code>const * const int x;</code>	<code>int const * const x;</code>	<i>/* const pointer to const integer */</i>
<code>const * [10] const int y;</code>	<code>const int (* const y)[10]</code>	<i>/* const pointer to array of 10 const integers */</i>

All declaration qualifiers, i.e., **extern**, **static**, etc., are used in the normal way with the new declarations but can only appear at the start of a C \forall routine declaration⁴, e.g.:

C \forall	C	
<code>extern [10] int x;</code>	<code>int extern x[10];</code>	<i>/* externally visible array of 10 integers */</i>
<code>static * const int y;</code>	<code>const int static *y;</code>	<i>/* internally visible pointer to constant int */</i>

7 Type Operators

The new declaration syntax can be used in other contexts where types are required, e.g., casts and the pseudo-routine **sizeof**:

C \forall	C
<code>y = (* int)x;</code>	<code>y = (int *)x;</code>
<code>x = sizeof([10] * int);</code>	<code>x = sizeof(int *[10]);</code>

³At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each structure declaration and type name [Int99, § 6.7.2(2)]

⁴The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. [Int99, § 6.11.5(1)]

8 Routine Definition

C \forall also supports a new syntax for routine definition, as well as ANSI C and K&R routine syntax. The point of the new syntax is to allow returning multiple values from a routine [LAB⁺81, Gal96], e.g.:

```
[ int o1, int o2, char o3 ] f(int i1, char i2, char i3) {
    routine body
}
```

where routine *f* has three output (return values) and three input parameters. Existing C syntax cannot be extended with multiple return types because it is impossible to embed a single routine name within multiple return type specifications.

In detail, the brackets, [], enclose the result type, where each return value is named and that name is a local variable of the particular return type⁵. The value of each local return variable is automatically returned at routine termination. Declaration qualifiers can only appear at the start of a routine definition⁴, e.g.:

```
extern [ int x ] g( int y ) {}
```

Lastly, if there are no output parameters or input parameters, the brackets and/or parentheses must still be specified; in both cases the type is assumed to be **void** as opposed to old style C defaults of **int** return type and unknown parameter types, respectively, as in:

```
[ ] g ()          /* no input or output parameters */
[ void ] g (void) /* no input or output parameters */
```

Routine *f* is called as follows:

```
[ i, j, ch ] = f( 3, 'a', ch );
```

The list of return values from *f* and the grouping on the left-hand side of the assignment is called a **tuple** and discussed in Section 12.

C \forall style declarations cannot be used to declare parameters for K&R style routine definitions because of the following ambiguity:

```
int (*f(x))[ 10 ] int x; {}
```

The string “**int (*f(x))[10]**” declares a K&R style routine of type returning a pointer to an array of 10 integers, while the string “[10] **int x**” declares a C \forall style parameter *x* of type array of 10 integers. Since the strings overlap starting with the open bracket, [, there is an ambiguous interpretation for the string. As well, C \forall style declarations cannot be used to declare parameters for ANSI style routine definitions because of the following ambiguity:

```
typedef int foo;
int f( int (* foo) );          /* foo is redefined as a parameter name */
```

The string “**int (* foo)**” declares an ANSI style named parameter of type pointer to an integer (the parenthesis are superfluous), while the same string declares a C \forall style unnamed parameter of type routine returning integer with unnamed parameter of type pointer to *foo*. The redefinition of a type name in a parameter list is the only context in C where the character *** can appear to the left of a type name, and C \forall relies on all type modifier characters appearing to the right of the type name. The inability to use C \forall declarations in these two contexts is probably a blessing because it precludes programmers from arbitrarily switching between declarations forms within a declaration contexts.

ANSI style declarations *can* be used to declare parameters for C \forall style routine definitions, e.g.:

```
[ int ] f( * int, int * );          /* returns an integer, accepts 2 pointers to integers */
[ * int, int * ] f( int );          /* returns 2 pointers to integers, accepts an integer */
```

The reason for allowing both declaration styles in the new context is for backwards compatibility with existing pre-processor macros that generate ANSI style declaration syntax, as in:

```
#define ptoa( n, d ) int (*n)[d]
int f( ptoa(p,10) ) ...             /* expands to int f( int (*p)[ 10 ] ) */
[ int ] f( ptoa(p,10) ) ...         /* expands to [ int ] f( int (*p)[ 10 ] ) */
```

Again, programmers are highly encouraged to use one declaration form or the other, rather than mixing the forms.

8.1 Returning Values

Named return values handle the case where it is necessary to define a local variable whose value is then returned in a **return** statement, as in:

⁵Michael Tiemann, with help from Doug Lea, provided named return values in g++, circa 1989.

```

int f() {
    int x;
    ... x = 0; ... x = y; ...
    return x;
}

```

Because the value in the return variable is automatically returned when a C \forall routine terminates, the **return** statement *does not* contain an expression, as in:

```

[int x] f() {
    ... x = 0; ... x = y; ...
    return; /* implicitly return x */
}

```

When the **return** is encountered, the current value of *x* is returned to the calling routine. As well, “falling off the end” of a routine without a **return** statement is permitted, as in:

```

[int x] f() {
    ... x = 0; ... x = y; ...
} /* implicitly return x */

```

In this case, the current value of *x* is returned to the calling routine just as if a **return** had been encountered.

9 Routine Prototype

The syntax of the new routine prototype declaration follows directly from the new routine definition syntax; as well, parameter names are optional, e.g.:

```

[int x] f ();           /* returning int with no parameters */
[* int] g (int y);     /* returning pointer to int with int parameter */
[ ] h (int,char);     /* returning no result with int and char parameters */
[* int,int] j (int);  /* returning pointer to int and int, with int parameter */

```

This syntax allows a prototype declaration to be created by cutting and pasting source text from the routine definition header (or vice versa). It is possible to declare multiple routine-prototypes in a single declaration, but the entire type specification is distributed across *all* routine names in the declaration list (see Section 6), e.g.:

C \forall	C
[int] f(int), g;	int f(int), g(int);

Declaration qualifiers can only appear at the start of a C \forall routine declaration⁴, e.g.:

```

extern [int] f (int);
static [int] g (int);

```

10 Routine Pointers

The syntax for pointers to C \forall routines specifies the pointer name on the right, e.g.:

```

* [int x] () fp;       /* pointer to routine returning int with no parameters */
* [* int] (int y) gp;  /* pointer to routine returning pointer to int with int parameter */
* [ ] (int,char) hp;   /* pointer to routine returning no result with int and char parameters */
* [* int,int] (int) jp; /* pointer to routine returning pointer to int and int, with int parameter */

```

While parameter names are optional, *a routine name cannot be specified*; for example, the following is incorrect:

```

* [int x] f () fp;     /* routine name "f" is not allowed */

```

11 Named and Default Arguments

C \forall provides two extensions to routine call, with corresponding changes in routine definition to support them. These extensions are: named and default arguments [Har76].⁶

⁶Francez [Fra77] proposed a further extension to the named-parameter passing style, which specifies what type of communication (by value, by reference, by name) the argument is passed to the routine.

Named (or Keyword) Arguments: provide the ability to specify an argument to a routine call using the parameter name rather than the position of the parameter. For example, given the routine:

```
void p( int x, int y, int z ) { ... }
```

a positional call is:

```
p( 4, 7, 3 );
```

whereas a named (keyword) call may be:

```
p( z : 3, x : 4, y : 7 );      // rewrite => p( 4, 7, 3 )
```

Here the order of the arguments is unimportant, and the names of the parameters are used to associate argument values with the corresponding parameters. Implementationally, most compilers simply rewrite a named call into a positional call. The advantages of named parameters are:

- Remembering the names of the parameters may be easier than the order in the routine definition.
- Parameter names provide documentation at the call site (assuming the names are descriptive).
- Changes can be made to the order or number of parameters without affecting the call (although the call must still be recompiled).

Default Arguments provides the ability to associate a default value with a parameter so it can be optionally specified in the argument list. For example, given the routine:

```
void p( int x = 1, int y = 2, int z = 3 ) { ... }
```

the allowable positional calls are:

```
p();           // rewrite => p( 1, 2, 3 )
p( 4 );       // rewrite => p( 4, 2, 3 )
p( 4, 4 );    // rewrite => p( 4, 4, 3 )
p( 4, 4, 4 ); // rewrite => p( 4, 4, 4 )
```

Here the missing arguments are inserted from the default values in the parameter list. Implementationally, the compiler simply rewrites missing default values into explicit positional arguments. The advantages of default values are:

- Routines with a large number of parameters are often very generalized, giving a programmer a number of different options on how a computation is performed. For many of these kinds of routines, there are standard or default settings that work for the majority of computations. Without default values for parameters, a programmer is forced to specify these common values all the time, resulting in long argument lists which are error prone.
- When a routine's interface is augmented with new parameters, it extends the interface providing generalizability⁷ (somewhat like the generalization provided by inheritance for classes). That is, all existing calls are still valid, although the call must still be recompiled.

The only disadvantage of default arguments is that unintentional omission of an argument may not result in a compiler-time error. Instead, a default value is used, which may not be the programmer's intent.

Default values may only appear in a prototype versus definition context:

```
void p( int x, int y = 2, int z = 3 );      // prototype: allowed
void p( int, int = 2, int = 3 );          // prototype: allowed
void p( int x, int y = 2, int z = 3 ) { }  // definition: not allowed
```

The reason for this restriction is to allow separate compilation. Multiple prototypes with different default values is an error.

⁷"It should be possible for the implementor of an abstraction to increase its generality. So long as the modified abstraction is a generalization of the original, existing uses of the abstraction will not require change. It might be possible to modify an abstraction in a manner which is not a generalization without affecting existing uses, but, without inspecting the modules in which the uses occur, this possibility cannot be determined. This criterion precludes the addition of parameters, unless these parameters have default or inferred values that are valid for all possible existing applications." [CW90, p. 128]

When combining positional and named arguments, the general rule in most programming languages is that all positional arguments precede all named arguments. For example, given the routine and calls:

```
void p( int x, int y, int z ) { ... }
p( 4, z : 3, y : 7 );           // allowed
p( z : 3, 4, y : 7 );          // not allowed
```

only the first call is allowed as all positional arguments precede the named arguments. C \forall adopts this restriction. The restriction is not technical; given a specific meaning for intermixing, such as named arguments are ignored when determining positional parameters, the compiler can correctly deal with intermixed positional and named arguments. However, any form of intermixing blurs the meaning of positional arguments, making it is confusing for people. For example, given the previous meaning for intermixing, the call:

```
p( z : 3, 4, y : 7 );           // => p( 4, 7, 3 )
```

but now the argument at position 2 is actually at position 1. Essentially, there is no rule for intermixing that preserves the intuitive meaning of positional for programmers.

When defining positional and default parameters for a routine, the general rule is that all non-default parameters must precede all default parameters. Again, this restriction is not technical; it is possible to use empty positional arguments to designate default usage. For example, given the routine and calls:

```
void p( int x, int y = 2, int z = 3 ) { ... }
p( 1, /* default */, 5 );       // => p( 1, 2, 5 )
p( 1, 8, /* default */ );      // => p( 1, 8, 3 )
p( 1, 8 );                      // => p( 1, 8, 3 )
```

the empty argument allows precise specification of when to use default values. C \forall adopts empty arguments in calls (see also Section 20 for other uses of empty list items).

Ellipse ("...") arguments present problems when used with named and default arguments. The conflict occurs because both named and ellipse arguments must appear after positional arguments, giving two possibilities:

```
p( /* positional */, ..., /* named */ );
p( /* positional */, /* named */, ... );
```

While it is possible to implement both approaches, the first possibly is more complex than the second, e.g.:

```
p( int x, int y, int z, ... );
p( 1, 4, 5, 6, z : 3, y : 2 ); // assume p( /* positional */, ..., /* named */ );
p( 1, z : 3, y : 2, 4, 5, 6 ); // assume p( /* positional */, /* named */, ... );
```

In the first call, it is necessary for the programmer to conceptually rewrite the call, changing named arguments into positional, before knowing where the ellipse arguments begin. Hence, this approach seems significantly more difficult, and hence, confusing and error prone. In the second call, the named arguments separate the positional and ellipse arguments, making it trivial to read the call.

The problem is exacerbated with default arguments, e.g.:

```
void p( int x, int y = 2, int z = 3... );
p( 1, 4, 5, 6, z : 3 ); // assume p( /* positional */, ..., /* named */ );
p( 1, z : 3, 4, 5, 6 ); // assume p( /* positional */, /* named */, ... );
```

The first call is an error because arguments 4 and 5 are actually positional not ellipse arguments; therefore, argument 5 subsequently conflicts with the named argument `z : 3`. In the second call, the default value for `y` is implicitly inserted after argument 1 and the named arguments separate the positional and ellipse arguments, making it trivial to read the call. For these reasons, C \forall requires named arguments before ellipse arguments. Finally, while ellipse arguments are needed for a small set of existing C routines, like `printf`, the extended C \forall type system largely eliminates the need for ellipse arguments (see Section 24), making much of this discussion moot.

Default arguments and overloading (see Section 24) are complementary. While in theory default arguments can be simulated with overloading, as in:

default arguments	overloading
<code>void p(int x, int y = 2, int z = 3) { ... }</code>	<code>void p(int x, int y, int z) { ... }</code> <code>void p(int x) { p(x, 2, 3); }</code> <code>void p(int x, int y) { p(x, y, 3); }</code>

the number of required overloaded routines is linear in the number of default values, which is unacceptable growth. In general, overloading should only be used over default arguments if the body of the routine is significantly different.

Furthermore, overloading cannot handle accessing default arguments in the middle of a positional list, via a missing argument, such as:

```
p( 1, /* default */, 5 );      // => p( 1, 2, 5 )
```

Given the $C\forall$ restrictions above, both named and default arguments are backwards compatible. C++ only supports default arguments; Ada supports both named and default arguments.

12 Tuples

In C and $C\forall$, lists of elements appear in several contexts, such as the parameter list for a routine call. (More contexts are added shortly.) A list of such elements is called a **tuple**.

The general syntax of a tuple is:

```
[ exprlist ]
```

where *exprlist* is a list of one or more expressions separated by commas. The brackets, [], allow differentiating between tuples and expressions containing the C comma operator. The following are examples of tuples:

```
[x, y, z]
[2]
[v+w, x*y, 3.14159, f()]
```

Tuples are permitted to contain sub-tuples (i.e., nesting), such as [[14, 21], 9], which is a 2-element tuple whose first element is itself a tuple. Note, a tuple is *not* a record (structure); a record denotes a single value with substructure, whereas a tuple is multiple values with no substructure (see flattening coercion in Section 12.1). In essence, tuples are largely a compile time phenomenon, having little or no runtime presence.

Tuples can be stored in **tuple variables**; these variables are of **tuple type**. Tuple variables and types can be used anywhere lists of conventional variables and types can be used. The general syntax of a tuple type is:

```
[ typelist ]
```

where *typelist* is a list of one or more legal $C\forall$ or ANSI type specifications separated by commas, which may include other tuple type specifications. Examples of tuple types include:

```
[unsigned int, char]
[double, double, double]
[* int, int *]           /* mix of CFA and ANSI */
[* [10] int, * * char, * [[int, int]] (int, int)]
```

Like tuples, tuple types may be nested, such as [[**int, int**], **int**], which is a 2-element tuple type whose first element is itself a tuple type.

Examples of declarations using tuple types are:

```
[int, int] x;           /* 2 element tuple, each element of type int */
* [char, char] y;     /* pointer to a 2 element tuple */
[[int, int]] z ([int, int]);
```

The last example declares an external routine that expects a 2 element tuple as an input parameter and returns a 2 element tuple as its result.

As mentioned, tuples can appear in contexts requiring a list of value, such as an argument list of a routine call. In unambiguous situations, the tuple brackets may be omitted, e.g., a tuple that appears as an argument may have its square brackets omitted for convenience; therefore, the following routine invocations are equivalent:

```
f( [1, x+2, fred()] );
f( 1, x+2, fred() );
```

Also, a tuple or a tuple variable may be used to supply all or part of an argument list for a routine expecting multiple input parameters or for a routine expecting a tuple as an input parameter. For example, the following are all legal:

```

[int, int] w1;
[int, int, int] w2;
[void] f (int, int, int);      /* three input parameters of type int */
[void] g ([int, int, int]);    /* 3 element tuple as input */

f( [ 1, 2, 3 ] );
f( w1, 3 );
f( 1, w1 );
f( w2 );
g( [ 1, 2, 3 ] );
g( w1, 3 );
g( 1, w1 );
g( w2 );

```

Note, in all cases 3 arguments are supplied even though the syntax may appear to supply less than 3. As mentioned, a tuple does not have structure like a record; a tuple is simply converted into a list of components.

- The present implementation of C^v does not support nested routine calls when the inner routine returns multiple values; i.e., a statement such as `g(f())` is not supported. Using a temporary variable to store the results of the inner routine and then passing this variable to the outer routine works, however. □

A tuple can contain a C comma expression, provided the expression containing the comma operator is enclosed in parentheses. For instance, the following tuples are equivalent:

```

[1, 3, 5]
[1, (2, 3), 5]

```

The second element of the second tuple is the expression `(2, 3)`, which yields the result 3. This requirement is the same as for comma expressions in argument lists.

Type qualifiers, i.e., **const** and **volatile**, may modify a tuple type. The meaning is the same as for a type qualifier modifying an aggregate type [Int99, § 6.5.2.3(7), § 6.7.3(11)], i.e., the qualifier is distributed across all of the types in the tuple, e.g.:

```
const volatile [int, float, const int] x;
```

is equivalent to:

```
[const volatile int, const volatile float, const volatile int] x;
```

Declaration qualifiers can only appear at the start of a C^v tuple declaration⁴, e.g.:

```
extern [int, int] w1;
static [int, int, int] w2;
```

- Unfortunately, C's syntax for subscripts precluded treating them as tuples. The C subscript list has the form `[i][j] ..` and not `[i, j, ...]`. Therefore, there is no syntactic way for a routine returning multiple values to specify the different subscript values, for example, `f[g()]` always means a single subscript value because there is only one set of brackets. Fixing this requires a major change to C because the syntactic form `M[i,j,k]` already has a particular meaning: `i,j,k` is a comma expression (see Section 22). □

12.1 Tuple Coercions

There are four coercions that can be performed on tuples and tuple variables: closing, opening, flattening and structuring. In addition, the coercion of dereferencing can be performed on a tuple variable to yield its value(s), as for other variables.

A **closing coercion** takes a set of values and converts it into a tuple value, which is a contiguous set of values, as in:

```
[int, int, int, int] w;
w = [1, 2, 3, 4];
```

First the right-hand tuple is closed into a tuple value and then the tuple value is assigned.

An **opening coercion** is the opposite of closing; a tuple value is converted into a tuple of values, as in:

```
[a, b, c, d] = w
```

w is implicitly opened to yield a tuple of four values, which are then assigned individually.

A **flattening coercion** coerces a nested tuple, i.e., a tuple with one or more components, which are themselves tuples, into a flattened tuple, which is a tuple whose components are not tuples, as in:

```
[a, b, c, d] = [1, [2, 3], 4];
```

First the right-hand tuple is flattened and then the values are assigned individually. Flattening is also performed on tuple types. For example, the type `[int, [int, int], int]` can be coerced, using flattening, into the type `[int, int, int, int]`.

A **structuring coercion** is the opposite of flattening; a tuple is structured into a more complex nested tuple. For example, structuring the tuple `[1, 2, 3, 4]` into the tuple `[1, [2, 3], 4]` or the tuple type `[int, int, int, int]` into the tuple type `[int, [int, int], int]`.

In the following example, the last assignment illustrates all the tuple coercions:

```
[int, int, int, int] w = [1, 2, 3, 4];
int x = 5;
[x, w] = [w, x];    /* all four tuple coercions */
```

Starting on the right-hand tuple in the last assignment statement, w is opened, producing a tuple of four values; therefore, the right-hand tuple is now the tuple `[[1, 2, 3, 4], 5]`. This tuple is then flattened, yielding `[1, 2, 3, 4, 5]`, which is structured into `[1, [2, 3, 4, 5]]` to match the tuple type of the left-hand side. The tuple `[2, 3, 4, 5]` is then closed to create a tuple value. Finally, x is assigned 1 and w is assigned the tuple value using multiple assignment (see Section 14).

- A possible additional language extension is to use the structuring coercion for tuples to initialize a complex record with a tuple. □

13 Mass Assignment

C \forall permits assignment to several variables at once using **mass assignment** [LAB⁺81]. Mass assignment has the following form:

```
[ lvalue, ..., lvalue ] = expr ;
```

The left-hand side is a tuple of *lvalues*, which is a list of expressions each yielding an address, i.e., any data object that can appear on the left-hand side of a conventional assignment statement. *expr* is any standard arithmetic expression. Clearly, the types of the entities being assigned must be type compatible with the value of the expression.

Mass assignment has parallel semantics, e.g., the statement:

```
[x, y, z] = 1.5;
```

is equivalent to:

```
x = 1.5; y = 1.5; z = 1.5;
```

This semantics is not the same as the following in C:

```
x = y = z = 1.5;
```

as conversions between intermediate assignments may lose information. A more complex example is:

```
[i, y[i], z] = a + b;
```

which is equivalent to:

```
t = a + b;
a1 = &i; a2 = &y[i]; a3 = &z;
*a1 = t; *a2 = t; *a3 = t;
```

The temporary t is necessary to store the value of the expression to eliminate conversion issues. The temporaries for the addresses are needed so that locations on the left-hand side do not change as the values are assigned. In this case, `y[i]` uses the previous value of i and not the new value set at the beginning of the mass assignment.

14 Multiple Assignment

C \forall also supports the assignment of several values at once, known as **multiple assignment** [LAB⁺81, Gal96]. Multiple assignment has the following form:

```
[ lvalue, ..., lvalue ] = [ expr, ..., expr ];
```

The left-hand side is a tuple of *lvalues*, and the right-hand side is a tuple of *exprs*. Each *expr* appearing on the right-hand side of a multiple assignment statement is assigned to the corresponding *lvalue* on the left-hand side of the statement using parallel semantics for each assignment.

An example of multiple assignment is:

```
[x, y, z] = [1, 2, 3];
```

Here, the values 1, 2 and 3 are assigned, respectively, to the variables x, y and z. A more complex example is:

```
[i, y[i], z] = [1, i, a + b];
```

Here, the values 1, i and a + b are assigned to the variables i, y[i] and z, respectively. Note, the parallel semantics of multiple assignment ensures:

```
[x, y] = [y, x];
```

correctly interchanges (swaps) the values stored in x and y.

The following cases are errors:

```
[a, b, c] = [1, 2, 3, 4];
```

```
[a, b, c] = [1, 2];
```

because the number of entities in the left-hand tuple is unequal with the right-hand tuple.

As for all tuple contexts in C, side effects should not be used because C does not define an ordering for the evaluation of the elements of a tuple; both these examples produce indeterminate results:

```
f(x++, x++);          /* C routine call with side effects in arguments */
[v1, v2] = [x++, x++]; /* side effects in RHS of multiple assignment */
```

15 Cascade Assignment

As in C, C \forall mass and multiple assignments can be cascaded, producing **cascade assignment**. Cascade assignment has the following form:

```
tuple = tuple = ... = tuple;
```

and it has the same parallel semantics as for mass and multiple assignment. Some examples of cascade assignment are:

```
x1 = y1 = x2 = y2 = 0;
[x1, y1] = [x2, y2] = [x3, y3];
[x1, y1] = [x2, y2] = 0;
[x1, y1] = z = 0;
```

As in C, the rightmost assignment is performed first, i.e., assignment parses right to left.

16 Record Field Tuples

Tuples may be used to select multiple fields of a record by field name. Its general form is:

```
expr . [ fieldlist ]
expr -> [ fieldlist ]
```

expr is any expression yielding a value of type record, e.g., **struct**, **union**. Each element of *fieldlist* is an element of the record specified by *expr*.

A record-field tuple may be used anywhere a tuple can be used. An example of the use of a record-field tuple is the following:

```
struct s {
    int f1, f2;
    char f3;
    double f4;
} v;

v.[f3, f1, f2] = [ 'x', 11, 17 ]; /* equivalent to v.f3 = 'x', v.f1 = 11, v.f2 = 17 */
f( v.[f3, f1, f2] ); /* equivalent to f( v.f3, v.f1, v.f2 ) */
```

Note, the fields appearing in a record-field tuple may be specified in any order; also, it is unnecessary to specify all the fields of a **struct** in a multiple record-field tuple.

If a field of a **struct** is itself another **struct**, multiple fields of this subrecord can be specified using a nested record-field tuple, as in the following example:

```

struct inner {
    int f2, f3;
};
struct outer {
    int f1;
    struct inner i;
    double f4;
} o;

o.[f1, i.[f2, f3], f4] = [11, 12, 13, 3.14159];

```

17 Labelled Break/Continue

While C provides **break** and **continue** statements for altering control flow, both are restricted to one level of nesting for a particular control structure. Unfortunately, this restriction forces programmers to use **goto** to achieve the equivalent for more than one level of nesting. To prevent having to make this switch, the **break** and **continue** are extended with a target label to support static multi-level exit [Buh85, GJS96]. For the labelled **break**, it is possible to specify which control structure is the target for exit, as in:

C	C \forall
<pre> for (...) { for (...) { for (...) { ... goto L1; goto L2; goto L3; // or break ... } L3; ; } L2; ; } L1; ; </pre>	<pre> L1: for (...) { L2: for (...) { L3: for (...) { ... break L1; break L2; break L3; // or break ... } } } </pre>

The inner most loop has three exit points, which cause termination of one or more of the three nested loops, respectively. For the labelled **continue**, it is possible to specify which control structure is the target for the next loop iteration, as in:

C	C \forall
<pre> for (...) { for (...) { for (...) { ... goto L1; goto L2; goto L3; ... } L3; ; } L2; ; } L1; ; </pre>	<pre> L1: for (...) { L2: for (...) { L3: for (...) { ... continue L1; continue L2; continue L3; ... } } } </pre>

The inner most loop has three restart points, which cause the next loop iteration to begin, respectively. For both **break** and **continue**, the target label must be directly associated with a **for**, **while** or **do** statement; for **break**, the target label can also be associated with a **switch** statement. Both **break** and **continue** with target labels are simply a **goto** restricted in the following ways:

- They cannot be used to create a loop. This means that only the looping construct can be used to create a loop. This restriction is important since all situations that can result in repeated execution of statements in a program are clearly delineated.

- Since they always transfers out of containing control structures, they cannot be used to branch into a control structure.

The advantage of the labelled **break/continue** is that it allows static multi-level exits without having to use the **goto** statement and ties control flow to the target control structure rather than an arbitrary point in a program. Furthermore, the location of the label at the *beginning* of the target control structure informs the reader that complex control-flow is occurring in the body of the control structure. With **goto**, the label at the end of the control structure fails to convey this important clue early enough to the reader. Finally, using an explicit target for the transfer instead of an implicit target allows new nested loop or **switch** constructs to be added or removed without affecting other constructs. The implicit targets of the current **break** and **continue**, i.e., the closest enclosing loop or **switch**, change as certain constructs are added or removed.

18 Switch Statement

C allows a number of questionable forms for the **switch** statement:

1. By default, the end of a case clause⁸ “falls through” to the next case clause in the **switch** statement; to exit a **switch** statement from a case clause requires explicitly terminating the clause with a transfer statement, most commonly **break**, as in:

```
switch ( i ) {
  case 1:
    ...
    // fall-through
  case 2:
    ...
    break; // exit switch statement
}
```

The ability to fall-through to the next clause is a useful form of control flow, specifically when a sequence of case actions compound, as in:

```
switch ( argc ) {
  case 3:
    // open output file
    // fall-through
  case 2:
    // open input file
    break; // exit switch statement
  default:
    // usage message
}
```

In this example, case 2 is always done if case 3 is done. This control flow is difficult to simulate with **if** statements or a **switch** statement without fall-through as code must be duplicated or placed in a separate routine. C also uses fall-through to handle multiple case-values resulting in the same action, as in:

```
switch ( i ) {
  case 1: case 3: case 5: // odd values
    // same action
    break;
  case 2: case 4: case 6: // even values
    // same action
    break;
}
```

However, this situation is handled in other languages without fall-through by allowing a list of case values.

⁸In this section, the term “case clause” refers to either a **case** or **default** clause.

While fall-through itself is not a problem, the problem occurs when fall-through is the default, as this semantics is not intuitive to most programmers and is different from virtually all other programming languages with a **switch** statement. Hence, default fall-through semantics results in a large number of programming errors as programmers often forget the **break** statement at the end of a case clause, resulting in inadvertent fall-through.

2. It is possible to place case clauses on statements nested *within* the body of the **switch** statement, as in:

```
switch ( i ) {
  case 0:
    if ( j < k ) {
      ...
      case 1:      // transfer into "if" statement
        if ( j < m ) {
          ...
          case 2:  // transfer into "if" statement
            ...
        }
    }
  case 3:
    while ( j < 5 ) {
      ...
      case 4:      // transfer into "while" statement
        ...
    }
}
```

The problem with this usage is branching *into* control structures, which is known to cause both comprehension and technical difficulties. The comprehension problem occurs from the inability to determine how control reaches a particular point due to the number of branches leading to it. The technical problem results from the inability to ensure allocation and initialization of variables when blocks are not entered at the beginning. Often transferring into a block can bypass variable declaration and/or its initialization, which results in subsequent errors. There are virtually no positive arguments for this kind of control flow, and therefore, there is a strong impetus to eliminate it.

Nevertheless, C does have an idiom where this capability is used, known as “Duff’s device” [Duf83]:

```
register int n = (count + 7) / 8;
switch ( count % 8 ) {
  case 0: do{ *to = *from++;
  case 7:  *to = *from++;
  case 6:  *to = *from++;
  case 5:  *to = *from++;
  case 4:  *to = *from++;
  case 3:  *to = *from++;
  case 2:  *to = *from++;
  case 1:  *to = *from++;
          } while ( --n > 0 );
}
```

which unrolls a loop N times ($N = 8$ above) and uses the **switch** statement to deal with any iterations not a multiple of N . While efficient, this sort of special purpose usage is questionable:

Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. [Duf83]

3. It is possible to place the **default** clause anywhere in the list of labelled clauses for a **switch** statement, rather than only at the end. Virtually all programming languages with a **switch** statement require the **default** clause to appear last in the case-clause list. The logic for this semantics is that after checking all the **case** clauses without

success, the **default** clause is selected, and hence, physically placing the **default** clause at the end of the case-clause list matches with this semantics. This physical placement can be compared to the physical placement of an **else** clause at the end of a series of connected **if/else if** statements.

4. It is possible to place unreachable code at the start of a **switch** statement, as in:

```
switch ( x ) {
    int y = 1;           // unreachable initialization
    x = 7;              // unreachable code
    case 3: ...
        y = 3;
        ...
}
```

While the declaration of the local variable *y* is useful and its scope is across all case clauses, the initialization for such a variable is defined to never be executed because control always transfers over it. Furthermore, any statements before the first case clause can only be executed if labelled and transferred to using a **goto**, either from outside or inside of the **switch**. As mentioned, transfer *into* control structures should be forbidden. Transfers from *within* the **switch** body using a **goto** are equally unpalatable.

Before discussing potential language changes to deal with these problems, it is worth observing that in a typical C program:

- the number of **switch** statements is small,
- most **switch** statements are well formed (i.e., no Duff's device),
- the **default** clause is usually written as the last case-clause,
- and there is only a medium amount of fall-through from one case clause to the next, and most of these result from a list of case values executing common code, rather than a sequence of case actions that compound.

These observations should help to put the effects of suggested changes into perspective. Figure 1 shows the grammar change that attempts to minimize the effect on existing C programs.

1. Eliminating the default fall-through problem has the greatest potential for affecting existing code. However, even if fall-through is removed, most **switch** statements would continue to work because of the explicit transfers already present at the end of each case clause, and the common placement of the **default** clause at the end of the case list. In addition, the above grammar provides for the most common use of fall-through, i.e., a list of **case** clauses executing common code, e.g.:

```
case 1: case 2: case 3: ...
```

Nevertheless, reversing the default action would have a non-trivial effect on case actions that compound, such as the above example of processing shell arguments. Therefore, to preserve backwards compatibility, it is necessary to introduce a new kind of switch statement, called **choose**, with no fall-through semantics. The **choose** statement is identical to the new **switch** statement, except there is no implicit fall-through between case-clauses and the **break** statement applies to the enclosing loop construct (as for the **continue** statement in a **switch** statement). It is still possible to fall-through if a case-clause ends with the new keyword **fallthru**, e.g.:

```
choose ( i ) {
    int i;
    case 3, 4:
        i = 3;
        fallthru;
    case 8, 10:
    default:
        j = 3;
}
```

The ability to fall-through is retained because it is a sufficient C-idiom that most C programmers simply expect it, and its absence might discourage these programmers from using the **choose** statement.

```

selection_statement:
    IF '(' comma_expression ')' statement
    IF '(' comma_expression ')' statement ELSE statement
    SWITCH '(' comma_expression ')' case_clause
    SWITCH '(' comma_expression ')' '{' declaration_list_opt switch_clause_list_opt '}'
    CHOOSE '(' comma_expression ')' case_clause
    CHOOSE '(' comma_expression ')' '{' declaration_list_opt choose_clause_list_opt '}'

case_value_list:
    constant_expression
    case_value_list ',' constant_expression

case_label:
    CASE case_value_list ':'
    DEFAULT ':'

case_label_list:
    case_label
    case_label_list case_label

case_clause:
    case_label_list statement

switch_clause_list_opt:
    /* empty */
    switch_clause_list

switch_clause_list:
    case_label_list statement_list
    switch_clause_list case_label_list statement_list

choose_clause_list_opt:
    /* empty */
    choose_clause_list

choose_clause_list:
    case_label_list fall_through
    case_label_list statement_list fall_through_opt
    choose_clause_list case_label_list fall_through
    choose_clause_list case_label_list statement_list fall_through_opt

fall_through_opt:
    /* empty */
    fall_through

fall_through:
    FALLTHRU
    FALLTHRU ';'

```

Figure 1: Switch/Choose Statement Grammar

2. Eliminating Duff's device is straightforward and only invalidates a small amount of very questionable code. The solution is to allow case clauses to only appear at the same nesting level as the switch body, as is done in most other programming languages with **switch** statements.
3. The issue of **default** at locations other than at the end of the cause clause can be solved by using good programming style, and there are a few reasonable situations involving fall-through where the **default** clause may appear in locations other than at the end. Therefore, no language change is made for this issue.
4. Dealing with unreachable code at the start of a **switch** statement is solved by defining the *declaration-list*, including any associated initialization, at the start of a **switch** statement body to be executed *before* the transfer to the appropriate case clause. This semantics is the same as for declarations at the start of a loop body, which are executed before each iteration of the loop body. As well, this grammar does not allow statements to appear before the first case clause. The change is compatible for declarations with initialization in this context because existing code cannot assume the initialization has occurred. The change is incompatible for statements, but any existing code using it is highly questionable, as in:

```

switch ( i ) {
    L: x = 5;    // questionable code
    case 0:
        ...
}

```

The statement after the **switch** can never be executed unless it is labelled. If it is labelled, it must be transferred to from outside or inside the switch statement, neither of which is acceptable control flow.

19 Case Clause

C restricts the **case** clause of a **switch** statement to a single value. For multiple **case** clauses associated with the same statement, it is necessary to have multiple **case** clauses rather than multiple values. Requiring a **case** clause for each value does not seem to be in the spirit of brevity normally associated with C. Therefore, the **case** clause is extended with a list of values, as in:

<pre> C∀ switch (i) { case 1, 3, 5: ... case 2, 4, 6: ... } </pre>	<pre> C switch (i) { case 1: case 3 : case 5: /* odd values */ ... case 2: case 4 : case 6: /* even values */ ... } </pre>
--	--

In addition, two forms of subranges are allowed to specify case values: the GNU C form and a new C∀ form.

<pre> GNU C switch (i) { case 1 ... 5: ... case 10 ... 15: ... } </pre>	<pre> C∀ switch (i) { case 1~5 /* 1, 2, 3, 4, 5 */ ... case 10~15 /* 10, 11, 12, 13, 14, 15 */ ... } </pre>
---	---

20 Unnamed Structure Fields

C requires each field of a structure to have a name, except for a bit field associated with a basic type, e.g.:

```

struct {
    int f1;           /* named field */
    int f2 : 4;      /* named field with bit field size */
    int : 3;         /* unnamed field for basic type with bit field size */
    int ;           /* disallowed, unnamed field */
    int *;          /* disallowed, unnamed field */
    int (*)(int);   /* disallowed, unnamed field */
};

```

This requirement is relaxed by making the field name optional for all field declarations; therefore, all the field declarations in the example are allowed. As for unnamed bit fields, an unnamed field is used for padding a structure to a particular size. A list of unnamed fields is also supported, e.g.:

```
struct {
    int , , ;           /* 3 unnamed fields */
}
```

21 Attribute

During compilation, a significant amount of important information is accumulated in the compiler's symbol table, i.e., multiple attributes exist for each symbol-table entry. Furthermore, a compiler must know a significant amount about the target architecture to generate code. Unfortunately, most of this useful attribute information is unavailable to a programmer. Currently, the only accessible attribute is the size of a type via the pseudo-routine **sizeof**. GNU C provides access to a variable or an expression type using the pseudo-routine **typeof**. However, a symbol table contains other important and useful information. An additional attempt at providing target-architecture information is provided in C via **#include** files such as `limits.h`, `float.h`, etc. However, include files are not the best approach because they are not part of the compiler, which means they can become inconsistent. Furthermore, include files should be sharable among computers through a common file system, but this approach forces specific include files for each architecture. Finally, this style of include file contains many **#define**, which in effect introduces new keywords into the language, as these names cannot be used in a program nor can the compiler detect any renaming conflicts.

A new attribute capability is added similar to the one available in Ada, which is of the form:

```
{ variable-name | '<' type-name '>' } '! ' attribute-name
```

Attribute names are in a different name space, and hence, do not introduce new keywords into the language. The punctuation symbol used to introducing an attribute is the exclamation mark or bang, `!`, which is currently used only as a unary operator. A disadvantage is that it precludes the unary operator `!` from being used as a binary operator in the future, which may or may not be an issue. (There seems to be no precedent for binary `!` in any common programming language.)

Exact attribute names have not been selected. These names can be selected based on prior art, such as **sizeof**, **typeof**, `CHAR_BIT`, `INT_MIN`, `INT_MAX`, etc. However, most of these prior-art names can be simplified, e.g., it is unnecessary to have the type name in the attribute name because the type is implied from the variable or type name, as in:

```
int x = <short int>!max;    /* maximum int */
int y = x!size;           /* size of int */
```

Some attribute names with obvious meaning (as above) are used in further discussion where attribute information is necessary. Notice the attribute size makes superfluous the unary operator **sizeof**, which could ultimately be deprecated and subsequently removed.

The following is a list of possible attributes (many of these come from Ada):

```
!alignment
!base
!bit_order
!delta
!digits
!exponent
```

It is possible to use a **type** attribute to determine the type of a variable (same as GNU C **typeof**), and it can be used to declare variables of that type, as in:

```
<x> y;                    /* same type as x */
<x> *z;                   /* pointer to same type as x */
const <z> *w;            /* pointer to same constant type as x */
```

22 New Arrays

Arrays are one of the biggest stumbling blocks in C for most programmers. The reason is that arrays are not a first class type, but pointers with special syntax for pointer arithmetic (i.e., subscripting). Therefore, the number and size of the dimensions for an array are often unknown to the compiler, making it impossible to provide adequate support to

the programmer, such as subscript checking, conformant array checking, dynamic array allocation on either the stack or heap, subarrays, etc. Unless the problems with arrays are fixed, C is unacceptable as a language for teaching or mathematical/scientific work.

The following new style arrays are suggested:

$C\forall$	C
<code>[0~9,0~9,0~9] int a;</code>	<code>int a[10][10][10];</code>

New style array specification can also be used with ANSI style declarations:

$C\forall$	C
<code>int a[0~9,0~9,0~9];</code>	<code>int a[10][10][10];</code>

New style arrays must use a subrange because the trivial case `int x[10]` cannot be distinguished as a new or old style array.

Formal array definition are possible with or without conformant dimensions:

$C\forall$	C
<code>void f([1~] int);</code>	N/A
<code>void f(int [1~,~10]);</code>	N/A
<code>void f([~,~,~] int);</code>	<code>void f(int [][*][*]);</code>
<code>void f(int [~,~,~]);</code>	<code>void f(int [][*][*]);</code>

Finally, there is a new style of subscript:

$C\forall$	C
<code>a[1,2,3] = 3;</code>	<code>a[1][2][3] = 3;</code>

A new subscript can contain a C comma expression, provided the expression containing the comma operator is enclosed in parentheses. For instance, the following subscripts are equivalent:

```
[1, 3, 5]
[1, (2, 3), 5]
```

The second element of the second subscript is the expression `(2, 3)`, which yields the result 3. This requirement is the same as for comma expressions in argument lists.

New arrays have array descriptors, which are passed around to routines.

Access the bounds of the array with attributes.

New style subscripts are tuple contexts, like arguments, and hence, support all the same tuple usages.

In many cases, it should be possible to pass a new array to a routine expecting an old array, but not vice versa.

23 Exception Handling

```
exception x;
```

```
try {
} catch( ) {
} resume( ) {
}
```

24 Parametric Polymorphism

25 Concurrency

26 Example $C\forall$ Program

Figure 2 shows a brief example of the features of $C\forall$.

References

- [BDMN80] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA begin*. Studentlitteratur, Lund, Sweden, second edition, 1980.
- [BTZ94] P. A. Buhr, David Till, and C. R. Zarnke. Assignment as the Sole Means of Updating Objects. *Software—Practice and Experience*, 24(9):835–870, September 1994.

```

* int x11 = 0, x12 = 0; /* pointer to int for both x11 and x12 */
int * x21 = 0, x22 = 0; /* pointer to int for x21 and int for x22 */

[20] int y1, y2 = { 1, 2, 3 }; /* two arrays of 20 integers */
* [20] double z; /* pointer to array of doubles */
[20] * char w; /* array of pointers to char */

[int rc, int w] h ( int, double ); /* multiple return values */
static [int] g ( int a, int b, * int c, [ ] char d );

extern int v1;
static * int v2;
*[ ][ ]* [ ][ ][ ] int)( *[ ][ ] int, *[ ][ ] int ) v3; /* what is it? */

struct S {
    * int f1; /* new and old styles */
    int *f2;
    * [ int](int) f3; /* new and old styles */
    int *(f4)(int);
} s;

union U {
    [5] int f1; /* new and old styles */
    int f2[5];
    * int f3; /* new and old styles */
    int *f4;
} u;

[int rc] printf ( * char fmt, ... ); /* new routine prototype */

[short x, unsigned y] f( int w ) {
    [y, x] = [x, y] = [w, 23]; /* cascade, multiple assignment */
}

[[int, char, long, int] r] g() {
    short x;
    unsigned int y;
    [int, int] z; /* tuple declaration */

    [x, y, z] = [p, f( 17 ), 3]; /* multiple assignment */
    r = [x, y, z];
}

[int rc] main( int argc, ** char argv ) {
    struct {
        int f1, f2, f3, f4;
    } s;

    s.[f1, f2, f3, f4] = g(); /* field tuple assignment */
    printf( "expecting 3, 17, 23, 4; got %d, %d, %d, %d\n", s.[f4, f3, f2, f1] );
    rc = 0;
}

```

Figure 2: C∀ Example Program

- [Buh85] P. A. Buhr. A Case for Teaching Multi-exit Loops to Beginning Programmers. *SIGPLAN Notices*, 20(11):14–22, November 1985.
- [CW90] G. V. Cormack and A. K. Wright. Type-dependent Parameter Inference. *SIGPLAN Notices*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.
- [Dit92] Glen Jeffrey Ditchfield. *Contextual Polymorphism*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1992. <ftp://plg.uwaterloo.ca/pub/theses/DitchfieldThesis.ps.gz>.
- [Duf83] Tom Duff. Duff's Device. newsgroup message, see <http://www.lysator.liu.se/c/duffs-device.html>, November 1983.
- [For97] Unicom, Inc., 7660 E. Broadway, Tucson, Arizona, U.S.A, 85710. *Fortran 95 Standard, ISO/IEC 1539*, January 1997.
- [Fra77] Nissim Francez. Another Advantage of Key word Notation for Parameter Communication with Subprograms. *Communications of the ACM*, 20(8):604–605, August 1977.
- [Gal96] John Galletly. *OCCAM 2: Including OCCAM 2.1*. UCL (University College London) Press Ltd., second edition, 1996.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [Har76] W. T. Hardgrave. Positional versus Keyword Parameter Communication in Programming Languages. *SIGPLAN Notices*, 11(5):52–58, May 1976.
- [Int95] Intermetrics, Inc. *Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) with COR.1:2000 edition, December 1995. Language and Standards Libraries.
- [Int99] International Standard ISO/IEC 9899:1999 (E), www.ansi.org. *Programming Languages – C*, 1999.
- [LAB⁺81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
- [Til89] David W. Till. Tuples In Imperative Programming Languages. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1989.
- [WC96] Ben Werther and Damian Conway. A Modest Proposal: C++ Resyntaxed. *SIGPLAN Notices*, 31(11):74–82, November 1996.

Index

- (), 3
- *, 3
- CFA option, 2
- [], 3, 4, 8
- _, 2
- arguments
 - default, 6
 - keyword, 6
 - named, 6
- bit field, 2, 17
 - unnamed, 18
- break**, 12
 - labelled, 12
- cascade assignment, **11**
- case**, 17
- case clause, 13
- cast, 3
- cfa, 2
- choose**, 15
- closing coercion, **9**
- coercion
 - closing, 9
 - flattening, 10
 - opening, 9
 - structuring, 10
 - tuple, 9
- compilation
 - CFA option, 2
 - cfa, 2
- const**, 3, 9
- constant
 - numeric, 2
- continue**, 12
 - labelled, 12
- declaration
 - C style, 2
 - CFA style, 2
- declaration qualifier, 3–5, 9
- default arguments, 6
- definition
 - routine, 4
- Duff's device, 14
- extern**, 3
- fall-through, 13
- fallthru**, 15
- flattening coercion, **10**
- GNU C, 2
- goto**
 - restricted, 12
- input parameter, 4
- keyword arguments, 6
- mass assignment, **10**
- multi-level exit, 12
- multiple assignment, **10**
- named arguments, 6
- nested loops, 12
- opening coercion, **9**
- output parameter, 4
- parameter
 - input, 4
 - output, 4
- pointer
 - routine, 5
- prototype
 - routine, 5
- routine
 - definition, 4
 - pointer, 5
 - prototype, 5
- sizeof**, 3
- standard output, 2
- static**, 3
- static multi-level exit, 12
- struct**, 11
- structure, 17
 - unnamed fields, 17
- structuring coercion, **10**
- switch**, 13, 17
 - fall-through, 13
- tuple, **4, 8**
 - coercion, 9
 - type, 8
 - variable, 8
- tuple type, **8**
- tuple variables, **8**
- type operators, 3
- type qualifier, 3, 9
- underscore, 2
 - in constants, 2

union, 11

volatile, 3, 9