CÆ (Cforall) User Manual
Version 1.0

“describe not prescribe”

CÆ Team (past and present)
Andrew Beach, Richard Bilson, Michael Brooks, Peter A. Buhr, Thierry Delisle, Glen Ditchfield, Rodolfo G. Esteves, Aaron Moss, Colby Parsons, Rob Schluntz, Fangren Yu, Mubeen Zulfiqar
February 14, 2022

© 2016, 2018, 2021 CÆ Project
This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0.
# Contents

1. **Introduction**
   1.1 Background .................................................. 1

2. **Why fix C?** .................................................. 1

3. **History** ..................................................... 2

4. **Interoperability** ........................................... 2

5. **C99 Compilation** ............................................ 3

6. **Backquote Identifiers** ...................................... 5

7. **Constant Underscores** ...................................... 5

8. **Exponentiation Operator** ................................... 6

9. **Control Structures** ........................................ 6
   9.1 *if* / *while* Statement .................................... 6
   9.2 *case* Clause ................................................ 7
   9.3 *switch* Statement .......................................... 7
   9.4 Non-terminating and Labelled *fallthrough* ............ 10
   9.5 Loop Control .............................................. 10
   9.6 Labelled *continue* / *break* Statement ............... 12
   9.7 Extended *else* ............................................ 12
   9.8 *with* Statement ........................................... 13

10. **Exception Handling** ........................................ 15
    10.1 Exception Hierarchy ....................................... 16

11. **Alternative Declarations** ................................ 16

12. **Pointer / Reference** ...................................... 18
    12.1 Initialization ............................................. 20
    12.2 Address-of Semantics .................................... 22
    12.3 Conversions ............................................... 22

13. **Enumeration** .............................................. 23
    13.1 Enum type ................................................ 23
    13.2 Inheritance ............................................... 24

14. **Routine Definition** ....................................... 25
    14.1 Named Return Values ..................................... 26
    14.2 Routine Prototype ........................................ 26

15. **Routine Pointers** .......................................... 27

16. **Named and Default Arguments** .......................... 27

17. **Unnamed Structure Fields** ............................... 29

18. **Nesting** ................................................... 30
    18.1 Type Nesting ............................................... 30
    18.2 Routine Nesting ........................................... 30
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19 Tuple</td>
<td>31</td>
</tr>
<tr>
<td>19.1 Multiple-Return-Value Functions</td>
<td>31</td>
</tr>
<tr>
<td>19.2 Expressions</td>
<td>32</td>
</tr>
<tr>
<td>19.3 Variables</td>
<td>33</td>
</tr>
<tr>
<td>19.4 Indexing</td>
<td>33</td>
</tr>
<tr>
<td>19.5 Flattening and Structuring</td>
<td>33</td>
</tr>
<tr>
<td>19.6 Assignment</td>
<td>34</td>
</tr>
<tr>
<td>19.7 Construction</td>
<td>35</td>
</tr>
<tr>
<td>19.8 Member-Access Expression</td>
<td>36</td>
</tr>
<tr>
<td>19.9 Casting</td>
<td>37</td>
</tr>
<tr>
<td>19.10 Polymorphism</td>
<td>37</td>
</tr>
<tr>
<td>19.10.1 Assertion Inference</td>
<td>38</td>
</tr>
<tr>
<td>20 Tuples</td>
<td>39</td>
</tr>
<tr>
<td>20.1 Tuple Coercions</td>
<td>40</td>
</tr>
<tr>
<td>20.2 Mass Assignment</td>
<td>41</td>
</tr>
<tr>
<td>20.3 Multiple Assignment</td>
<td>41</td>
</tr>
<tr>
<td>20.4 Cascade Assignment</td>
<td>42</td>
</tr>
<tr>
<td>21 Stream I/O Library</td>
<td>42</td>
</tr>
<tr>
<td>21.1 Basic I/O</td>
<td>42</td>
</tr>
<tr>
<td>21.2 Implicit Separator</td>
<td>43</td>
</tr>
<tr>
<td>21.3 Separation Manipulators</td>
<td>45</td>
</tr>
<tr>
<td>21.4 Newline Manipulators</td>
<td>46</td>
</tr>
<tr>
<td>21.5 Output Value Manipulators</td>
<td>47</td>
</tr>
<tr>
<td>21.6 Input Value Manipulators</td>
<td>49</td>
</tr>
<tr>
<td>21.7 Concurrent Stream Access</td>
<td>50</td>
</tr>
<tr>
<td>21.8 Locale</td>
<td>51</td>
</tr>
<tr>
<td>22 String Stream</td>
<td>51</td>
</tr>
<tr>
<td>23 Structures</td>
<td>52</td>
</tr>
<tr>
<td>24 Constructors and Destructors</td>
<td>53</td>
</tr>
<tr>
<td>25 Overloading</td>
<td>53</td>
</tr>
<tr>
<td>25.1 Constant</td>
<td>55</td>
</tr>
<tr>
<td>25.2 Variable</td>
<td>55</td>
</tr>
<tr>
<td>25.3 Function Overloading</td>
<td>55</td>
</tr>
<tr>
<td>25.4 Operator</td>
<td>56</td>
</tr>
<tr>
<td>26 Auto Type-Inferencing</td>
<td>57</td>
</tr>
<tr>
<td>27 Concurrency</td>
<td>58</td>
</tr>
<tr>
<td>27.1 Coroutine</td>
<td>58</td>
</tr>
<tr>
<td>27.2 Monitors</td>
<td>58</td>
</tr>
<tr>
<td>27.3 Threads</td>
<td>59</td>
</tr>
<tr>
<td>28 Language Comparisons</td>
<td>59</td>
</tr>
<tr>
<td>28.1 C++</td>
<td>59</td>
</tr>
<tr>
<td>28.2 Go</td>
<td>60</td>
</tr>
<tr>
<td>28.3 Rust</td>
<td>61</td>
</tr>
<tr>
<td>28.4 D</td>
<td>61</td>
</tr>
<tr>
<td>A Syntax Ambiguities</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>Contents</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>B C Incompatibles</td>
</tr>
<tr>
<td>2</td>
<td>C C++ Keywords</td>
</tr>
<tr>
<td>3</td>
<td>D Standard Headers</td>
</tr>
<tr>
<td>4</td>
<td>E Standard Library</td>
</tr>
<tr>
<td>5</td>
<td>E.1 Storage Management</td>
</tr>
<tr>
<td>6</td>
<td>E.2 String to Value Conversion</td>
</tr>
<tr>
<td>7</td>
<td>E.3 Search / Sort</td>
</tr>
<tr>
<td>8</td>
<td>E.4 Absolute Value</td>
</tr>
<tr>
<td>9</td>
<td>E.5 Random Numbers</td>
</tr>
<tr>
<td>10</td>
<td>E.6 Algorithms</td>
</tr>
<tr>
<td>11</td>
<td>F Math Library</td>
</tr>
<tr>
<td>12</td>
<td>F.1 General</td>
</tr>
<tr>
<td>13</td>
<td>F.2 Exponential</td>
</tr>
<tr>
<td>14</td>
<td>F.3 Logarithm</td>
</tr>
<tr>
<td>15</td>
<td>F.4 Trigonometric</td>
</tr>
<tr>
<td>16</td>
<td>F.5 Hyperbolic</td>
</tr>
<tr>
<td>17</td>
<td>F.6 Error / Gamma</td>
</tr>
<tr>
<td>18</td>
<td>F.7 Nearest Integer</td>
</tr>
<tr>
<td>19</td>
<td>F.8 Manipulation</td>
</tr>
<tr>
<td>20</td>
<td>G Time Keeping</td>
</tr>
<tr>
<td>21</td>
<td>G.1 Duration</td>
</tr>
<tr>
<td>22</td>
<td>G.2 timeval</td>
</tr>
<tr>
<td>23</td>
<td>G.3 timespec</td>
</tr>
<tr>
<td>24</td>
<td>G.4 itimerval</td>
</tr>
<tr>
<td>25</td>
<td>G.5 Time</td>
</tr>
<tr>
<td>26</td>
<td>H Clock</td>
</tr>
<tr>
<td>27</td>
<td>H.1 C time</td>
</tr>
<tr>
<td>28</td>
<td>H.2 Clock</td>
</tr>
<tr>
<td>29</td>
<td>I Pseudo Random Number Generator</td>
</tr>
<tr>
<td>30</td>
<td>J Multi-precision Integers</td>
</tr>
<tr>
<td>31</td>
<td>K Rational Numbers</td>
</tr>
<tr>
<td>32</td>
<td>Index</td>
</tr>
</tbody>
</table>
1 Introduction

CV\(^1\) is a modern general-purpose concurrent programming language, designed as an evolutionary step forward for the C programming language. The syntax of CV builds from C and should look immediately familiar to C/C++ programmers. CV adds many modern features that directly lead to increased safety and productivity, while maintaining interoperability with existing C programs and achieving similar performance. Like C, CV is a statically typed, procedural (non-object-oriented) language with a low-overhead runtime, meaning there is no global garbage-collection, but regional garbage-collection is possible. The primary new features include polymorphic routines and types, exceptions, concurrency, and modules.

One of the main design philosophies of CV is to “describe not prescribe”, which means CV tries to provide a pathway from low-level C programming to high-level CV programming, but it does not force programmers to “do the right thing”. Programmers can cautiously add CV extensions to their C programs in any order and at any time to incrementally move towards safer, higher-level programming. A programmer is always free to reach back to C from CV, for any reason, and in many cases, new CV features can be locally switched back to their C counterpart. There is no notion or requirement for rewriting a legacy C program to CV; instead, a programmer evolves a legacy program into CV by incrementally incorporating CV features. As well, new programs can be written in CV using a combination of C and CV features. In many ways, CV is to C as Scala \([29]\) is to Java, providing a vehicle for new typing and control-flow capabilities on top of a highly popular programming language allowing immediate dissemination.

C++ \([30]\) had a similar goal 30 years ago, allowing object-oriented programming to be incrementally added to C. However, C++ currently has the disadvantages of a strong object-oriented bias, multiple legacy design-choices that are difficult to update, and active divergence of the language model from C, requiring significant effort and training to incrementally add C++ to a C-based project. In contrast, CV has 30 years of hindsight and a clean starting point.

Like C++, there may be both old and new ways to achieve the same effect. For example, the following programs compare the C, CV, and C++ I/O mechanisms, where the programs output the same result.

```c
#include <stdio.h>  
int main( void ) {  
  int x = 0, y = 1, z = 2;  
  printf("%d %d %d\n", x, y, z);  
}
```

```cv
#include <stream>  
using namespace std;  
void main( void ) {  
  int x = 0, y = 1, z = 2;  
  cout << x << y << z << endl;  
}
```

```cpp
#include <iostream>  
int main( void ) {  
  int x = 0, y = 1, z = 2;  
  cout << x << y << z << endl;  
}
```

While CV I/O (see Section 21, p. 42) looks similar to C++, there are important differences, such as automatic spacing between variables and an implicit newline at the end of the expression list, similar to Python \([26]\).

1.1 Background

This document is a programmer reference-manual for the CV programming language. The manual covers the core features of the language and runtime-system, with simple examples illustrating syntax and semantics of features. The manual does not teach programming, i.e., how to combine the new constructs to build complex programs. The reader must have an intermediate knowledge of control flow, data structures, and concurrency issues to understand the ideas presented, as well as some experience programming in C/C++. Implementers should refer to the CV Programming Language Specification for details about the language syntax and semantics. Changes to the syntax and additional features are expected to be included in later revisions.

2 Why fix C?

The C programming language is a foundational technology for modern computing with billions of lines of code implementing everything from hobby projects to commercial operating-systems. This installation base and the programmers producing it represent a massive software-engineering investment spanning decades and likely to continue for decades more. Even with all its problems, C continues to be popular because it allows writing software at virtually any level in a computer system without restriction. For system programming, where direct access to hardware, storage management, and real-time issues are a requirement, C is the only language of choice. The TIOBE index \([32]\) for February 2021 ranks the top six most popular programming languages as C 17.4%, Java 12%, Python 12%, C++ 7.6%, C# 4%,

\(^1\)Pronounced “C-for-all”, and written CV, CFA, or Cfforall.
Visual Basic 3.8% = 56.8%, where the next 50 languages are less than 2% each, with a long tail. The top 4 rankings over the past 35 years are:

<table>
<thead>
<tr>
<th>Year</th>
<th>C</th>
<th>Java</th>
<th>Python</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2006</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1996</td>
<td>1</td>
<td>3</td>
<td>23</td>
<td>13</td>
</tr>
<tr>
<td>1991</td>
<td>1</td>
<td>28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1986</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
</tbody>
</table>

Hence, C is still an extremely important programming language, with double the usage of C++; in many cases, C++ is often used solely as a better C. Love it or hate it, C has been an important and influential part of computer science for 40 years and its appeal is not diminishing. Nevertheless, C has many problems and omissions that make it an unacceptable programming language for modern needs.

As stated, the goal of the CV project is to engineer modern language-features into C in an evolutionary rather than revolutionary way. C++ [22, 12] is an example of a similar project; however, it largely extended the C language, and did not address many of C’s existing problems.2 Fortran [14], Cobol [6], and Ada [1] are examples of programming languages that took an evolutionary approach, where modern language-features (e.g., objects, concurrency) are added and problems fixed within the framework of the existing language. Java [18], Go [19], Rust [28] and D [3] are examples of the revolutionary approach for modernizing C/C++, resulting in a new language rather than an extension of the descendent. These languages have different syntax and semantics from C, do not interoperate directly with C, and are not systems languages because of restrictive memory-management or garbage collection. As a result, there is a significant learning curve to move to these languages, and C legacy-code must be rewritten. These costs can be prohibitive for many companies with a large software-base in C/C++, and a significant number of programmers require retraining in the new programming language.

The result of this project is a language that is largely backwards compatible with C11 [21], but fixes many of the well known C problems while adding modern language-features. To achieve these goals required a significant engineering exercise, i.e., “thinking inside the C box”. Considering the large body of existing C code and programmers, there is significant impetus to ensure C is transformed into a modern language. While C11 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 may make C complex and inefficient, it is clear a language without modern capabilities is insufficient for the advanced programming problems existing today.

3 History
The CV project started with Dave Till’s K-W C [5, 31], which extended C with new declaration syntax, multiple return values from routines, and advanced assignment capabilities using the notion of tuples (see [33] for similar work in C++). The first CV implementation of these extensions was by Rodolfo Esteves [13].

The signature feature of CV is overloadable parametric-polymorphic functions [7, 8, 11] with functions generalized using a forall clause (giving the language its name):

```fortran
forall( T ) T identity( T val ) { return val; }
```

CV’s polymorphism was originally formalized by Glen Ditchfield [9], and first implemented by Richard Bilson [2]. However, at that time, there was little interesting in extending C, so work did not continue. As the saying goes, “What goes around, comes around.”, and there is now renewed interest in the C programming language because of the legacy code-base, so the CV project was restarted in 2015.

4 Interoperability
CV is designed to integrate directly with existing C programs and libraries. The most important feature of interoperability is using the same calling conventions, so there is no complex interface or overhead to call existing C routines. This feature allows CV programmers to take advantage of the existing panoply of C libraries to access thousands of external software features. Language developers often state that adequate library support takes more work than

2Two important existing problems addressed were changing the type of character literals from int to char and enumerator from int to the type of its enumerators.
designing and implementing the language itself. Fortunately, CV, like C++, starts with immediate access to all exiting C libraries, and in many cases, can easily wrap library routines with simpler and safer interfaces, at zero or very low cost. Hence, CV begins by leveraging the large repository of C libraries, and than allows programmers to incrementally augment their C programs with modern backward-compatible features.

However, it is necessary to differentiate between C and CV code because of name overloading, as for C++. For example, the C math-library provides the following routines for computing the absolute value of the basic types: abs, labs, llabs, fabs, fabsf, fabsl, cabs, cabsf, and cabsl. Whereas, CV wraps each of these routines into one overloaded name abs:

```c
    char abs( char );
    extern "C" { int abs( int ); } // use default C routine for int
    long int abs( long int );
    long long int abs( long long int );
    float abs( float );
    double abs( double );
    long double abs( long double );
    float _Complex abs( _Complex );
    double _Complex abs( _Complex );
    long double _Complex abs( long double _Complex );
```

The problem is a name clash between the C name abs and the CV names abs, resulting in two name linkages: extern "C" and extern "Cforall" (default). Overloaded names must use name mangling to create unique names that are different from unmangled C names. Hence, there is the same need as in C++ to know if a name is a C or CV name, so it can be correctly formed. The only way around this problem is C’s approach of creating unique names for each pairing of operation and type.

This example illustrates a core idea in CV: the power of a name. The name “abs” evokes the notion of absolute value, and many mathematical types provide the notion of absolute value. Hence, knowing the name abs is sufficient to apply it to any type where it is applicable. The time savings and safety of using one name uniformly versus $N$ unique names cannot be underestimated.

## 5 CV Compilation

The command cfa is used to compile a CV program and is based on the GNU gcc command, e.g.:

```
cfa [ gcc/C/CV-flags ] [ C/CV source-files ] [ assembler/loader files ]
```

There is no ordering among options (flags) and files, unless an option has an argument, which must appear immediately after the option possibly with or without a space separating option and argument.

CV has the following gcc flags turned on:

- `-std=gnu11` The 2011 C standard plus GNU extensions.
- `-std=gnu-` Use the traditional GNU semantics for inline routines in C11 mode, which allows inline routines in header files.

CV has the following new options:

- `-CFA` Only the C preprocessor (flag -E) and the CV translator steps are performed and the transformed program is written to standard output, which makes it possible to examine the code generated by the CV translator. The generated code starts with the standard CV prelude.
- `-XCFA` Pass next flag as-is to the cfa-cpp translator (see details below).
- `-debug` The program is linked with the debugging version of the runtime system. The debug version performs runtime checks to aid the debugging phase of a CV program, but can substantially slow program execution. The runtime checks should only be removed after a program is completely debugged. This option is the default.
- `-nodebug` The program is linked with the non-debugging version of the runtime system, so the execution of the program is faster. However, no runtime checks or asserts are performed so errors usually result in abnormal program behaviour or termination.
- `-help` Information about the set of CV compilation flags is printed.
- `-nohelp` Information about the set of CV compilation flags is not printed. This option is the default.
- `-quiet` The CV compilation message is not printed at the beginning of a compilation.
The C preprocessor allows only integer values in a preprocessor variable so a value like “1.0.0” is not allowed. Hence, the need to have three variables for the major, minor and patch version number.
6 Backquote Identifiers

CWA introduces several new keywords (see Section C, p. 65) that can clash with existing C variable-names in legacy code. Keyword clashes are accommodated by syntactic transformations using the CWA backquote escape-mechanism:

```c
int "\coroutine" = 3;                     // make keyword an identifier

double "\forallall" = 3.5;                // make keyword an identifier
```

Existing C programs with keyword clashes can be converted by prefixing the keyword identifiers with double backquotes, and eventually the identifier name can be changed to a non-keyword name. Figure 1 shows how clashes in existing C header-files (see Section D, p. 65) can be handled using preprocessor `interposition`: `#include_next` and `-I` filename. Several common C header-files with keyword clashes are fixed in the standard CWA header-library, so there is a seamless programming-experience.

7 Constant Underscores

Numeric constants are extended to allow underscores as a separator, *e.g.*:

```c
2_147_483_648;                           // decimal constant
56_ul;                                  // decimal unsigned long constant
0_377;                                  // octal constant
0x_ffe_ff;                               // hexadecimal constant
0x_ef3d_aa5c;                            // hexadecimal constant
3.141_592_654;                          // floating constant
10_e_1_00;                               // floating constant
0x_f_3f_3f_p_3;                         // hexadecimal floating
0x_1.f_fff_fff_p_128_;                  // hexadecimal floating long constant
L_"\x_ffe_ee*";                         // wide character constant
```

The rules for placement of underscores are:

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, 1.0\_E\_10 or 1.0\_E\_10; the type suffixes U, L, etc. may start with an underscore 1\_U, 1\_L or 1.0\_E10\_f.

It is significantly easier to read and enter long constants when they are broken up into smaller groupings (many cultures use comma and/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 and Java 8. C++ uses the single quote (') as a separator, restricted within a sequence of digits, e.g., 0xaaa'ff, 3.141'592E1'1.

7 Exponentiation Operator

C, C++, and Java (and other programming languages) have no exponentiation operator, i.e., \( x^i \), and instead use a routine, like pow(x,y), to perform the exponentiation operation. CV extends the basic operators with the exponentiation operator \(^i\) and \(\neq i\), as in, \( x \mid y \) and \( x \leftarrow x^i \). The priority of the exponentiation operator is between the cast and multiplicative operators, so that \( w \ast (\text{int})x \ast (\text{int})y \ast z \) is parenthesized as \((w \ast ((\text{int})x) \ast ((\text{int})y)) \ast z\).

There are exponentiation operators for integral and floating types, including the built-in complex types. Integral exponentiation is performed with repeated multiplication\(^4\) (or shifting if the exponent is 2). Overflow for a large exponent or negative exponent returns zero. Floating exponentiation is performed using logarithms, so the exponent cannot be negative.

The multiplication computation is \(O(\log x)\).

4Declarations in the do-while condition are not useful because they appear after the loop body.

9 Control Structures

CV identifies inconsistent, problematic, and missing control structures in C, and extends, modifies, and adds control structures to increase functionality and safety.

9.1 if / while Statement

The if/while expression allows declarations, similar to for declaration expression.\(^5\)

\[
\begin{align*}
\text{if ( int x = f(); ... } & \quad // x != 0 \\
\text{if ( int x = f(); y = g(); ... } & \quad // x != 0 \&\& y != 0 \\
\text{if ( int x = f(); y = g(); x < y ; ... } & \quad // relational expression \\
\text{if ( struct S ( int i; } x = \{ f() ; x.i < 4 \} } & \quad // relational expression \\
\text{while ( int x = f(); y = g(); ... } & \quad // x != 0 \&\& y != 0 \\
\text{while ( int x = f(); y = g(); x < y ; ... } & \quad // relational expression \\
\text{while ( struct S ( int i; } x = \{ f() ; x.i < 4 \} } & \quad // relational expression
\end{align*}
\]

Unless a relational expression is specified, each variable is compared not equal to 0, which is the standard semantics for the if/while expression, and the results are combined using the logical && operator. The scope of the declaration(s) is local to the if/while statement, i.e., in both then and else clauses for if, and loop body for while. C++ only provides a single declaration always compared != to 0.

\(^5\)The multiplication computation is \(O(\log x)\).
9.2 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 is not in the spirit of brevity normally associated with C. Therefore, the case clause is extended with a list of values.

```c
switch (i) {
  case 1, 3, 5: // odd values
  ...  
  case 2, 4, 6: // even values
  ...  
}
```

In addition, subranges are allowed to specify case values.

```c
switch (i) {
  case 1~5:       // 1, 2, 3, 4, 5
  ...  
  case 10~15:     // 10, 11, 12, 13, 14, 15
  ...  
}
```

Lists of subranges are also allowed.

```c
case 1~5, 12~21, 35~42:
```

9.3 switch Statement

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

1. By default, the end of a case clause 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:

```c
switch (i) {
  case 1:
  ...  
  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:

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

---

6 gcc has the same mechanism but awkward syntax, 2 ...42, because a space is required after a number, otherwise the period is a decimal point.

7 In this section, the term case clause refers to either a case or default clause.
This situation is better handled by a list of case values (see Section 9.2). While fall-through itself is not a problem, the problem occurs when fall-through is the default, as this semantics is unintuitive to many programmers and is different from most 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:

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

This usage branches into control structures, which is known to cause both comprehension and technical difficulties. The comprehension problem results 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 declaration and initialization of variables when blocks are not entered at the beginning. There are few arguments for this kind of control flow, and therefore, there is a strong impetus to eliminate it.

This C idiom is known as “Duff’s device” [10], from this example:

```c
register int n = (count + 7) / 8;
switch (count % 8) {
    case 0: do { // transfer into "if" statement
        ... 
        case 1:   // transfer into "if" statement
            ... 
    } 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. [10]
```

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. Most 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; 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 statements.

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

```c
switch (x) {
    case 1: case 3: case 5: // odd values
        // odd action
        break;
    case 2: case 4: case 6: // even values
        // even action
        break;
}
```
9.3 **switch** Statement

```
int y = 1;  // unreachable initialization
x = 7;  // unreachable code without label/branch
```

```statement
  case 0: ...
  ...
int z = 0;  // unreachable initialization, cannot appear after case
z = 2;
```

```statement
  case 1:
    x = z;  // without fall through, z is uninitialized
  ...
  case 5:
...  
  fallthru;  // explicit fall through
  case 7:
...  
  break  // explicit end of switch (redundant)
default:
  j = 3;
```

While the declaration of the local variable `y` is useful with a scope 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`, where both are problematic. As well, the declaration of `z` cannot occur after the case because a label can only be attached to a statement, and without a fall-through to case 3, `z` is uninitialized. The key observation is that the `switch` statement branches into a control structure, i.e., there are multiple entry points into its statement body.

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 put into perspective the C changes to the `switch`.

1. Eliminating default fall-through 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, the common placement of the default clause at the end of the case list, and the most common use of fall-through, i.e., a list of case clauses executing common code, e.g.:

```
choose ( i ) {
  case 1: case 2: case 3: ...
    // implicit end of switch (break)
  case 5:
...  
  fallthru;  // explicit fall through
  case 7:
...  
  break  // explicit end of switch (redundant)
default:
  j = 3;
}
```

Like the `switch` statement, the `choose` statement retains the fall-through semantics for a list of case clauses. An implicit `break` is applied only at the end of the statements following a case clause. An explicit `fallthru` is retained because it is a C-idiom most C programmers expect, and its absence might discourage programmers from using the `choose` statement. As well, allowing an explicit `break` from the `choose` is a carry over from the `switch` statement, and expected by C programmers.
2. Duff’s device is eliminated from both switch and choose statements, and only invalidates a small amount of very questionable code. Hence, the case clause must appear at the same nesting level as the switch/choose 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 needs to appear is locations other than at the end. Therefore, no change is made for this issue.

4. Dealing with unreachable code in a switch/choose body is solved by restricting declarations and initialization to the start of statement body, which is executed before the transfer to the appropriate case clause and precluding statements before the first case clause. Further declarations at the same nesting level as the statement body are disallowed to ensure every transfer into the body is sound.

```
switch (x) {
    int i = 0;          // allowed only at start
    case 0:
        ...
        int j = 0;       // disallowed
    case 1:
        {
            int k = 0;   // allowed at different nesting levels
            ...
            case 2:      // disallow case in nested statements
        }
    ...
}
```

9.4 Non-terminating and Labelled fallthrough

The fallthrough clause may be non-terminating within a case clause or have a target label to common code from multiple case clauses.

```
choose ( ... ) {
    case 3:          // implicit break
        if ( ... ) {
            ... fallthru; goto case 4
        } else {
            ...
            common: // below fallthrough // at case-clause level
        }
    case 4:          // implicit break // common code for cases 3/4
        ...
        common: // below fallthrough // at case-clause level
}
```

The target label must be below the fallthrough and may not be nested in a control structure, and the target label must be at the same or higher level as the containing case clause and located at the same level as a case clause; the target label may be case default, but only associated with the current switch/choose statement.

9.5 Loop Control

Looping a fixed number of times, possibly with a loop index, occurs frequently. C# condenses simply looping to facilitate coding speed and safety. The for, while, and do loop-control is augmented as follows (see examples in Figure 2):

- 0 is the implicit start value;
- 1 is the implicit increment value.

---

Essentially, these declarations are hoisted before the switch/choose statement and both declarations and statement are surrounded by a compound statement.
9.5 Loop Control

<table>
<thead>
<tr>
<th>loop control</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>while () { sout</td>
<td>&quot;empty&quot;: break; }</td>
</tr>
<tr>
<td>do { sout</td>
<td>&quot;empty&quot;: break; } while ();</td>
</tr>
<tr>
<td>for () { sout</td>
<td>&quot;empty&quot;: break; }</td>
</tr>
<tr>
<td>for (0) { sout</td>
<td>&quot;A&quot;; sout</td>
</tr>
<tr>
<td>for (1) { sout</td>
<td>&quot;A&quot;; }</td>
</tr>
<tr>
<td>for (10) { sout</td>
<td>&quot;A&quot;; }</td>
</tr>
<tr>
<td>for (10) { sout</td>
<td>&quot;B&quot;; }</td>
</tr>
<tr>
<td>for (10 ~&lt; 1 ~&lt; 2) { sout</td>
<td>&quot;C&quot;; }</td>
</tr>
<tr>
<td>for (0.5 ~&lt; 5.5) { sout</td>
<td>&quot;D&quot;; }</td>
</tr>
<tr>
<td>for (5.5 ~&lt; 0.5) { sout</td>
<td>&quot;E&quot;; }</td>
</tr>
<tr>
<td>for (i; 10) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; = 10) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 1 ~&lt; 10 ~&lt; 2) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 10 ~&lt; 1 ~&lt; 2) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 0.5 ~&lt; 5.5) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 5.5 ~&lt; 0.5) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (ui; 2u ~&lt; 10u ~&lt; 2u) { sout</td>
<td>ui; }</td>
</tr>
<tr>
<td>for (ui; 10u ~&lt; 10u ~&lt; 2u) { sout</td>
<td>ui; }</td>
</tr>
<tr>
<td>enum { N = 10 };</td>
<td></td>
</tr>
<tr>
<td>for (N) { sout</td>
<td>&quot;N&quot;; }</td>
</tr>
<tr>
<td>for (i; N ~&lt; 0) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>const int start = 3, comp = 10, inc = 2;</td>
<td></td>
</tr>
<tr>
<td>for (i; start ~ comp ~ inc + 1) { sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 1 ~&lt; @) { if (i &gt; 10) break; sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 10 ~&lt; @) { if (i &lt; 0) break; sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 2 ~&lt; @ ~&lt; 2) { if (i &gt; 10) break; sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 2.1 ~&lt; @ ~&lt; @) { if (i &gt; 10.5) break; sout</td>
<td>i; i += 1.7;}</td>
</tr>
<tr>
<td>for (i; 10.1 ~&lt; @ ~&lt; @) { if (i &gt; 0) break; sout</td>
<td>i; }</td>
</tr>
<tr>
<td>for (i; 12.1 ~&lt; @ ~&lt; @) { if (i &lt; 2.5) break; sout</td>
<td>i; i += 1.7;}</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; @) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; @) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
<tr>
<td>for (i; 5 ~&lt; j; ~&lt; 2) { sout</td>
<td>i</td>
</tr>
</tbody>
</table>
| Figure 2: Loop Control Examples

- The up-to range uses operator += for increment;
- The down-to range uses operator -= for decrement.
- The loop index is polymorphic in the type of the comparison value N (when the start value is implicit) or the start value M.
- An empty conditional implies comparison value of 1 (true).
- while () { sout | "empty": break; }
- do () { sout | "empty": break; } while ()
- for () { sout | "empty": break; }
- for (0) { sout | "A"; } sout | "zero"; 
- for (1) { sout | "A"; }
• A comparison $N$ is implicit up-to exclusive range $[0,N)$.

```c
for ( 5 ) // for ( typeof(5) i; i < 5; i += 1 )
```

• A comparison $= N$ is implicit up-to inclusive range $[0,N]$.

```c
for ( =5 ) // for ( typeof(5) i; i <= 5; i += 1 )
```

• The up-to range $M$ to $N$ means exclusive range $[M,N)$.

```c
for ( 1~5 ) // for ( typeof(1) i = 1; i < 5; i += 1 )
```

• The up-to range $M$ to $N$ means inclusive range $[M,N]$.

```c
for ( 1~=5 ) // for ( typeof(1) i = 1; i <= 5; i += 1 )
```

• The down-to range $M$ to $N$ means exclusive range $[N,M)$.

```c
for ( 1~ 5 ) // for ( typeof(1) i = 5; i > 0; i -= 1 )
```

• The down-to range $M$ to $N$ means inclusive range $[N,M]$.

```c
for ( 1~= 5 ) // for ( typeof(1) i = 5; i >= 0; i -= 1 )
```

• @ means put nothing in this field.

```c
for ( 1~ @~2 ) // for ( typeof(1) i = 1; /*empty*/; i += 2 )
```

• : means start another index.

```c
for ( i; 5 : j; 2~12~3 ) // for ( typeof(i) i = 1, j = 2; i < 5 & j < 12; i += 1, j += 3 )
```

### 9.6 Labelled continue / break Statement

C continue and break statements, for altering control flow, are restricted to one level of nesting for a particular control structure. This restriction forces programmers to use goto to achieve the equivalent control-flow for more than one level of nesting. To prevent having to switch to the goto, C4 extends the continue and break with a target label to support static multi-level exit [4], as in Java. For both continue and break, 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, if or compound (block) statement. Figure 3 shows a comparison between labelled continue and break and the corresponding C equivalent using goto and labels. The innermost loop has 8 exit points, which cause continuation or termination of one or more of the 7 nested control-structures.

Both labelled continue and break are a goto restricted in the following ways:

• They cannot create a loop, which means only the looping constructs cause looping. This restriction means all situations resulting in repeated execution are clearly delineated.

• They cannot branch into a control structure. This restriction prevents missing declarations and/or initializations at the start of a control structure resulting in undefined behaviour.

The advantage of the labelled continue/break is allowing static multi-level exits without having to use the goto statement, and tying control flow to the target control flow rather than an arbitrary point in a program via a label. Furthermore, the location of the label at the beginning of the target control structure informs the reader (eye candy) that complex control-flow is occurring in the body of the control structure. With goto, the label is at the end of the control structure, which 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 constructs to be added or removed without affecting existing constructs. Otherwise, the implicit targets of the current continue and break, i.e., the closest enclosing loop or switch, change as certain constructs are added or removed.

### 9.7 Extended else

The if statement has an optional else clause executed if the conditional is false. This concept is extended to the while, for, and do looping constructs (like Python). Hence, if the loop conditional becomes false, looping stops and the corresponding else clause is executed, if present.

The following example is a linear search for the key 3 in an array, where finding the key is handled with a break and not finding with the else clause on the loop construct.

```c
int a[10];
```
Figure 3: Multi-level Exit

```
while ( int i = 0; i < 10 ) {
    if ( a[i] == 3 ) break; // found
    i += 1;
} else { // i == 10
    sout | "not found";
}
```

```c
int i = 0;
for ( i; i < 10 ) {
    do {
        if ( a[i] == 3 ) break; // found
        i += 1;
    } else { // i == 10
        sout | "not found";
    }
}
```

Note, dangling else now occurs with if, while, for, do, and waitfor.

### 3.8 with Statement

Grouping heterogeneous data into an aggregate (structure/union) is a common programming practice, and aggregates may be nested:

```c
struct Person {
    // aggregate
    struct Name { char first[20], last[20]; } name
    // nesting
    struct Address { ... } address
    // nesting
    int sex;
};
```

Functions manipulating aggregates must repeat the aggregate name to access its containing fields.

```c
Person p
p.name; p.address; p.sex;
// access containing fields
```

which extends to multiple levels of qualification for nested aggregates and multiple aggregates.

```c
struct Ticket { ... } t;
```

Repeated aggregate qualification is tedious and makes code difficult to read. Therefore, reducing aggregate qualification is a useful language design goal.
C partially addresses the problem by eliminating qualification for enumerated types and unnamed *nested* aggregates, which open their scope into the containing aggregate. This feature is used to group fields for attributes and/or with union aggregates.

```c
struct S {
  struct /* unnamed */ {
    int g, h; /*attribute*/( aligned(64 ));
  } __attribute__((aligned(64)));
  int tag;
union /* unnamed */ {
  struct {
    char c1, c2; /*attribute*/( aligned(128 ));
  } __attribute__((aligned(128)));
  struct {
    int i1, i2;
  } __attribute__((aligned(128)));
  struct {
    double d1, d2;
  } __attribute__((aligned(128)));
};
enum { R, G, B };
s.g; s.h; s.tag = R; s.c1; s.c2; s.i1 = G; s.i2 = B; s.d1; s.d2;
}
```

Object-oriented languages reduce qualification for class variables within member functions, *e.g.*, C++:

```c
struct S {
  char c; int i; double d;
  void f( /* *S *this */ ) {
    // implicit "this" parameter
    c; i; d;
  } // with statement
}
```

In general, qualification is elided for the variables and functions in the lexical scopes visible from a member function. However, qualification is necessary for name shadowing and explicit aggregate parameters.

```c
void f( S & this ) with ( this ) {
  c; i; d;
} // with statement
```

Note the three different forms of qualification syntax in C++, *..* →*, *::*, which is confusing.

Since C in not object-oriented, it has no implicit parameter with its implicit qualification. Instead C introduces a general mechanism using the *with* statement (see Pascal [23, § 4.F]) to explicitly elide aggregate qualification by opening a scope containing the field identifiers. Hence, the qualified fields become variables with the side-effect that it is simpler to write, easier to read, and optimize field references in a block.

```c
void f( S & this ) with ( this ) {
  c; i; d;
} // with statement
```

with the generality of opening multiple aggregate-parameters:

```c
void g( S & s, T & t ) with ( s, t ) {
  c; s.i; d; // s.c, s.i, s.d
  m; t.i; n; // t.m, t.i, t.n
} // multiple aggregate parameters
```

where qualification is only necessary to disambiguate the shadowed variable i. In detail, the with statement may form a function body or be nested within a function body.

The with clause takes a list of expressions, where each expression provides an aggregate type and object. (Enumerations are already opened.) To open a pointer type, the pointer must be dereferenced to obtain a reference to the aggregate type.

```c
S * sp;
with ( *sp ) {
  ... }
```

The expression object is the implicit qualifier for the open structure-fields.
Exception Handling

CV’s ability to overload variables (see Section 25.2, p. 55) and use the left-side of assignment in type resolution means most fields with the same name but different types are automatically disambiguated, eliminating qualification.

All expressions in the expression list are open in parallel within the compound statement. This semantic is different from Pascal, which nests the openings from left to right. The difference between parallel and nesting occurs for fields with the same name and type:

```c
struct Q { int i; int k; int m; } q, w;
struct R { int i; int j; } r, w;
with ( r, q ) {
    j + k; // unambiguous, r.j + q.k
    m = 5.0; // unambiguous, q.m = 5.0
    m = 1; // unambiguous, r.m = 1
    int a = m; // unambiguous, a = r.i
    double b = m; // unambiguous, b = q.m
    int c = r.i + q.i; // disambiguate with qualification
    (double)m; // disambiguate with cast
}
```

For parallel semantics, both r.i and q.i are visible, so i is ambiguous without qualification; for nested semantics, q.i hides r.i, so i implies q.i. Pascal nested-semantics is possible by nesting with statements.

```c
with ( r ) {
    i; // unambiguous, r.i
}
with ( q ) {
    i; // unambiguous, q.i
}
```

A cast or qualification can be used to disambiguate variables within a with statement. A cast can also be used to disambiguate among overload variables in a with expression:

```c
with ( w ) { ... } // ambiguous, same name and no context
with ( Q w ) { ... } // unambiguous, cast
```

Because there is no left-side in the with expression to implicitly disambiguate between the w variables, it is necessary to explicitly disambiguate by casting w to type Q or R.

Finally, there is an interesting problem between parameters and the function-body with, e.g.:

```c
void f( S & s, char c ) with ( s ) {
    s.c = c; i = 3; d = 5.5; // initialize fields
}
```

Here, the assignment s.c = c means s.c = s.c, which is meaningless, and there is no mechanism to qualify the parameter c, making the assignment impossible using the function-body with. To solve this problem, parameters not explicitly opened are treated like an initialized aggregate:

```c
struct Params {
    char c;
} params;
```

and implicitly opened after a function-body open, to give them higher priority:

```c
void f( S & s, char c ) with ( s ) with( params ) { // syntax not allowed, illustration only
    s.c = c; i = 3; d = 5.5;
}
```

This implicit semantic matches with programmer expectation.

10 Exception Handling

Exception handling provides two mechanism: change of control flow from a raise to a handler, and communication from the raise to the handler. Transfer of control can be local, within a routine, or non-local, among routines. Non-local transfer can cause stack unwinding, i.e., non-local routine termination, depending on the kind of raise.

```c
exception_t E {}; // exception type
void f(...) {
    ... throw E(); ...
    // termination
```

10 Exception Handling
... throwResume E(); ... // resumption
try {
  f(...);
} catch( E e ; boolean-predicate ) {
    // recover and continue
    // termination handler
  ) catchResume( E e ; boolean-predicate ) {
    // resumption handler
    // repair and return
  ) finally {
    // always executed
}

The kind of raise and handler match: throw with catch and throwResume with catchResume. Then the exception type
must match along with any additional predicate must be true. The catch and catchResume handlers may appear in any
order. However, the finally clause must appear at the end of the try statement.

10.1 Exception Hierarchy
An exception type can be derived from another exception type, just like deriving a subclass from a class, providing
a kind of polymorphism among exception types. The exception-type hierarchy that is created is used to organize
exception types, similar to a class hierarchy in object-oriented languages, e.g.:

```
        Exception
          \   /       \   /
           IO File    Arithmetic
                         \   /
                          Network DivideByZero Overflow Underflow
```
A programmer can then choose to handle an exception at different degrees of specificity along the hierarchy; derived
exception-types support a more flexible programming style. For example, higher-level code should catch general
exceptions to reduce coupling to the specific implementation at the lower levels; unnecessary coupling may force
changes in higher-level code when low-level code changes. A consequence of derived exception-types is that multiple
exceptions may match, e.g.:

```
catch( Arithmetic )
```
matches all three derived exception-types: DivideByZero, Overflow, and Underflow. Because the propagation mecha-
nisms perform a simple linear search of the handler clause for a guarded block, and selects the first matching handler,
the order of catch clauses in the handler clause becomes important, e.g.:

```
try {
  ...
  ) catch( Overflow ) { // must appear first
    // handle overflow
  ) catch( Arithmetic )
    // handle other arithmetic issues
  )
```

Multiple derivation among exception is not supported.

11 Alternative 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[5]
```

Is this an array of 5 pointers to integers or a pointer to an array of 5 integers? If there is any doubt, it implies
productivity and safety issues even for basic programs. 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:

```c
int (*f())[5] ...;  // definition
... (*f())[3] += 1;   // 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 is a laudable goal, it has not worked out in practice, even though Dennis Richie
believed otherwise:

```quote
In spite of its difficulties, I believe that the C’s approach to declarations remains plausible, and am comfort-
able with it; it is a useful unifying principle. [27, p. 12]
```

C provides its own type, variable and routine declarations, using a different syntax. The new declarations place
qualifiers to the left of the base type, while C declarations place qualifiers to the right of the base type. In the following
element, red is the base type and blue is qualifiers. The C declarations move the qualifiers to the left of the base
type, i.e., move the blue to the left of the red, while the qualifiers have the same meaning but are ordered left to right
to specify a variable’s type.

```c
C
[5] * int x1;
* [5] int x2;
[• [5] int] f(int p);

C
int • x1[5];
int (*x2)[5];
int (*f(int p))[5];
```

The only exception is bit field specification, which always appear to the right of the base type. However, unlike C, C

type declaration tokens are distributed across all variables in the declaration list. For instance, variables x and y of type
pointer to integer are defined in C as follows:

```c
C
const * int x, y;
static * int x, * y;
```

The downside of this semantics is the need to separate regular and pointer declarations:

```c
C
int • x, y;
```

which is prescribing a safety benefit. Other examples are:

```c
C
[5] int z;                     // array of 5 integers
[5] char w;                  // array of 5 pointers to char
[5] double v;                // pointer to array of 5 doubles
struct s {
  int f0:3;
  int f1;
};
```

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

```c
C
const * int x;            // const pointer to const integer
const * [5] const int y;  // const pointer to array of 5 const integers
```

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

```c
C
extern [5] int x;         // externally visible array of 5 integers
static * const int y;    // internally visible pointer to const integer
```

The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. [21, § 6.11.5(1)]
The new declaration syntax can be used in other contexts where types are required, e.g., casts and the pseudo-
routine `sizeof`

\[
\begin{align*}
\text{C} & \quad \text{CV} \\
\text{y} &= (\star \text{int})x; & \quad \text{y} &= (\text{int} \star) x; \\
i &= \text{sizeof}(\[ 5 \] \star \text{int}); & \quad i &= \text{sizeof}(\text{int} \star [ 5 ]); \\
\end{align*}
\]

Finally, new CV declarations may appear together with C declarations in the same program block, but cannot be
mixed within a specific declaration. 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
C-style header-files, particularly for UNIX-like systems.

### 8 Pointer / Reference

C provides a pointer type; CV adds a reference type. These types may be derived from an object or routine type, called
the referenced type. Objects of these types contain an address, which is normally a location in memory, but may also
address memory-mapped registers in hardware devices. An integer constant expression with the value 0, or such an
expression cast to type `void *`, is called a null-pointer constant. An address is sound, if it points to a valid memory
location in scope, i.e., within the program’s execution environment and has not been freed. Dereferencing an unsound
address, including the null pointer, is undefined, often resulting in a memory fault.

A program object is a region of data storage in the execution environment, the contents of which can represent
values. In most cases, objects are located in memory at an address, and the variable name for an object is an implicit
address to the object generated by the compiler and automatically dereferenced, as in:

```
int x;  x = 3;  y = x;
int * y;  *y = x;
```

where the right example is how the compiler logically interprets the variables in the left example. Since a variable
name only points to one address during its lifetime, it is an immutable pointer; hence, the implicit type of pointer
variables \(x\) and \(y\) are constant pointers in the compiler interpretation. In general, variable addresses are stored in
instructions instead of loaded from memory, and hence may not occupy storage. These approaches are contrasted in
the following:

<table>
<thead>
<tr>
<th>explicit variable address</th>
<th>implicit variable address</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda ( r1,100 ) // load address of (x)</td>
<td>ld ( r2,(100) ) // load value of (x)</td>
</tr>
<tr>
<td>ld ( r2,(r1) ) // load value of (x)</td>
<td>st ( r2,(r3) ) // store (x) into (y)</td>
</tr>
</tbody>
</table>

Finally, the immutable nature of a variable’s address and the fact that there is no storage for the variable pointer means
pointer assignment is impossible. Therefore, the expression \(x = y\) has only one meaning, \(\star x = \star y\), i.e., manipulate
values, which is why explicitly writing the dereferences is unnecessary even though it occurs implicitly as part of
instruction decoding.

A pointer/reference object is a generalization of an object variable-name, i.e., a mutable address that can point to
more than one memory location during its lifetime. (Similarly, an integer variable can contain multiple integer literals
during its lifetime versus an integer constant representing a single literal during its lifetime, and like a variable name,
may not occupy storage if the literal is embedded directly into instructions.) Hence, a pointer occupies memory to
store its current address, and the pointer’s value is loaded by dereferencing, e.g.:

\[
\begin{align*}
\text{int } x, y, \& p1, \& p2, \& p3; \\
p1 &= \& x; \quad \text{// } p1 \text{ points to } x \\
p2 &= \& p1; \quad \text{// } p2 \text{ points to } x \\
p1 &= \& y; \quad \text{// } p1 \text{ points to } y \\
p3 &= \& p2; \quad \text{// } p3 \text{ points to } p2 \\
x &= 3; \quad y &= 3; \quad p1 &= 100; \\
p3 &= 112; \quad \text{int } * p3 &= 116; \\
\end{align*}
\]

One way to conceptualize the null pointer is that no variable is placed at this address, so the null-pointer address can be used to denote an uninitialized
pointer/reference object; i.e., the null pointer is guaranteed to compare unequal to a pointer to any object or routine. In general, a value with special
meaning among a set of values is called a sentinel value, e.g., \(-1\) as a return code value.
Notice, an address has a duality: a location in memory or the value at that location. In many cases, a compiler might be able to infer the best meaning for these two cases. For example, Algol68 [24] infers pointer dereferencing to select the best meaning for each pointer usage:

\[ p2 = p1 + x; \] // compiler infers \( *p2 = *p1 + x; \)

Algol68 infers the following dereferencing \( *p2 = *p1 + x, \) because adding the arbitrary integer value in \( x \) to the address of \( p1 \) and storing the resulting address into \( p2 \) is an unlikely operation. Unfortunately, automatic dereferencing does not work in all cases, and so some mechanism is necessary to fix incorrect choices. Rather than inferring dereference, most programming languages pick one implicit dereferencing semantics, and the programmer explicitly indicates the other to resolve address-duality. In C, objects of pointer type always manipulate the pointer object’s address:

\[ p1 = p2; \] // \( p1 = p2 \) rather than \( *p1 = *p2 \)
\[ p2 = p1 + x; \] // \( p2 = p1 + x \) rather than \( *p2 = *p1 + x \)

even though the assignment to \( p2 \) is likely incorrect, and the programmer probably meant:

\[ p1 = p2; \] // pointer address assignment
\[ *p2 = *p1 + x; \] // pointed-to value assignment / operation

The C semantics work well for situations where manipulation of addresses is the primary meaning and data is rarely accessed, such as storage management (malloc/free).

However, in most other situations, the pointed-to value is requested more often than the pointer address.

\[ \star p2 = (((\star p1 + \star p2) + (\star p3 - \star p1)) / (\star p3 - 15)); \]

In this case, it is tedious to explicitly write the dereferencing, and error prone when pointer arithmetic is allowed. It is better to have the compiler generate the dereferencing and have no implicit pointer arithmetic:

\[ p2 = ((p1 + p2) \cdot (p3 - p1)) / (p3 - 15); \]

To support this common case, a reference type is introduced in CV, denoted by \&\&, which is the opposite dereference semantics to a pointer type, making the value at the pointed-to location the implicit semantics for dereferencing (similar but not the same as C+ reference types).

**Exhibit A**

| &r1 = &x; | // \( \&r1 \) points to \( x \) |
| &r2 = &r1; | // \( \&r2 \) points to \( x \) |
| &r1 = &y; | // \( \&r1 \) points to \( y \) |
| &\&r3 = &\&r2; | // \( \&\&r3 \) points to \( r2 \) |
| r2 = ((r1 + r2) + (r3 - r1)) / (r3 - 15); | // implicit dereferencing |

Except for auto-dereferencing by the compiler, this reference example is the same as the previous pointer example. Hence, a reference behaves like the variable name for the current variable it is pointing-to. One way to conceptualize a reference is via a rewrite rule, where the compiler inserts a dereference operator before the reference variable for each reference qualifier in a declaration, so the previous example becomes:

\[ \star r2 = (((\star r1 + \star r2) + (\star r3 - \star r1)) / (\star r3 - 15)); \]

When a reference operation appears beside a dereference operation, e.g., \&\&, they cancel out. However, in C, the cancellation always yields a value (value).\(^{11}\) For a \&\&\& reference type, the cancellation on the left-hand side of assignment leaves the reference as an address (Ivalue):

\[ (\&\&\&) r1 = &x; \] // \( (\&\&\&) \) canceling giving address in \( r1 \) not variable pointed-to by \( r1 \)

Similarly, the address of a reference can be obtained for assignment or computation (value):

\[ (\&\&\&\&\&) r3 = (\&\&\&\&\&) r2; \] // \( (\&\&\&\&\&) \) cancel giving address in \( r2, (\&\&\&\&\&) \) cancel giving address in \( r3 \)

Cancellation works to arbitrary depth. Fundamentally, pointer and reference objects are functionally interchangeable because both contain addresses.

**Exhibit B**

| int x, \*p1 = &x, \*\*p2 = &p1, \*\*\*p3 = &p2, |
| &r1 = x, &\&r2 = r1, &\&\&r3 = r2; |

\[ \star \star \star p3 = 3; \] // change \( x \)
\[ r3 = 3; \] // change \( x, \star \star r3 \)

---

11\(^{11}\) The unary \& operator yields the address of its operand. If the operand has type "type", the result has type "pointer to type". If the operand is the result of a unary \* operator, neither that operator nor the \& operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an Ivalue. [21, § 6.5.3.2–3]
12.1 Initialization

Initialization is different than assignment because initialization occurs on the empty (uninitialized) storage on an object, while assignment occurs on possibly initialized storage of an object. There are three initialization contexts in C:\V: declaration initialization, argument/parameter binding, return/temporary binding. Because the object being initialized has no value, there is only one meaningful semantics with respect to address duality: it must mean address

The reason for disallowing arrays of reference is unknown, but possibly comes from references being ethereal (like a textual macro), and hence, replaceable by the referent object.
as there is no pointed-to value. In contrast, the left-hand side of assignment has an address that has a duality. Therefore, for pointer/reference initialization, the initializing value must be an address not a value.

```c
int * p = &x; // assign address of x
int * p = x; // assign value of x
int & r = x; // must have address of x
```

Like the previous example with C pointer-arithmetic, it is unlikely assigning the value of `x` into a pointer is meaningful (again, a warning is usually given). Therefore, for safety, this context requires an address, so it is superfluous to require explicitly taking the address of the initialization object, even though the type is incorrect. Note, this is strictly a convenience and safety feature for a programmer. Hence, `Cv` allows `r` to be assigned `x` because it infers a reference for `x`, by implicitly inserting a address-of operator, `&`, and it is an error to put an `&` because the types no longer match due to the implicit dereference. Unfortunately, `C` allows `p` to be assigned with `&x` (address) or `x` (value), but most compilers warn about the latter assignment as being potentially incorrect. Similarly, when a reference type is used for a parameter/return type, the call-site argument does not require a reference operator for the same reason.

```c
int & f( int & r ); // reference parameter and return
```

Within routine `f`, it is possible to change the argument by changing the corresponding parameter, and parameter `r` can be locally reassigned within `f`. Since operator routine `?+?` takes its arguments by value, the references returned from `f` are used to initialize compiler generated temporaries with value semantics that copy from the references.

```c
int temp1 = f( x ), temp2 = f( y );
```

This implicit referencing is crucial for reducing the syntactic burden for programmers when using references; otherwise references have the same syntactic burden as pointers in these contexts.

When a pointer/reference parameter has a `const` (immutable), it is possible to pass literals and expressions.

```c
void f( const int & cr );
void g( const int * cp );

f( 3 ); g( &3 );
f( x + y ); g( & (x + y) );
```

Here, the compiler passes the address to the literal `3` or the temporary for the expression `x + y`, knowing the argument cannot be changed through the parameter. The `&` before the constant/expression for the pointer-type parameter (`g`) is a `Cv` extension necessary to type match and is a common requirement before a variable in C (e.g., `scanf`). Importantly, `&3` may not be equal to `&3`, where the references occur across calls because the temporaries maybe different on each call.

`Cv extends` this semantics to a mutable pointer/reference parameter, and the compiler implicitly creates the necessary temporary (copying the argument), which is subsequently pointed-to by the reference parameter and can be changed.13

```c
void f( int & r );
void g( int * p );

f( 3 ); g( &3 ); // compiler implicit generates temporaries
f( x + y ); g( & (x + y) ); // compiler implicit generates temporaries
```

Essentially, there is an implicit value to lvalue conversion in this case.14 The implicit conversion allows seamless calls to any routine without having to explicitly name/copy the literal/expression to allow the call.

Finally, `C` handles routine objects in an inconsistent way. A routine object is both a pointer and a reference (particle and wave).

```c
void f( int i );
void (* fp)( int ); // routine pointer
fp = f; // reference initialization
fp = &f; // pointer initialization
fp = +f; // reference initialization
fp(3); // reference invocation
```

---

13If whole program analysis is possible, and shows the parameter is not assigned, i.e., it is `const`, the temporary is unnecessary.

14This conversion attempts to address the `const hell` problem, when the innocent addition of a `const` qualifier causes a cascade of type failures, requiring an unknown number of additional `const` qualifiers, until it is discovered a `const` qualifier cannot be added and all the `const` qualifiers must be removed.
While C’s treatment of routine objects has similarity to inferring a reference type in initialization contexts, the examples are assignment not initialization, and all possible forms of assignment are possible (f, &f, *f) without regard for type. Instead, a routine object should be referenced by a const reference:

```c
const void (&fr)(int) = f;  // routine reference
```

because the value of the routine object is a routine literal, i.e., the routine code is normally immutable during execution.\(^\text{15}\) CV allows this additional use of references for routine objects in an attempt to give a more consistent meaning for them.

### 12.2 Address-of Semantics

In C, &E is an rvalue for any expression E. CV extends the & (address-of) operator as follows:

- if R is an rvalue of type T &\(_1\) · · · \(_r\)_ & \(_s\), where \(r \geq 1\) references (\& symbols), then &R has type T \&\(_1\) · · · \(_r\)_ & \(_s\), i.e., T pointer with \(r - 1\) references (\& symbols).
- if L is an lvalue of type T \&\(_1\) · · · \(_l\)_ & \(_s\), where \(l \geq 0\) references (\& symbols), then &L has type T \&\(_1\) · · · \(_l\)_ & \(_s\), i.e., T pointer with \(l\) references (\& symbols).

The following example shows the first rule applied to different rvalue contexts:

```c
int x, *px, **ppx, ***pppx, ****ppppx;
int &rx = x, &&rrx = rx, &&&rrrrx = rrx;
x = rrrx;  // rrrx is an lvalue with type int && (equivalent to x)
px = &rrrx;  // starting from rrrx, &rrrx is an rvalue with type int *&& (rx)
pppx = &&rrrx;  // starting from &rrrx, &&rrrx is an rvalue with type int **&& (rrx)
```

The following example shows the second rule applied to different lvalue contexts:

```c
int x, *px, **ppx, ***pppx;
int &rx = x, &&rrx = rx, &&&rrrrx = rrx;
```

CV provides three new implicit conversion for reference types to simplify reference usage.

- integer to reference conversion: cv T \& converts to T, which allows implicit reference dereferencing.
- reference to integer conversion: cv T \& \& \& \& converts to cv int, which allows implicit variable dereferencing.

C provides a basic implicit conversion to simplify variable usage:

```
int x;
x++;  // lvalue variable (int) converts to rvalue for expression
```

An rvalue has no type qualifiers (cv), so the lvalue qualifiers are dropped.

CV provides three new implicit conversion for reference types to simplify reference usage.

- reference to integer conversion: cv T \& \& \& \& \& \& converts to cv int, which allows implicitly converting variables to references.

\(^\text{15}\) Dynamic code rewriting is possible but only in special circumstances.
13.1 Enum type

The type of enums can be any type, and an enum’s value comes from this type. Because an enum is a constant, it cannot appear in a mutable context, e.g., Mon = Sun is disallowed, and has no address (it is an rvalue). Therefore, an enum is automatically converted to its constant’s base-type, e.g., comparing/printing an enum compares/prints its value rather than the enum name; there is no mechanism to print the enum name.

The default enum type is int. Hence, Days is the set type Mon, Tue, ..., Sun, while the type of each enum is int and each enum represents a fixed integral value. If no values are specified for an integral enum type, the enums are automatically numbered by one from left to right starting at zero. Hence, the value of enum Mon is 0, Tue is 1, ..., Sun is 6. If an enum value is specified, numbering continues by one from that value for subsequent unnumbered enums. If an enum value is an expression, the compiler performs constant-folding to obtain a constant value.

CV allows other integral types with associated values.

```
enum( long long int ) BigNum { X = 123_456_789_012_345, Y = 345_012_789_456_123 };
```
For enumeration `Letter`, enum `A`'s value is explicitly set to 'A', with `B` and `C` implicitly numbered with increasing values from 'A', and similarly for enums `I`, `J`, and `K`.

Non-integral enum types must be explicitly initialized, e.g., `double` is not automatically numbered by one.

```c
// non-integral numeric
enum( double ) Math { PI_2 = 1.570796, PI = 3.141597, E = 2.718282 }
// pointer
enum( char * ) Name { Fred = "Fred", Mary = "Mary", Jane = "Jane" }
int i, j, k;
enum( int * ) ptr { l = &i, j = &j, K = &k }
enum( int & ) ref { l = i, J = j, K = k }
// tuple
enum( [int, int] ) { T = [ 1, 2 ] }
// function
void () ... void g() {...}
enum( void () ) funs { F = f, F = g }
// aggregate
struct S { int i, j; }
enum( S ) S { A = { 3, 4 }, B = { 7, 8 } }
// enumeration
enum( Letter ) Greek { Alph = A, Beta = B, /* more enums */ }; // alphabet intersection
```

 Enumeration `Greek` may have more or less enums than `Letter`, but the enum values must be from `Letter`. Therefore, `Greek` enums are a subset of type `Letter` and are type compatible with enumeration `Letter`, but `Letter` enums are not type compatible with enumeration `Greek`.

The following examples illustrate the difference between the enumeration type and the type of its enums.

```c
Math m = PI; // allowed
double d = PI; // allowed, conversion to base type
m = E; // allowed
m = Alph; // disallowed
m = 3.141597; // disallowed
d = m; // allowed
d = Alph; // disallowed
Letter l = A; // allowed
Greek g = Alph; // allowed
l = Alph; // allowed, conversion to base type
g = A; // disallowed
```

A constructor cannot be used to initialize enums because a constructor executes at runtime. A fallback is explicit C-style initialization using `@=`.

```c
enum( struct vec3 ) Axis { Up @= { 1, 0, 0 }, Left @= { 0, 1, 0 }, Front @= { 0, 0, 1 } }
```

Finally, enumeration variables are assignable and comparable only if the appropriate operators are defined for its enum type.

### 13.2 Inheritance

Plan-9 inheritance may be used with enumerations.

```c
enum( char * ) Name2 { inline Name, Jack = "Jack", Jill = "Jill" }
enum /* inferred */ Name3 { inline Name2, Sue = "Sue", Tom = "Tom" }
```

Enumeration `Name2` inherits all the enums and their values from enumeration `Name` by containment, and a Name enumeration is a subtype of enumeration `Name2`. Note, enums must be unique in inheritance but enum values may be repeated. The enum type for the inheriting type must be the same as the inherited type; hence the enum type may be omitted for the inheriting enumeration and it is inferred from the inherited enumeration, as for `Name3`. When inheriting from integral types, automatic numbering may be used, so the inheritance placement left to right is important, e.g., the placement of Sue and Tom before or after inline `Name2`.

Specifically, the inheritance relationship for `Names` is:

```
Name ⊆ Name2 ⊆ Name3 ⊆ const char * // enum type of Name
```

Hence, given
void f( Name );
void g( Name2 );
void h( Name3 );
void j( const char * );

the following calls are valid
f( Fred );
g( Fred ); g( Jill );
h( Fred ); h( Jill ); h( Sue );
j( Fred ); j( Jill ); j( Sue ); j( 'W' );

Note, the validity of calls is the same for call-by-reference as for call-by-value, and const restrictions are the same as for other types.

Enums cannot be created at runtime, so inheritance problems, such as contra-variance do not apply. Only instances of the enum base-type may be created at runtime.

14 Routine Definition

C supports a new syntax for routine definition, as well as C11 and K&R routine syntax. The point of the new syntax is to allow returning multiple values from a routine [16, 25], 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 type. The value of each local return variable is automatically returned at routine termination. Declaration qualifiers can only appear at the start of a routine definition, 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 return list and discussed in Section 12.

C-style declarations cannot be used to declare parameters for K&R style routine definitions because of the following ambiguity:
```
int (*f(x))[ 5 ] int x; {}
```
The string "int (*f(x))[ 5 ]" declares a K&R style routine of type returning a pointer to an array of 5 integers, while the string "[ 5 ] int x" declares a C-style parameter x of type array of 5 integers. Since the strings overlap starting with the open bracket, [ , there is an ambiguous interpretation for the string. As well, C-style declarations cannot be used to declare parameters for C-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 a C-style named-parameter of type pointer to an integer (the parenthesis are superfluous), while the same string declares a C-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 relies on all type qualifier characters appearing to the right of the type name. The inability to use C declarations in these two contexts is probably a blessing because it precludes programmers from arbitrarily switching between declarations forms within a declaration contexts.

16 Michael Tiemann, with help from Doug Lea, provided named return values in g++, circa 1989.
C-style declarations can be used to declare parameters for C-style routine definitions, e.g.:

```c
[ 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 preprocessor macros that generate C-style declaration-syntax, as in:

```c
#define ptoa( n, d ) int(*n)[ d ]
```

```c
int f( ptoa( p, 5 ) ... // expands to int f( int (*p)[ 5 ] )
[ int ] f( ptoa( p, 5 ) ... // expands to [ int ] f( int (*p)[ 5 ] )
```

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

### 14.1 Named Return 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:

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

Because the value in the return variable is automatically returned when a C-style routine terminates, the `return` statement does not contain an expression, as in:

```c
[ int x, int y ] f() {
    int z;
    ... x = 0; ... y = z; ...
    return; // implicitly return x, y
}
```

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

```c
[ int x, int y ] f() {
    ... // implicitly return x, y
}
```

In this case, the current values of `x` and `y` are returned to the calling routine just as if a `return` had been encountered. Named return values may be used in conjunction with named parameter values; specifically, a return and parameter can have the same name.

```c
[ int x, int y ] f( ( int, x, int y ) {
    ... // implicitly return x, y
}
```

This notation allows the compiler to eliminate temporary variables in nested routine calls.

```c
[ int x, int y ] f( int, x, int y ); // prototype declaration
int a, b;
[a, b] = f( f( f( a, b ) ) );
```

While the compiler normally ignores parameters names in prototype declarations, here they are used to eliminate temporary return-values by inferring that the results of each call are the inputs of the next call, and ultimately, the left-hand side of the assignment. Hence, even without the body of routine `f` (separate compilation), it is possible to perform a global optimization across routine calls. The compiler warns about naming inconsistencies between routine prototype and definition in this case, and behaviour is undefined if the programmer is inconsistent.

### 14.2 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.:

```c
[ 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
```
15 Routine Pointers

15.1 Syntax and Usage

- **Prototype declarations:**
  - [• int, int] j (int);
    // returning pointer to int and int, with int parameter

- **Declaration qualifiers:**
  - extern [• int] f (int);
  - static [• int] g (int);

- **C functions:**
  - const double bar1(), bar2(int), bar3(double);

- **CA functions:**
  - [const double] foo(), foo(int), foo(double) { return 3.0; }

- **Example declaration:**
  - CA allows the last routine in the list to define its body.

- **Function pointers:**
  - *[• 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

- **Named arguments:**
  - 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) {...}

- **Positional call:**
  - p(4, 7, 3);

- **Named (keyword) call:**
  - p(z:3, x:4, y:7);
    // rewrite ⇒ p(4, 7, 3)

- **Advantages of named parameters:**
  - 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).

16 Named and Default Arguments

- **Named and default arguments** are two mechanisms to simplify routine call. Both mechanisms are discussed with respect to CA.

- **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) {...}

- **Positional call:**
  - p(4, 7, 3);

- **Named (keyword) call:**
  - p(4, 7, 3);
    // rewrite ⇒ p(z:3, x:4, y:7)

- **Here the order of the arguments is unimportant, and the names of the parameters are used to associate argument values with the corresponding parameters. The compiler rewrites 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).

- **Unfortunately, named arguments do not work in C-style programming-languages because a routine prototype is not required to specify parameter names, nor do the names in the prototype have to match with the actual definition.**

- **For example, the following routine prototypes and definition are all valid:**
  - void p(int, int, int);
    // equivalent prototypes
  - void p(int x, int y, int z);
  - void p(int y, int x, int z);
  - void p(int z, int y, int x);
  - void p(int q, int r, int s) {} // match with this definition

17 Francez [15] 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.
Forcing matching parameter names in routine prototypes with corresponding routine definitions is possible, but goes against a strong tradition in C programming. Alternatively, prototype definitions can be eliminated by using a two-pass compilation, and implicitly creating header files for exports. The former is easy to do, while the latter is more complex.

Furthermore, named arguments do not work well in a C-style programming languages because they potentially introduces a new criteria for type matching. For example, it is technically possible to disambiguate between these two overloaded definitions of \( f \) based on named arguments at the call site:

\[
\begin{align*}
\text{int } f(\text{ int } i, \text{ int } j); \\
\text{int } f(\text{ int } x, \text{ double } y);
\end{align*}
\]

\[
\begin{align*}
f( j : 3, i : 4 ); & \quad \text{// 1st } f \\
f( x : 7, y : 8.1 ); & \quad \text{// 2nd } f \\
f( 4, 5 ); & \quad \text{// ambiguous call}
\end{align*}
\]

However, named arguments compound routine resolution in conjunction with conversions:

\[
\begin{align*}
f( i : 3, 5.7 ); & \quad \text{// ambiguous call ?}
\end{align*}
\]

Depending on the cost associated with named arguments, this call could be resolvable or ambiguous. Adding named argument into the routine resolution algorithm does not seem worth the complexity. Therefore, C\text{-}\textsf{\textcopyright} does not attempt to support named arguments.

**Default Arguments** provide 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:

\[
\begin{align*}
\text{void } p(\text{ int } x = 1, \text{ int } y = 2, \text{ int } z = 3 ) {...}
\end{align*}
\]

the allowable positional calls are:

\[
\begin{align*}
p(); & \quad \text{// rewrite } \Rightarrow p( 1, 2, 3 ) \\
p( 4 ); & \quad \text{// rewrite } \Rightarrow p( 4, 2, 3 ) \\
p( 4, 4 ); & \quad \text{// rewrite } \Rightarrow p( 4, 4, 3 ) \\
p( 4, 4, 4 ); & \quad \text{// rewrite } \Rightarrow p( 4, 4, 4 ) \\
\end{align*}
\]

\[
\begin{align*}
\text{// empty arguments}
\end{align*}
\]

\[
\begin{align*}
p(\ , 4, 4 ); & \quad \text{// rewrite } \Rightarrow p( 1, 4, 4 ) \\
p( 4,\ , 4 ); & \quad \text{// rewrite } \Rightarrow p( 4, 2, 4 ) \\
p( 4, 4,\ ); & \quad \text{// rewrite } \Rightarrow p( 4, 4, 3 ) \\
p( 4,\ ,\ ); & \quad \text{// rewrite } \Rightarrow p( 4, 2, 3 ) \\
p(\ , 4,\ ); & \quad \text{// rewrite } \Rightarrow p( 1, 4, 3 ) \\
p(\ ,\ , 4 ); & \quad \text{// rewrite } \Rightarrow p( 1, 2, 4 ) \\
p(\ ,\ ,\ ); & \quad \text{// rewrite } \Rightarrow p( 1, 2, 3 )
\end{align*}
\]

Here the missing arguments are inserted from the default values in the parameter list. The compiler 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 that are error prone.
- When a routine’s interface is augmented with new parameters, it extends the interface providing generalizability (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.

---

18“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.” [8, p. 128]
Unnamed Structure Fields

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

```c
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.

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

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

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

```c
p(int x, int y, int z, ...);
p(1, z:3, y:2);
```

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.:

```c
void p(int x, int y = 2, int z = 3...);
p(1, z:3, y:2);
```

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 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 type system largely eliminates the need for ellipse arguments (see Section 25, p. 53), making much of this discussion moot.

Default arguments and overloading (see Section 25, p. 53) are complementary. While in theory default arguments can be simulated with overloading, as in:

```c
default arguments overloading
void p(int x, int y = 2, int z = 3) {...} // assume p( /*positional*/..., /*named*/...);
void p(int x, int y, int z = 3); // assume p( /*positional*/..., /*named*/...);
```

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:

```c
p(1, /*default*/5); // rewrite ⇒ p(1, 2, 5)
```

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

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

```c
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
}```
18 Nesting

Nesting of types and routines is useful for controlling name visibility (name hiding).

18.1 Type Nesting

C\textit{\textbf{V}} allows type nesting, and type qualification of the nested types (see Figure 4), where as C hoists (refactors) nested types into the enclosing scope and has no type qualification. In the left example in C, types C, U and T are implicitly hoisted outside of type S into the containing block scope. In the right example in C\textit{\textbf{V}}, the types are not hoisted and accessed using the field-selection operator "." for type qualification, as does Java, rather than the C++ type-selection operator ":.".

18.2 Routine Nesting

While C\textit{\textbf{V}} does not provide object programming by putting routines into structures, it does rely heavily on locally nested routines to redefine operations at or close to a call site. For example, the C quick-sort is wrapped into the following polymorphic C\textit{\textbf{V}} routine:

```c
forall( T | { int ?<?( T, T ); } )
void qsort( const T * arr, size_t dimension );
```

which can be used to sort in ascending and descending order by locally redefining the less-than operator into greater-than.

```c
const unsigned int size = 5;
int ia[size];
... // assign values to array ia
qsort( ia, size ); // sort ascending order using builtin ?<?
{
    int ?<?( int x, int y ) { return x > y; } // nested routine
    qsort( ia, size ); // sort descending order by local redefinition
```
Nested routines are not first-class, meaning a nested routine cannot be returned if it has references to variables in its enclosing blocks; the only exception is references to the external block of the translation unit, as these variables persist for the duration of the program. The following program in undefined in C and C++ (and Indexcgcc)

```c
[- [int](int) foo() {
    int i = 7;
    int bar(int p) {
        i += 1;
        // dependent on local variable
        sout | i;
    }
    return bar;
}
```

because

Currently, there are no lambda expressions, i.e., unnamed routines because routine names are very important to properly select the correct routine.

In C and C++, lists of elements appear in several contexts, such as the parameter list of a routine call.

```c
({ 2, x, 3 + i }); // element list
```

A list of elements is called a tuple, and is different from a comma expression.

19.1 Multiple-Return-Value Functions

In C and most programming languages, functions return at most one value; however, many operations have multiple outcomes, some exceptional (see Section 10, p. 15). To emulate functions with multiple return values, aggregation and/or aliasing is used.

In the former approach, a record type is created combining all of the return values. For example, consider C’s `div` function, which returns the quotient and remainder for a division of an integer value.

```c
typedef struct { int quot, rem; } div_t; // from include stdlib.h
div_t div(int num, int den);
div_t qr = div(13, 5); // return quotient/remainder aggregate
printf("%d \%d\n", qr.quot, qr.rem); // print quotient/remainder
```

This approach requires a name for the return type and fields, where naming is a common programming-language issue. That is, naming creates an association that must be managed when reading and writing code. While effective when used sparingly, this approach does not scale when functions need to return multiple combinations of types.

In the latter approach, additional return values are passed as pointer parameters. A pointer parameter is assigned inside the routine to emulate a return. For example, consider C’s `modf` function, which returns the integral and fractional part of a floating value.

```c
double modf(double x, double * i); // from include math.h
double intp, frac = modf(13.5, &intp); // return integral and fractional components
printf("%g \%g\n", intp, frac); // print integral/fractional components
```

This approach requires allocating storage for the return values, which complicates the call site with a sequence of variable declarations leading to the call. Also, while a disciplined use of `const` can give clues about whether a pointer parameter is used as an out parameter, it is not obvious from the routine signature whether the callee expects such a parameter to be initialized before the call. Furthermore, while many C routines that accept pointers are safe for a `NULL` argument, there are many C routines that are not null-safe. Finally, C does not provide a mechanism to state that a parameter is going to be used as an additional return value, which makes the job of ensuring that a value is returned more difficult for the compiler. Still, not every routine with multiple return values should be required to return an
error code, and error codes are easily ignored, so this is not a satisfying solution. As with the previous approach, this
technique can simulate multiple return values, but in practice it is verbose and error prone.

C\textit{\textゥ}} allows functions to return multiple values by extending the function declaration syntax. Multiple return values
are declared as a comma-separated list of types in square brackets in the same location that the return type appears in
standard C function declarations.

\begin{verbatim}
[ char, int, double ] f( ... );
\end{verbatim}

The ability to return multiple values from a function requires a new syntax for the return statement. For consistency,
the return statement in C\textit{\textゥ}} accepts a comma-separated list of expressions in square brackets.

\begin{verbatim}
return [ c, i, d ];
\end{verbatim}

The expression resolution ensures the correct form is used depending on the values being returned and the return type
of the current function. A multiple-returning function with return type \texttt{T} can return any expression that is implicitly
convertible to \texttt{T}.

A common use of a function’s output is input to another function. C\textit{\textゥ}} allows this case, without any new syntax; a
multiple-returning function can be used in any of the contexts where an expression is allowed. When a function call is
passed as an argument to another call, the best match of actual arguments to formal parameters is evaluated given all
possible expression interpretations in the current scope.

\begin{verbatim}
void g( int, int );           // 1
void g( double, double );     // 2
g( div( 13, 5 ) );            // select 1
\end{verbatim}

In this case, there are two overloaded \texttt{g} routines. Both calls to \texttt{g} expect two arguments that are matched by the two
return values from \texttt{div} and \texttt{modf}, respectively, which are fed directly to the first and second parameters of \texttt{g}. As well,
both calls to \texttt{g} have exact type matches for the two different versions of \texttt{g}, so these exact matches are chosen. When
type matches are not exact, conversions are used to find a best match.

The previous examples can be rewritten passing the multiple returned-values directly to the \texttt{printf} function call.

\begin{verbatim}
[ int, int ] div( int x, int y );   // from include stdlib
printf( "%d %d\n", div( 13, 5 ) );  // print quotient/remainder
\end{verbatim}

This approach provides the benefits of compile-time checking for appropriate return statements as in aggregation, but
without the required verbosity of declaring a new named type.

Finally, the addition of multiple-return-value functions necessitates a syntax for retaining the multiple values at the
call-site versus their temporary existence during a call. The simplest mechanism for retaining a return value in C is
variable assignment. By assigning the multiple return-values into multiple variables, the values can be retrieved later.
As such, C\textit{\textゥ}} allows assigning multiple values from a function into multiple variables, using a square-bracketed list of
value expressions on the left side.

\begin{verbatim}
int quot, rem;
[ quot, rem ] = div( 13, 5 );      // assign multiple variables
\end{verbatim}

Here, the multiple return-values are matched in much the same way as passing multiple return-values to multiple
parameters in a call.

\section{Expressions}

Multiple-return-value functions provide C\textit{\textゥ}} with a new syntax for expressing a combination of expressions in the
return statement and a combination of types in a function signature. These notions are generalized to provide C\textit{\textゥ}} with
tuple expressions and tuple types. A tuple expression is an expression producing a fixed-size, ordered list of values of
heterogeneous types. The type of a tuple expression is the tuple of the subexpression types, or a tuple type.

In C\textit{\textゥ}}, a tuple expression is denoted by a comma-separated list of expressions enclosed in square brackets. For
example, the expression \texttt{[5, ‘x’, 10.5]} has type \texttt{[int, char, double]}. The previous expression has 3 \textit{components}. Each
component in a tuple expression can be any C\textit{\textゥ}} expression, including another tuple expression. The order of evalu-
ation of the components in a tuple expression is unspecified, to allow a compiler the greatest flexibility for program
19.3 Variables

It is, however, guaranteed that each component of a tuple expression is evaluated for side-effects, even if the result is not used. Multiple-return-value functions can equivalently be called \textit{tuple-returning functions}.

The previous call of \texttt{div} still requires the preallocation of multiple return-variables in a manner similar to the aliasing example. In C\textsc{\small V}, it is possible to overcome this restriction by declaring a \textit{tuple variable}.

\begin{verbatim}
[ int, int ] qr = div( 13, 5 ); // initialize tuple variable
printf( "%d %d\n", qr ); // print quotient/remainder
\end{verbatim}

It is now possible to match the multiple return-values to a single variable, in much the same way as aggregation. As well, the components of the tuple value are passed as separate parameters to \texttt{printf}, allowing direct printing of tuple variables. One way to access the individual components of a tuple variable is with assignment.

\begin{verbatim}
[ quot, rem ] = qr; // assign multiple variables
\end{verbatim}

In addition to variables of tuple type, it is also possible to have pointers to tuples, and arrays of tuples. Tuple types can be composed of any types, except for array types, since array assignment is disallowed, which makes tuple assignment difficult when a tuple contains an array.

\begin{verbatim}
[ double, int ] di;
[ double, int ] * pdi
[ double, int ] adi[10];
\end{verbatim}

This examples declares a variable of type \texttt{[double, int]}, a variable of type pointer to \texttt{[double, int]}, and an array of ten \texttt{[double, int]}.

19.4 Indexing

It is also possible to access a single component of a tuple-valued expression without creating temporary variables. Given a tuple-valued expression \texttt{enp} and a compile-time constant integer \texttt{i} where \(0 \leq i < n\), where \(n\) is the number of components in \texttt{enp}, \texttt{e.\texttt{i}} accesses the \texttt{i}th component of \texttt{enp}, e.g.:

\begin{verbatim}
[ int, double ] x;
[ char *, int ] f();
void \texttt{g}\texttt{(double, int)};
[ int, double ] * \texttt{p};
\end{verbatim}

\begin{verbatim}
int \texttt{y} = x.0; // access int component of \texttt{x}
y = f().1; // access int component of \texttt{f}
p->0 = 5; // access int component of tuple pointed-to by \texttt{p}
g( x.1, x.0 ); // rearrange \texttt{x} to pass to \texttt{g}
double \texttt{z} = [ x, f() ].0.1; // access second component of first component of tuple expression
\end{verbatim}

Tuple-index expressions can occur on any tuple-typed expression, including tuple-returning functions, square-bracketed tuple expressions, and other tuple-index expressions, provided the retrieved component is also a tuple. This feature was proposed for K-W C but never implemented [31, p. 45].

19.5 Flattening and Structuring

As evident in previous examples, tuples in C\textsc{\small V} do not have a rigid structure. In function call contexts, tuples support implicit flattening and restructuring conversions. Tuple flattening recursively expands a tuple into the list of its basic components. Tuple structuring packages a list of expressions into a value of tuple type.

\begin{verbatim}
int \texttt{f}\texttt{(int, int)};
int \texttt{g}\texttt{([int, int])};
int \texttt{h}\texttt{(int, [int, int])};
[ int, int ] x;
int \texttt{y};
f(\texttt{x}); // flatten
\texttt{g(\texttt{y, 10})}; // structure
\texttt{h(\texttt{x, y})}; // flatten & structure
\end{verbatim}
In C\(\text{V}\), each of these calls is valid. In the call to \(f\), \(x\) is implicitly flattened so that the components of \(x\) are passed as the two arguments to \(f\). For the call to \(g\), the values \(y\) and 10 are structured into a single argument of type \([\text{int, int}]\) to match the type of the parameter of \(g\). Finally, in the call to \(h\), \(x\) is flattened to yield an argument list of length 3, of which the first component of \(x\) is passed as the first parameter of \(h\), and the second component of \(x\) and \(y\) are structured into the second argument of type \([\text{int, int}]\). The flexible structure of tuples permits a simple and expressive function-call syntax to work seamlessly with both single- and multiple-return-value functions, and with any number of arguments of arbitrarily complex structure.

In K-W C [5, 31], there were 4 tuple coercions: opening, closing, flattening, and structuring. Opening coerces a tuple value into a tuple of values, while closing converts a tuple of values into a single tuple value. Flattening coerces a nested tuple into a flat tuple, \(i.e.,\) it takes a tuple with tuple components and expands it into a tuple with only non-tuple components. Structuring moves in the opposite direction, \(i.e.,\) it takes a flat tuple value and provides structure by introducing nested tuple components.

In C\(\text{V}\), the design has been simplified to require only the two conversions previously described, which trigger only in function call and return situations. This simplification is a primary contribution of this thesis to the design of tuples in C\(\text{V}\). Specifically, the expression resolution algorithm examines all of the possible alternatives for an expression to determine the best match. In resolving a function call expression, each combination of function value and list of argument alternatives is examined. Given a particular argument list and function value, the list of argument alternatives is flattened to produce a list of non-tuple valued expressions. Then the flattened list of expressions is compared with each value in the function’s parameter list. If the parameter’s type is not a tuple type, then the current argument value is unified with the parameter type, and on success the next argument and parameter are examined. If the parameter’s type is a tuple type, then the structuring conversion takes effect, recursively applying the parameter matching algorithm using the tuple’s component types as the parameter list types. Assuming a successful unification, eventually the algorithm gets to the end of the tuple type, which causes all of the matching expressions to be consumed and structured into a tuple expression. For example, in

\[
\text{int } f(\text{int, [double, int]});
\]

\[
f([5, 10.2], 4);
\]

There is only a single definition of \(f\), and 3 arguments with only single interpretations. First, the argument alternative list \([5, 10.2], 4\) is flattened to produce the argument list \(5, 10.2, 4\). Next, the parameter matching algorithm begins, with \(P = \text{int}\) and \(A = \text{int}\), which unifies exactly. Moving to the next parameter and argument, \(P = \text{[double, int]}\) and \(A = \text{double}\). This time, the parameter is a tuple type, so the algorithm applies recursively with \(P' = \text{double}\) and \(A = \text{double}\), which unifies exactly. Then \(P' = \text{int}\) and \(A = \text{double}\), which again unifies exactly. At this point, the end of \(P'\) has been reached, so the arguments \(10.2, 4\) are structured into the tuple expression \([10.2, 4]\). Finally, the end of the parameter list \(P\) has also been reached, so the final expression is \(f(5, [10.2, 4])\).

### 19.6 Assignment

An assignment where the left side of the assignment operator has a tuple type is called \textit{tuple assignment}. There are two kinds of tuple assignment depending on whether the right side of the assignment operator has a non-tuple or tuple type, called \textit{mass} and \textit{multiple} assignment, respectively.

\[
\text{int } x;
\]
\[
\text{double } y;
\]
\[
\text{[int, double]} z;
\]
\[
[y, x] = 3.14;
\] // mass assignment
\[
[x, y] = z;
\] // multiple assignment
\[
z = 10;
\] // mass assignment
\[
z = [x, y];
\] // multiple assignment

Let \(L_i\) for \(i\) in \([0, n]\) represent each component of the flattened left side, \(R_i\) represent each component of the flattened right side of a multiple assignment, and \(R\) represent the right side of a mass assignment.

For a multiple assignment to be valid, both tuples must have the same number of elements when flattened. For example, the following is invalid because the number of components on the left does not match the number of components on the right.

\[
\text{[int, int]} x, y, z;
\]
\[
[x, y] = z;
\] // multiple assignment, invalid \(4 != 2\)
Multiple assignment assigns $R_i$ to $L_i$ for each $i$. That is, $\&\$S\_L\_S\_R\_S$ must be a well-typed expression. In the previous example, $[x, y] = z$, $z$ is flattened into $z.0$, $z.1$, and the assignments $x = z.0$ and $y = z.1$ happen.

A mass assignment assigns the value $R$ to each $L_i$. For a mass assignment to be valid, $\&\$S\_L\_S\_R\_S$ must be a well-typed expression. These semantics differ from C cascading assignment (e.g., $a=b=c$) in that conversions are applied to $R$ in each individual assignment, which prevents data loss from the chain of conversions that can happen during a cascading assignment. For example, $[y, x] = 3.14$ performs the assignments $y = 3.14$ and $x = 3.14$, which results in the value $3.14$ in $y$ and the value $3$ in $x$. On the other hand, the C cascading assignment $y = x = 3.14$ performs the assignments $x = 3.14$ and $y = x$, which results in the value $3$ in $x$, and as a result the value $3$ in $y$ as well.

Both kinds of tuple assignment have parallel semantics, such that each value on the left side and right side is evaluated before any assignments occur. As a result, it is possible to swap the values in two variables without explicitly creating any temporary variables or calling a function.

```c
int x = 10, y = 20;
[x, y] = [y, x];
```

After executing this code, $x$ has the value 20 and $y$ has the value 10.

In CV, tuple assignment is an expression where the result type is the type of the left side of the assignment, as in normal assignment. That is, a tuple assignment produces the value of the left-hand side after assignment. These semantics allow cascading tuple assignment to work out naturally in any context where a tuple is permitted. These semantics are a change from the original tuple design in K-W C [31], wherein tuple assignment was a statement that allows cascading assignments as a special case. Restricting tuple assignment to statements was an attempt to fix what was seen as a problem with side-effects, wherein assignment can be used in many different locations, such as in function-call argument position. While permitting assignment as an expression does introduce the potential for subtle complexities, it is impossible to remove assignment expressions from CV without affecting backwards compatibility. Furthermore, there are situations where permitting assignment as an expression improves readability by keeping code succinct and reducing repetition, and complicating the definition of tuple assignment puts a greater cognitive burden on the user. In another language, tuple assignment as a statement could be reasonable, but it would be inconsistent for tuple assignment to be the only kind of assignment that is not an expression. In addition, K-W C permits the compiler to optimize tuple assignment as a block copy, since it does not support user-defined assignment operators. This optimization could be implemented in CV, but it requires the compiler to verify that the selected assignment operator is trivial.

The following example shows multiple, mass, and cascading assignment used in one expression

```c
int a, b;
double c, d;
[ void ] f([ int, int ]); // assignments in parameter list
f([c, a] = [b, d] = 1.5);
The tuple expression begins with a mass assignment of 1.5 into [b, d], which assigns 1.5 into b, which is truncated to 1, and 1.5 into d, producing the tuple [1, 1.5] as a result. That tuple is used as the right side of the multiple assignment (i.e., [c, a] = [1, 1.5]) that assigns 1 into c and 1.5 into a, which is truncated to 1, producing the result [1, 1]. Finally, the tuple [1, 1] is used as an expression in the call to f.
```

### 19.7 Construction

Tuples construction and destruction follow the same rules and semantics as tuple assignment, except that in the case where there is no right side, the default constructor or destructor is called on each component of the tuple. As constructors and destructors did not exist in previous versions of CV or in K-W C, this is a primary contribution of this thesis to the design of tuples.

```c
struct S;
void ?() S *; // (1)
void ?() S * int; // (2)
void ?() S * double; // (3)
void ?() S * S; // (4)
[S, S] x = [3, 6.28]; // uses (2), (3), specialized constructors
[S, S] y; // uses (1), (1), default constructor
[S, S] z = x.0; // uses (4), (4), copy constructor
```
In this example, `x` is initialized by the multiple constructor calls `?{}(&x.0, 3)` and `?{}(&x.1, 6.28)`, while `y` is initialized by two default constructor calls `?{}(&y.0)` and `?{}(&y.1)`. `z` is initialized by mass copy constructor calls `?{}(&z.0, x.0)` and `?{}(&z.1, x.0)`. Finally, `x`, `y`, and `z` are destructured, i.e., the calls `^{}?{}(&x.0, ^{}?{}(&x.1, ^{}?{}(&y.0, ^{}?{}(&y.1, ^{}?{}(&z.0, and `^{}?{}(&z.1).`  

It is possible to define constructors and assignment functions for tuple types that provide new semantics, if the existing semantics do not fit the needs of an application. For example, the function `void ?{}([T, U] * S);` can be defined to allow a tuple variable to be constructed from a value of type `S`.

```c
struct S { int x; double y; };
void ?{}([int, double] * this, S s) {
  this->x = s.x;
  this->y = s.y;
}
```

Due to the structure of generated constructors, it is possible to pass a tuple to a generated constructor for a type with a member prefix that matches the type of the tuple. For example,

```c
struct S { int x; double y; int z; };
struct [int, double] t;
S s = t;
```

The initialization of `s` with `t` works by default because `t` is flattened into its components, which satisfies the generated field constructor `?{}([S * int, double]` to initialize the first two values.

### 19.8 Member-Access Expression

Tuples may be used to select multiple fields of a record by field name. The result is a single tuple-valued expression whose type is the tuple of the types of the members. For example,

```c
struct S { char x; int y; double z; } s;
s.[x, y, z];
```

Here, the type of `s.[x, y, z]` is `[char, int, double]`. A member tuple expression has the form `e.[x, y, z]` where `e` is an expression with type `T`, where `T` supports member access expressions, and `x`, `y`, `z` are all members of `T` with types `T$\_x$`, `T$\_y$`, and `T$\_z$` respectively. Then the type of `e.[x, y, z]` is `[T$\_x$, T$\_y$, and T$\_z$]` respectively. Then the type of `e.[x, y, z]` is `[T$\_x$, T$\_y$, and T$\_z$]` respectively.

A member-access tuple may be used anywhere a tuple can be used, e.g.:  

```c
s.[y, z, x] = [3, 3.2, 'x'];  // equivalent to s.x = 'x', s.y = 3, s.z = 3.2
f(s.[y, z]);  // equivalent to f(s.y, s.z)
```

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.

Since tuple-index expressions are a form of member-access expression, it is possible to use tuple-index expressions in conjunction with member-access expressions to restructure a tuple (e.g., rearrange components, drop components, duplicate components, etc.).

```c
[int, int, long, double] x;
void f(double, long);
```

It is possible for a member tuple expression to contain other member access expressions, e.g.:

```c
struct A { double i; int j; };
struct B { int * k; short l; };
struct C { int x; A y; B z; } v;
v.[x, y.[i, j], z.k];
```

This expression is equivalent to `[x, y.[v.y.i, v.y.j]., v.z.k]`. That is, the aggregate expression is effectively distributed across the tuple allowing simple and easy access to multiple components in an aggregate without repetition. It is guaranteed that the aggregate expression to the left of the . in a member tuple expression is evaluated exactly once. As such, it is safe to use member tuple expressions on the result of a function with side-effects.

```c
[x, [int, float, double] f()];
[x, [double, float] x = f()[2, 1];  // f() called once
In C, member tuple expressions are known as record field tuples [31]. Since C\textvisiblespace A permits these tuple-access expressions using structures, unions, and tuples, member tuple expression or field tuple expression is more appropriate.

### 19.9 Casting

In C, the cast operator is used to explicitly convert between types. In C\textvisiblespace A, the cast operator has a secondary use, which is type ascription, since it forces the expression resolution algorithm to choose the lowest cost conversion to the target type. That is, a cast can be used to select the type of an expression when it is ambiguous, as in the call to an overloaded function.

```c
int f(); // (1)
double f(); // (2)
f(); // ambiguous - (1), (2) both equally viable
(int)f(); // choose (2)
```

Since casting is a fundamental operation in C\textvisiblespace A, casts need to be given a meaningful interpretation in the context of tuples. Taking a look at standard C provides some guidance with respect to the way casts should work with tuples.

```c
1 int f();
2 void g();
3
4 (void)f(); // valid, ignore results
5 (int)g(); // invalid, void cannot be converted to int
6
7 struct A { int x; };
8 (struct A)f(); // invalid, int cannot be converted to A
```

In C, line 4 is a valid cast, which calls f and discards its result. On the other hand, line 5 is invalid, because g does not produce a result, so requesting an int to materialize from nothing is nonsensical. Finally, line 8 is also invalid, because in C casts only provide conversion between scalar types [21, p. 91]. For consistency, this implies that any case wherein the number of components increases as a result of the cast is invalid, while casts that have the same or fewer number of components may be valid.

Formally, a cast to tuple type is valid when \( T_n \leq S_m \), where \( T_n \) is the number of components in the target type and \( S_m \) is the number of components in the source type, and for each \( i \) in \([0,n]\), \( S_i \) can be cast to \( T_i \). Excess elements (\( S_j \) for all \( j \) in \([n,m]\)) are evaluated, but their values are discarded so that they are not included in the result expression. This discarding naturally follows the way that a cast to void works in C.

For example,

```c
[int, int, int] f();
[int, [int, int], int] g();

([int, double])f(); // (1) valid
([int, int])g(); // (2) valid
([void, [int, int]])g(); // (3) valid
([int, int, int])g(); // (4) invalid
([int, [int, int]])g(); // (5) invalid
```

(1) discards the last element of the return value and converts the second element to type double. Since int is effectively a 1-element tuple, (2) discards the second component of the second element of the return value of g. If g is free of side effects, this is equivalent to \([\text{int}](\text{g}().0), (\text{int}](\text{g}().1.0), (\text{int}](\text{g}().2))\]. Since void is effectively a 0-element tuple, (3) discards the first and third return values, which is effectively equivalent to \([\text{int}](\text{g}().1.0), (\text{int}](\text{g}().1.1))\). Note that a cast is not a function call in C\textvisiblespace A, so flattening and structuring conversions do not occur for cast expressions. As such, (4) is invalid because the cast target type contains 4 components, while the source type contains only 3. Similarly, (5) is invalid because the cast \([\text{int}, \text{int}]](\text{g}().1)\) is invalid. That is, it is invalid to cast \([\text{int}, \text{int}]\) to \([\text{int}, \text{int}, \text{int}]\).

### 19.10 Polymorphism

Due to the implicit flattening and structuring conversions involved in argument passing, otype and dtype parameters are restricted to matching only with non-tuple types. The integration of polymorphism, type assertions, and monomorphic specialization of tuple-assertions are a primary contribution of this thesis to the design of tuples.
38
19.10 Polymorphism

```c
forall(T, dtype U)
void f(T x, U * y);

f([5, "hello"]);
```

In this example, [5, "hello"] is flattened, so that the argument list appears as 5, "hello". The argument matching algorithm binds T to int and U to const char, and calls the function as normal.

Tuples can contain polymorphic types. For example, a plus operator can be written to add two triples of a type together.

```c
forall(T | { T ?+?(T, T); })
return [x.0+y.0, x.1+y.1, x.2+y.2];
}
```

```c
int, int] x;
int i1, i2, i3;
[i1, i2, i3] = x + ([10, 20, 30]);
```

Note that due to the implicit tuple conversions, this function is not restricted to the addition of two triples. A call to this plus operator type checks as long as a total of 6 non-tuple arguments are passed after flattening, and all of the arguments have a common type that can bind to T with a pairwise ?+? over T. For example, these expressions also succeed and produce the same value.

```c
(x.0, x.1) + ((x.2, 10, 20, 30)); // x + ([10, 20, 30])
x.0 + (x.1, x.2, 10, 20, 30); // x + (10, 20, 30)
```

This presents a potential problem if structure is important, as these three expressions look like they should have different meanings. Furthermore, these calls can be made ambiguous by introducing seemingly different functions.

```c
forall(T | { T ?+?(T, T); })
forall(T | { T ?+?(T, T); })
[T, T, T] ?+?(T x, [T, T, T, T];)
```

It is also important to note that these calls could be disambiguated if the function return types were different, as they likely would be for a reasonable implementation of ?+?, since the return type is used in overload resolution. Still, these semantics are a deficiency of the current argument matching algorithm, and depending on the function, differing return values may not always be appropriate. These issues could be rectified by applying an appropriate conversion cost to the structuring and flattening conversions, which are currently 0-cost conversions in the expression resolver. Care would be needed in this case to ensure that exact matches do not incur such a cost.

```c
void f([[int, int], int, int]);
```

```c
f([0, 0], 0, 0); // no cost
f(0, 0, 0, 0);  // cost for structuring
f([0, 0, ], [0, 0]); // cost for flattening
f([0, 0, 0], 0); // cost for flattening and structuring
```

Until this point, it has been assumed that assertion arguments must match the parameter type exactly, modulo polymorphic specialization (i.e., no implicit conversions are applied to assertion arguments). This decision presents a conflict with the flexibility of tuples.

### 19.10.1 Assertion Inference

```c
int f([[int, double], double]);
```

```c
forall(T, U | { T f(T, U, U); })
void g(T, U);
g(5, 10.21);
```

If assertion arguments must match exactly, then the call to g cannot be resolved, since the expected type of f is flat, while the only f in scope requires a tuple type. Since tuples are fluid, this requirement reduces the usability of tuples in polymorphic code. To ease this pain point, function parameter and return lists are flattened for the purposes of type unification, which allows the previous example to pass expression resolution.

This relaxation is made possible by extending the existing thunk generation scheme, as described by Bilson [2]. Now, whenever a candidate’s parameter structure does not exactly match the formal parameter’s structure, a thunk is
generated to specialize calls to the actual function.

```c
int _thunk(int _p0, double _p1, double _p2) {
    return f([_p0, _p1, _p2];
}
```

Essentially, this provides flattening and structuring conversions to inferred functions, improving the compatibility of tuples and polymorphism.

## 20 Tuples

In C and C++, 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 lexical list. The general syntax of a lexical list is:

```
[ exprlist ]
```

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

- `[ 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 19.5, p. 33). In essence, tuples are largely a compile time phenomenon, having little or no runtime presence.

Tuples can be organized into compile-time 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 or C 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`
- `[ double, double, double ]`

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 x;` // 2 element tuple, each element of type int
- `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 ] f ( [[ int, int, int ]]);` // 3 element tuple as input
- `f( [1, 2, 3]);`
- `f( w1, 3 );`
- `f( 1, w1 );`
20.1 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:

\[
\begin{align*}
&\{\text{int, int, int, int}\} w; \\
&w = \{1, 2, 3, 4\};
\end{align*}
\]

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:

\[
\begin{align*}
&a, b, c, d = w
\end{align*}
\]

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:

\[
\begin{align*}
&a, b, c, d = \{1, 2, 3, 4\};
\end{align*}
\]

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 \([\text{int, int, int, int}]\) can be coerced, using flattening, into the type \([\text{int, int, int, int}]\).

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 \([\text{int, int, int, int}]\) into the tuple type \([\text{int, int, int, int}]\). In the following example, the last assignment illustrates all the tuple coercions:
20.2 Mass Assignment

```c
[ 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 19.6, p. 34).

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

20.2 Mass Assignment

C permits assignment to several variables at once using mass assignment [25]. Mass assignment has the following form:

```c
[ 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:

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

is equivalent to:

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

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

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

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

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

which is equivalent to:

```c
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.

20.3 Multiple Assignment

C also supports the assignment of several values at once, known as multiple assignment [25, 16]. Multiple assignment has the following form:

```c
[ 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 `lvalues` on the left-hand side of the statement using parallel semantics for each assignment. An example of multiple assignment is:

```c
[ 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:

```c
[ 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:

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

correctly interchanges (swaps) the values stored in `x` and `y`. The following cases are errors:
42

[72x746]42

[72x746]21.1 Basic I/O

[87x724]\[ a, b, c \] = \[ 1, 2, 3, 4 \];

[59x724]1

[87x714]\[ a, b, c \] = \[ 1, 2 \];

[59x714]2

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

[72x700]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:

[59x688]f( x++, x++ ); // C routine call with side effects in arguments

[72x688][ v1, v2 ] = \[ x++, x++ \]; // side effects in right-hand side of multiple assignment

8

20.4 Cascade Assignment

As in C, C\# 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.

21 Stream I/O Library

The goal of C\# stream input/output (I/O) is to simplify the common cases, while fully supporting polymorphism and
user defined types in a consistent way. Stream I/O can be implicitly or explicitly formatted. Implicit formatting means
C\# selects the output or input format for values that matches the variable’s type. Explicit formatting means additional
information is specified to augment how an output or input of value is interpreted. C\# formatting incorporates ideas
from C printf, C++ stream manipulators, and Python implicit spacing and newline. Specifically:

- print/Python format codes are dense, making them difficult to read and remember. C\#/C++ format manipulators
  are named, making them easier to read and remember.
- print/Python separates format codes from associated variables, making it difficult to match codes with variables. C\#/C++ co-locate codes with associated variables, where C\# has the tighter binding.
- Format manipulators in C\# have local effect, whereas C++ have global effect, except setw. Hence, it is common
  programming practice to toggle manipulators on and then back to the default to prevent downstream side-effects.
  Without this programming style, errors occur when moving prints, as manipulator effects incorrectly flow into the
  new location. Furthermore, to guarantee no side-effects, manipulator values must be saved and restored across
  function calls.
- C\# has more sophisticated implicit value spacing than Python, plus implicit newline at the end of a print.

The standard polymorphic I/Os stream are stdin/sin (input), stdout/sout and stderr/serr (output) (like C++ cin/cout/cerr).
Polymorphic streams exit and abort provide implicit program termination without and with generating a stack trace
and core file. Stream exit implicitly returns EXIT\_FAILURE to the shell.

exit | "x (" | x | ") negative value."; // terminate and return EXIT\_FAILURE to shell
abort | "x (" | x | ") negative value."; // terminate and generate stack trace and core file

Note, C\# stream variables stdin, stdout, stderr, exit, and abort overload C variables stdin, stdout, stderr, and functions
exit and abort, respectively. The C\# header file for the I/O library is fstream.hfa.

21.1 Basic I/O

For implicit formatted output, the common case is printing a series of variables separated by whitespace.

<table>
<thead>
<tr>
<th>C#</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x = 1, y = 2, z = 3;</td>
<td>cout &lt;&lt; x &lt;&lt; &quot; &quot; &lt;&lt; y &lt;&lt; &quot; &quot; &lt;&lt; z &lt;&lt; endl;</td>
<td>print( x, y, z )</td>
</tr>
<tr>
<td>sout</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>1_2_3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The C form has half the characters of the C++ form, and is similar to Python I/O with respect to implicit separators and newline. Similar simplification occurs for tuple I/O, which flattens the tuple and prints each value separated by "$\ldots$".

```
[int, [ int, int ]] t1 = [ 1, [ 2, 3 ] ], t2 = [ 4, [ 5, 6 ] ];
sout | t1 | t2; // print tuples
1, 2, 3, 4, 5, 6
```

Finally, C uses the logical-or operator for I/O as it is the lowest-priority overloadable operator, other than assignment. Therefore, fewer output expressions require parenthesis.

```
C: sout | x * 3 | y + 1 | z < 2 | x == y | (x | y) | (x || y) | (x > z ? 1 : 2);
C++: cout << x * 3 << y + 1 << (z < 2) << (x == y) << (x | y) << (x || y) << (x > z ? 1 : 2) << endl;
```

Input and output use a uniform operator, $\mid$, rather than C++’s $\gg$ and $\ll$ input/output operators. There is a weak similarity between the C logical-or operator and the Shell pipe-operator for moving data, where data flows in the correct direction for input but the opposite direction for output.

For implicit formatted input, the common case is reading a sequence of values separated by whitespace, where the type of an input constant must match with the type of the input variable.

```
int x; double y; char z;
C: sin | x | y | z; cin >> x >> y >> z; x = int(input()); y = float(input()); z = input();
C++: 1, 2.5_A 1, 2.5_A 1, 2.5_A
Python: 1 2.5 A
```

Figure 5 shows idiomatic C command-line processing and copying an input file to an output file. Note, a stream variable may be copied because it is a reference to an underlying stream data-structures. All I/O errors are handles as exceptions, but end-of-file is not an exception as C programmers are use to explicitly checking for it.

Figure 6, p. 45 shows the stream operations.

- fail tests the stream error-indicator, returning nonzero if it is set.
- clear resets the stream error-indicator.
- flush (ofstream only) causes any unwritten data for a stream to be written to the file.
- eof (ifstream only) tests the end-of-file indicator for the stream pointed to by stream. Returns true if the end-of-file indicator is set, otherwise false.
- open binds the file with name to a stream accessed with mode (see fopen).
- close flushes the stream and closes the file.
- write (ofstream only) write size bytes to the stream. The bytes are written lazily to file when internal buffers fill. Eager buffer writes are done with flush
- read (ifstream only) read size bytes to the stream.
- ungetc (ifstream only) pushes the character back to the input stream. Pushed-back characters returned by subsequent reads in the reverse order of pushing.

The constructor functions:

- create an unbound stream, which is subsequently bound to a file with open.
- create a bound stream to the associated file with given mode.

The destructor closes the stream.

### 21.2 Implicit Separator

The implicit separator character (space/blank) is a separator not a terminator for output. The rules for implicitly adding the separator are:

1. A separator does not appear at the start or end of a line.

```
sout | 1 | 2 | 3;
```
```cpp
#include <fstream.hfa>

int main(int argc, char * argv[]) {
    ifstream in = stdin;     // copy default files
    ofstream out = stdout;
    try {
        choose( argc ) {
            case 2, 3:
                open( in, argv[1] );     // open input file first as output creates file
                if ( argc == 3 ) open( out, argv[2] );   // do not create output unless input opens
            case 1: ;                      // use default files
            default:
                exit | "Usage" | argv[0] | "[ input-file (default stdin) "
                    | "[ output-file (default stdout) ] ]";
        } // choose
    } catch( Open_Failure * ex; ex->istream == &in ) {
        exit | "Unable to open input file" | argv[1];
    } catch( Open_Failure * ex; ex->ostream == &out ) {
        close( in );           // optional
        exit | "Unable to open output file" | argv[2];
    } // try

    out | nlOff;       // turn off auto newline
    in | nlOn;         // turn on reading newline
    char ch;
    for () {        // read/write characters
        in | ch;
        if ( eof( in ) ) break;   // eof ?
        out | ch;
    } // for
} // main
```
21.3 Separation Manipulators

The following manipulators control implicit output separation. The effect of these manipulators is global for an output
stream (except sepOn and sepOff).

1. sepSet and sep/sepGet set and get the separator string. The separator string can be at most 16 characters
including the '0' string terminator (15 printable characters).

```
sepSet( sout, ", \$" ); // set separator from " " to ", $"
sout [1 | 2 | 3 | " \" \" | sep | " \" ];
1 | 2 | 3 | " \" \" | sepGet( sout ) | " \" ;
```

sepGet can be used to store a separator and then restore it:

```
x(1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

6. A separator does not appear before/after a C string starting/ending with the ASCII quote or whitespace charac-
ters: "": \t\v\f\r\n
sout | 1 | 2 | 3 | 
sep | 2 | "\" ;
x(1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

7. If a space is desired before or after one of the special string start/end characters, simply insert a space.
```
sout | "x ( | 1 | " ) x" | 2 | " , x" | 3 | " : x : " | 4;
x(1 | 2 | 3 | 4

```

Figure 6: Stream Functions
21.4 Newline Manipulators

1. char store[sepSize]; // sepSize is the maximum separator size
2. strcpy( store, sepGet( sout ) ); // copy current separator
3. sepSet( sout, "_" ); // change separator to underscore
4. sout | 1 | 2 | 3;
5. // change separator back to original
6. sepSet( sout, store );
7. sout | 1 | 2 | 3;
8. 

2. sepSetTuple and sepTuple/sepGetTuple get and set the tuple separator-string. The tuple separator-string can be at most 16 characters including the \0 string terminator (15 printable characters).
10. sepSetTuple( sout, " "); // set tuple separator from " " to " "
11. sout | t1 | t2 | " \n " | sepTuple | "\n";
12. 1_2_3_4_5_6_7_8_9_10_11_12_13_14_15_16_17
13. sepSetTuple( sout, ", " ); // reset tuple separator to ", , "
14. sout | t1 | t2 | " \n " | sepGetTuple( sout ) | "\n";
15. 1_2_3_4_5_6_7_8_9_10_11_12_13_14_15_16_17
16. As for sepGet, sepGetTuple can be use to store a tuple separator and then restore it.

3. sepDisable and sepEnable toggle printing the separator.
18. sout | sepDisable | 1 | 2 | 3; // turn off implicit separator
19. 123
20. sout | sepEnable | 1 | 2 | 3; // turn on implicit separator
21. 1_2_3

4. sepOn and sepOff toggle printing the separator with respect to the next printed item, and then return to the global separator setting.
23. sout | 1 | sepOff | 2 | 3; // turn off implicit separator for the next item
24. 12_3
25. sout | sepDisable | 1 | sepOn | 2 | 3; // turn on implicit separator for the next item
26. 1_2_3
27. The tuple separator also responses to being turned on and off.
29. sout | t1 | sepOff | t2; // turn off implicit separator for the next item
30. 1_2_3_4_5_6_7_8_9_10_11_12_13_14_15_16_17_18_19_20_21_22_23_24_25_26_27_28_29_30_31_32_33
31. sepOn cannot be used to start/end a line with a separator because separators do not appear at the start/end of a line; use sep to accomplish this functionality.
33. sout | sepOn | 1 | 2 | 3 | sepOn; // sepOn does nothing at start/end of line
34. 1_2_3
35. sout | sep | 1 | 2 | 3 | sep ; // use sep to print separator at start/end of line
36. 1_2_3

21.4 Newline Manipulators
38. The following manipulators control newline separation for input and output.
39. For input:
40. 1. nl scans characters until the next newline character, i.e., ignore the remaining characters in the line.
41. 2. nlOn reads the newline character, when reading single characters.
42. 3. nlOff does not read the newline character, when reading single characters.
43. For example, in:
44. i | nl | j;
45. 1 2
46. 3
21.5 Output Value Manipulators

variable i is assigned 1, the 2 is skipped, and variable j is assigned 3.

For output:

1. `nl` inserts a newline.

2. For output:

   1. `sout | nl;` // only print newline
   2. `sout | 2;` // implicit newline
   3. `sout | 3 | nl | 4 | nl;` // terminating nl merged with implicit newline
   4. `sout | 5 | nl | nl;` // again terminating nl merged with implicit newline
   5. `sout | 6;` // implicit newline

   Note, a terminating `nl` is merged (overrides) with the implicit newline at the end of the `sout` expression, otherwise it is impossible to to print a single newline

2. `nlOn` implicitly prints a newline at the end of each output expression.

3. `nlOff` does not implicitly print a newline at the end of each output expression.

21.5 Output Value Manipulators

The following manipulators control formatting of output values (printing), and only affect the format of the argument.

1. `bin( integer )` print value in base 2 preceded by `0b/0B`.

2. `oct( integer )` print value in base 8 preceded by `0`.

3. `hex( integer / floating-point )` print value in base 16 preceded by `0x/0X`.

4. `sci( floating-point )` print value in scientific notation with exponent. Default is 6 digits of precision.

5. `eng( floating-point )` print value in engineering notation with exponent, which means the exponent is adjusted to a multiple of 3.

   - `0xe5 0xffe5 0xffffffffffe5`
6. unit( engineering-notation ) print engineering exponent as a letter between the range \(10^{-24}\) and \(10^{24}\): \(y \Rightarrow 10^{-24}\), \(z \Rightarrow 10^{-21}\), \(a \Rightarrow 10^{-18}\), \(f \Rightarrow 10^{-15}\), \(p \Rightarrow 10^{-12}\), \(n \Rightarrow 10^{-9}\), \(u \Rightarrow 10^{-6}\), \(m \Rightarrow 10^{-3}\), \(K \Rightarrow 10^{3}\), \(M \Rightarrow 10^{6}\), \(G \Rightarrow 10^{9}\), \(T \Rightarrow 10^{12}\), \(P \Rightarrow 10^{15}\), \(E \Rightarrow 10^{18}\), \(Z \Rightarrow 10^{21}\), \(Y \Rightarrow 10^{24}\). For exponent \(10^{0}\), no decimal point or letter is printed.

\[
sout | \text{unit(eng( 0.0 ))} | \text{unit(eng( 27000.5 ))} | \text{unit(eng( -27.5e7 ))};
0 27.0005K \text{ –275M}
\]

7. upcase( bin / hex / floating-point ) print letters in a value in upper case. Lower case is the default.

\[
sout | \text{upcase( bin( 27 ) )} | \text{upcase( hex(27 ) )} | \text{upcase( 27.5e-10 )} | \text{upcase( hex( 27.5 ) )};
0B11011 0X1B 2.75E-09 0X1.B8P+4
\]

8. nobase( integer ) do not precede bin, oct, hex with 0b/0B, 0, or 0x/0X. Printing the base is the default.

\[
sout | \text{nobase( bin( 27 ) )} | \text{nobase( oct(27 ) )} | \text{nobase( hex( 27 ) )};
11011 33 1b
\]

9. nodp( floating-point ) do not print a decimal point if there are no fractional digits. Printing a decimal point is the default, if there are no fractional digits.

\[
sout | \text{0. } | \text{nodp( 0. )} | \text{27.0 } | \text{nodp( 27.0 )} | \text{nodp( 27.5 )};
0.0 27.0 27.5
\]

10. sign( integer / floating-point ) prefix with plus or minus sign (+ or –). Only printing the minus sign is the default.

\[
sout | \text{sign( 27 )} | \text{sign( -27 )} | \text{sign( 27. )} | \text{sign( -27. )} | \text{sign( 27.5 )} | \text{sign( -27.5 )};
+27 -27 +27.0 -27.0 +27.5 -27.5
\]

11. wd( minimum, value ), wd( minimum, precision, value ) For all types, minimum is the number of printed characters. If the value is shorter than the minimum, it is padded on the right with spaces.

\[
sout | \text{wd( 4, 34 )} | \text{wd( 3, 34 )} | \text{wd( 2, 34 )};
sout | \text{wd( 10, 4 )} | \text{wd( 9, 4 )} | \text{wd( 8, 4 )};
sout | \text{wd( 4, "ab" )} | \text{wd( 3, "ab" )} | \text{wd( 2, "ab" )};
34 34 34
34.000000 34.000000 34.000000
ab ab ab
\]

If the value is larger, it is printed without truncation, ignoring the minimum.

\[
sout | \text{wd( 4, 34567 )} | \text{wd( 3, 34567 )} | \text{wd( 2, 34567 )};
sout | \text{wd( 4, 3456. )} | \text{wd( 3, 3456. )} | \text{wd( 2, 3456. )};
sout | \text{wd( 4, "abcde" )} | \text{wd( 3, "abcde" )} | \text{wd( 2,"abcde" )};
3456 3456 3456
3456 3456 3456
abcde abcd abcd
\]

For integer types, precision is the minimum number of printed digits. If the value is shorter, it is padded on the left with leading zeros.

\[
sout | \text{wd( 4,3, 34 )} | \text{wd( 8,4, 34 )} | \text{wd( 10,10, 34 )};
034 00034 0000000034
\]

If the value is larger, it is printed without truncation, ignoring the precision.

\[
sout | \text{wd( 4,1, 3456 )} | \text{wd( 8,2, 3456 )} | \text{wd( 10,3, 3456 )};
3456 3456 3456
3456 3456 3456
\]

If precision is 0, nothing is printed for zero. If precision is greater than the minimum, it becomes the minimum.

\[
sout | \text{wd( 4,0, 0 )} | \text{wd( 3,10, 34 )};
0000000034
\]

For floating-point types, precision is the minimum number of digits after the decimal point.

\[
sout | \text{wd( 6,3, 27.5 )} | \text{wd( 8,1, 27.5 )} | \text{wd( 8,0, 27.5 )} | \text{wd( 3,8, 27.5 )};
27.500 27.5 28 27.50000000
\]

For the C-string type, precision is the maximum number of printed characters, so the string is truncated if it exceeds the maximum.
21.6 Input Value Manipulators

1. `sout | wd( 6,8, "abcd" ) | wd( 6,8, "abcdefg" ) | wd( 6,3, "abcd" );
   a...abc

2. `sout | wd( 6,8, "abcd" ) | wd( 6,8, "abcdefghijk" ) | wd( 6,3, "abcd" );
   a...abc

12. `ws( minimum, significant, floating-point )` For floating-point types, minimum is the same as for manipulator `wd`, but significant is the maximum number of significant digits to be printed for both the integer and fractions (versus only the fraction for `wd`). If a value’s significant digits is greater than significant, the last significant digit is rounded up.

3. `sout | ws(6,6, 234.567) | ws(6,5, 234.567) | ws(6,4, 234.567) | ws(6,3, 234.567);`
   234.567 234.5 234. 23

4. `sout | ws(6,6, 234567.) | ws(6,5, 234567.) | ws(6,4, 234567.) | ws(6,3, 234567.);`
   234567. 2.346e+05 2.35e+05

5. `sout | ws(3,6, 234567.) | ws(4,6, 234567.) | ws(5,6, 234567.) | ws(6,6, 234567.);`
   234567. 234567. 234567. 234567.

6. `left( field-width )` left justify within the given field.

7. `sout | left(wd(4, 27)) | left(wd(10, 27)) | left(wd(10, 27.5)) | left(wd(4,3, 27)) | left(wd(10,3, 27.5));`
   27.000000 27.500000 27.500000

8. `pad0( field-width )` left pad with zeroes (0).

9. `sout | pad0( wd(4, 27) ) | pad0( wd(4, 27) ) | pad0( wd(4, 27.5) ) | pad0( wd(8, 3, 27.5) );`
   0027 027 0027.500

21.6 Input Value Manipulators

The format of numeric input values in the same as C constants without a trailing type suffix, as the input value-type is denoted by the input variable. For `bool` type, the constants are true and false. For integral types, any number of digits, optionally preceded by a sign (+ or −), where a

1. 1-9 prefix introduces a decimal value (0-9),

2. 0 prefix introduces an octal value (0-7), and

3. 0x or 0X prefix introduces a hexadecimal value (0-f) with lower or upper case letters.

For floating-point types, any number of decimal digits, optionally preceded by a sign (+ or −), optionally containing a decimal point, and optionally followed by an exponent, e or E, with signed (optional) decimal digits. Floating-point values can also be written in hexadecimal format preceded by 0x or 0X with hexadecimal digits and exponent denoted by p or P.

For the C-string type, the input values are not the same as C-string constants, i.e., double quotes bracketing arbitrary text with escape sequences. Instead, the next sequence of non-whitespace characters are read, and the input sequence is terminated with delimiter ‘\0’. The string variable `must` be large enough to contain the input sequence.

The following manipulators control formatting of input values (reading), and only affect the format of the argument.

1. `skip( pattern ), skip( length )` The pattern is composed of white-space and non-white-space characters, where

   "any" white-space character matches 0 or more input white-space characters (hence, consecutive white-space characters in the pattern are combined), and each non-white-space character matches exactly with an input character. The length is composed of the next N characters, including the newline character. If the match succeeds, the input characters are discarded, and input continues with the next character. If the match fails, the input characters are left unread.

   ```cpp
   char sk[] = "abc";
   ``

   ```cpp
   sout | "abc" | skip(sk) | skip(5); // match input sequence
   ``

   ```cpp
   abcc
   abc
   ``

   ```cpp
   xx
   ```
2. \texttt{wdi}(\textit{maximum, reference-value}) For all types except \texttt{char}, maximum is the maximum number of characters read for the current operation.

\begin{verbatim}
  char s[10];  int i;  double d;
  sin | wdi( 4, s ) | wdi( 3, i ) | wdi( 8, d ); // c == "abcd", i == 123, d == 3.456E+2
  abcd1233.456E+2
\end{verbatim}

Note, input \texttt{wdi} cannot be overloaded with output \texttt{wd} because both have the same parameters but return different types. Currently, \texttt{CV} cannot distinguish between these two manipulators in the middle of an \texttt{sout/sin} expression based on return type.

3. \texttt{ignore}(\textit{reference-value}) For all types, the data is read from the stream depending on the argument type but ignored, \textit{i.e.}, it is not stored in the argument.

\begin{verbatim}
  double d;
  sin | ignore( d ); // d is unchanged
  d = 75.35e-4
\end{verbatim}

4. \texttt{incl}(\textit{scanset, input-string}) For C-string types, the scanset matches any number of characters \textit{in} the set. Matching characters are read into the C input-string and null terminated.

\begin{verbatim}
  char s[10];
  sin | incl( "abc", s );
  bcaxyz
\end{verbatim}

5. \texttt{excl}(\textit{scanset, input-string}) For C-string types, the scanset matches any number of characters \textit{not in} the set. Non-matching characters are read into the C input-string and null terminated.

\begin{verbatim}
  char s[10];
  sin | excl( "abc", s );
  xyzbca
\end{verbatim}

## 21.7 Concurrent Stream Access

When a stream is shared by multiple threads, input or output characters can be intermixed or cause failure. For example, if two threads execute the following:

\begin{verbatim}
thread1 : sout | "abc " | "def ";
thread2 : sout | "uvw " | "xyz ";
\end{verbatim}

possible outputs are:

\begin{verbatim}
  abc def  abc uvw xyz  uvw abc xyz def  abuvw def  uvw abc def
  uvw xyz  def  uvw xyz
\end{verbatim}

Concurrent operations can even corrupt the internal state of the stream resulting in failure. As a result, some form of mutual exclusion is required for concurrent stream access.

A coarse-grained solution is to perform all stream operations via a single thread or within a monitor providing the necessary mutual exclusion for the stream. A fine-grained solution is to have a lock for each stream, which is acquired and released around stream operations by each thread. \texttt{CV} provides a fine-grained solution where a recursive lock is acquired and released indirectly via a manipulator \texttt{acquire} or instantiating an RAII type specific for the kind of stream: \texttt{osacquire} for output streams and \texttt{isacquire} for input streams.

The common usage is the short form of the mutex statement to lock a stream during a single cascaded I/O expression, \textit{e.g.}:

\begin{verbatim}
thread1 : mutex(sout) sout | "abc " | "def ";
thread2 : mutex(sout) sout | "uvw " | "xyz ";
\end{verbatim}

Now, the order of the thread execution is still non-deterministic, but the output is constrained to two possible lines in either order.

\begin{verbatim}
  abc def  uvw xyz
  uvw xyz  abc def
\end{verbatim}

In summary, the stream lock is acquired by the \texttt{acquire} manipulator and implicitly released at the end of the cascaded I/O expression ensuring all operations in the expression occur atomically.

To lock a stream across multiple I/O operations, the long form of the mutex statement is used, \textit{e.g.}:
21.8 Locale

Cultures use different syntax, called a **locale**, for printing numbers so they are easier to read, *e.g.*:

- 12,345.123  // comma separator, period decimal-point
- 12.345,123  // period separator, comma decimal-point
- 12 345,1.23 // space separator, comma decimal-point, period terminator

A locale is selected with function `setlocale`, and the corresponding locale package **must** be installed on the underlying system; `setlocale` returns 0 if the requested locale is unavailable. Furthermore, a locale covers the syntax for many cultural items, *e.g.*, address, measurement, money, etc. This discussion applies to item `LC_NUMERIC` for formatting non-monetary integral and floating-point values. Figure 7 shows selecting different cultural syntax, which may be associated with one or more countries.

22 String Stream

The stream types `ostrstream` and `istrstream` provide all the stream formatting capabilities to/from a C string rather than a stream file. Figure 8, p. 53 shows writing (output) to and reading (input) from a C string. The only string stream operations different from a file stream are:

- constructors to create a stream that writes to a write buffer (`ostrstream`) of size, or reads from a read buffer (`istrstream`) containing a C string terminated with '\0'.
  ```cpp
  void ???( ostrstream &, char buf[], size_t size );
  void ???( istrstream & is, char buf[] );
  ```
- write (ostrstream only) writes all the buffered characters to the specified stream (stdout default).
  ```cpp
  ostrstream & write( ostrstream & os, FILE * stream = stdout );
  ```

There is no read for `istrstream`. 
#include <fstream.h>
#include <locale.h>  // setlocale
#include <stdlib.h>  // getenv

int main() {
    void print() {
        sout | 12 | 123 | 1234 | 12345 | 123456 | 1234567;
        sout | 12. | 123.1 | 1234.12 | 12345.123 | 123456.1234 | 1234567.12345;
        sout | nl;
    }

    sout | "Default locale off";
    print();
    sout | "Locale on" | setlocale( LC_NUMERIC, getenv( "LANG" ) ); // enable local locale
    print();
    sout | "German" | setlocale( LC_NUMERIC, "de_DE.UTF-8" ); // enable German locale
    print();
    sout | "Ukraine" | setlocale( LC_NUMERIC, "uk_UA.utf8" ); // enable Ukraine locale
    print();
    sout | "Default locale off" | setlocale( LC_NUMERIC, "C" ); // disable locale
    print();
}

Default locale off
12 123 1234 12345 123456 1234567
12. 123.1 1234.12 12345.123 123456.1234 1234567.12345

Locale on en_US.UTF-8
12 123 1.234 1234 123456 1.234 1234567 123456
12. 1.234 12.345 1.234 1234 123456 1.234 1234567

German de_DE.UTF-8
12 123 1.234 1234 123456 1.234 1234567 123456
12. 1.234 12.345 1.234 1234 123456 1.234 1234567

Ukraine uk_UA.utf8
12 123 1 234 1234 1 234 1 234 1 234
12. 1.234 12.345 1.234 1234 123456 1.234 1234567

Default locale off C
12 123 1234 12345 123456 1234567
12. 1.234 12.345 1.234 1234 123456 1.234 1234567

Figure 7: Stream Locale

23 Structures

Structures in C are basically the same as structures in C. A structure is defined with the same syntax as in C. When referring to a structure in C, users may omit the struct keyword.

struct Point {
    double x;
    double y;
};

Point p = (0.0, 0.0);

C does not support inheritance among types, but instead uses composition to enable reuse of structure fields. Composition is achieved by embedding one type into another. When type A is embedded in type B, an object with type B may be used as an object of type A, and the fields of type A are directly accessible. Embedding types is achieved using anonymous members. For example, using Point from above:
```cpp
#include <fstream.hfa>
#include <strstream.hfa>

int main() {
    enum { size = 256 };    // output buffer
    char buf[size];
    ostrstream osstr = { buf, size };    // bind output buffer/size
    int i = 3, j = 5, k = 7;
    double x = 12345678.9, y = 98765.4321e-11;

    osstr | i | hex(j) | wd(10, k) | sci(x) | unit(eng(y)) | "abc";
    write(osstr);        // write string to stdout
    printf( "%s" , buf ); // same lines of output
    sout | i | hex(j) | wd(10, k) | sci(x) | unit(eng(y)) | "abc";

    char buf2[] = "12 14 15 3.5 7e4 abc";    // input buffer
    istrstream isstr = { buf2 };
    char s[10];
    isstr | i | j | k | x | y | s;
    sout | i | j | k | x | y | s;
}
```

Figure 8: String Stream Processing

```cpp
void foo(Point p);

struct ColoredPoint {
    Point; // anonymous member (no identifier)
    int Color;
};

...  
ColoredPoint cp = ...;
    cp.x = 10.3; // x from Point is accessed directly
    cp.color = 0x33aaff; // color is accessed normally
    foo(cp); // cp can be used directly as a Point
```

## 24 Constructors and Destructors

C supports C initialization of structures, but it also adds constructors for more advanced initialization. Additionally, C adds destructors that are called when a variable is deallocated (variable goes out of scope or object is deleted). These functions take a reference to the structure as a parameter (see Section 2, p. 18 for more information).

## 25 Overloading

Overloading refers to the capability of a programmer to define and use multiple objects in a program with the same name. In C, a declaration may overload declarations from outer scopes with the same name, instead of hiding them as is the case in C. This may cause identical C and C programs to behave differently. The compiler selects the appropriate object (overload resolution) based on context information at the place where it is used. Overloading allows programmers to give functions with different signatures but similar semantics the same name, causing confusion, or that the compiler may resolve to a different function from what the programmer expected. C allows overloading of functions, operators, variables, and even the constants 0 and 1.
struct Widget {
    int id;
    float size;
    Parts *optionalParts;
};

// {} is the constructor operator identifier
// The first argument is a reference to the type to initialize
// Subsequent arguments can be specified for initialization

void {}(Widget &w) { // default constructor
    w.id = -1;
    w.size = 0.0;
    w.optionalParts = 0;
}

// constructor with values (does not need to include all fields)
void {}(Widget &w, int id, float size) {
    w.id = id;
    w.size = size;
    w.optionalParts = 0;
}

// ^ is the destructor operator identifier
void ^?(Widget &w) { // destructor
    w.id = 0;
    w.size = 0.0;
    if (w.optionalParts != 0) {
        // This is the only pointer to optionalParts, free it
        free(w.optionalParts);
        w.optionalParts = 0;
    }
}

Widget baz; // reserve space only
Widget foo{}; // calls default constructor
Widget bar{23, 2.45}; // calls constructor with values
baz{24, 0.91}; // calls constructor with values
{}(baz, 24, 0.91); // explicit call to constructor
^bar; // explicit call to destructor
^?(bar); // explicit call to destructor

Figure 9: Constructors and Destructors

The compiler follows some overload resolution rules to determine the best interpretation of all of these overloads. The best valid interpretations are the valid interpretations that use the fewest unsafe conversions. Of these, the best are those where the functions and objects involved are the least polymorphic. Of these, the best have the lowest total conversion cost, including all implicit conversions in the argument expressions. Of these, the best have the highest total conversion cost for the implicit conversions (if any) applied to the argument expressions. If there is no single best valid interpretation, or if the best valid interpretation is ambiguous, then the resulting interpretation is ambiguous. For details about type inference and overload resolution, please see the C Language Specification.

int foo(int a, int b) {
    float sum = 0.0;
    float special = 1.0;
    {
        int sum = 0;
        // both the float and int versions of sum are available
25.1 Constant

The constants 0 and 1 have special meaning. In C, as in C, all scalar types can be incremented and decremented, which is defined in terms of adding or subtracting 1. The operations &amp;, ||, and ! can be applied to any scalar arguments and are defined in terms of comparison against 0 (ex. `(a &amp; b)` becomes `(a != 0 &amp; &amp; b != 0)`).

In C, the integer constants 0 and 1 suffice because the integer promotion rules can convert them to any arithmetic type, and the rules for pointer expressions treat constant expressions evaluating to 0 as a special case. However, user-defined arithmetic types often need the equivalent of a 1 or 0 for their functions or operators, polymorphic functions often need 0 and 1 constants of a type matching their polymorphic parameters, and user-defined pointer-like types may need a null value. Defining special constants for a user-defined type is more efficient than defining a conversion to the type from bool.

Why just 0 and 1? Why not other integers? No other integers have special status in C. A facility that let programmers declare specific constants, for instance, would not be much of an improvement. Some facility for defining the creation of values of programmer-defined types from arbitrary integer tokens would be needed. The complexity of such a feature does not seem worth the gain.

For example, to define the constants for a complex type, the programmer would define the following:

```c
struct Complex {
    double real;
    double imaginary;
}
const Complex 0 = {0, 0};
const Complex 1 = {1, 0};
...
Complex a = 0;
...
    a++;
    ...
    if (a) { // same as if (a == 0)
    ...
}
```

25.2 Variable

The overload rules of C allow a programmer to define multiple variables with the same name, but different types. Allowing overloading of variable names enables programmers to use the same name across multiple types, simplifying naming conventions and is compatible with the other overloading that is allowed. For example, a developer may want to do the following:

```c
int pi = 3;
float pi = 3.14;
char pi = 'p';
```

25.3 Function Overloading

Overloaded functions in C are resolved based on the number and type of arguments, type of return value, and the level of specialization required (specialized functions are preferred over generic).

The examples below give some basic intuition about how the resolution works.
// Choose the one with less conversions
int doSomething(int value) {...} // option 1
int doSomething(short value) {...} // option 2

int a, b = 4;
short c = 2;

a = doSomething(b); // chooses option 1
a = doSomething(c); // chooses option 2

// Choose the specialized version over the generic
generic(type T)
T bar(T rhs, T lhs) {...} // option 3
float bar(float rhs, float lhs){...} // option 4
float a, b, c;
double d, e, f;
c = bar(a, b); // chooses option 4

// specialization is preferred over unsafe conversions
f = bar(d, e); // chooses option 5

25.4 Operator

C uses the following special identifiers to name overloaded operators:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[],[]</td>
<td>subscripting</td>
</tr>
<tr>
<td>?++</td>
<td>addition</td>
</tr>
<tr>
<td>?-=</td>
<td>subtraction</td>
</tr>
<tr>
<td>?+=</td>
<td>exponentiation assignment</td>
</tr>
<tr>
<td>?&lt;&lt;?</td>
<td>left shift</td>
</tr>
<tr>
<td>?-&gt;=</td>
<td>multiplication assignment</td>
</tr>
<tr>
<td>?&gt;&gt;&gt;?</td>
<td>right shift</td>
</tr>
<tr>
<td>?=?</td>
<td>division assignment</td>
</tr>
<tr>
<td>?&lt;=&gt;</td>
<td>less than or equal</td>
</tr>
<tr>
<td>?=??</td>
<td>remainder assignment</td>
</tr>
<tr>
<td>?++?</td>
<td>addition assignment</td>
</tr>
<tr>
<td>?-=?</td>
<td>subtraction assignment</td>
</tr>
<tr>
<td>?&gt;&lt;=?</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>?&gt;=??</td>
<td>subtraction assignment</td>
</tr>
<tr>
<td>?&gt;&lt;=?</td>
<td>left-shift assignment</td>
</tr>
<tr>
<td>?==?</td>
<td>equality</td>
</tr>
<tr>
<td>?&gt;=??</td>
<td>right-shift assignment</td>
</tr>
<tr>
<td>?&amp;?=</td>
<td>bitwise AND assignment</td>
</tr>
<tr>
<td>?&amp;=?</td>
<td>bitwise assignment</td>
</tr>
<tr>
<td>?&amp;?=</td>
<td>exclusive OR assignment</td>
</tr>
<tr>
<td>?</td>
<td>=?</td>
</tr>
<tr>
<td>?/=?</td>
<td>bitwise OR assignment</td>
</tr>
<tr>
<td>?%-?</td>
<td>remainder</td>
</tr>
</tbody>
</table>

Table 1: Operator Identifiers

These identifiers are defined such that the question marks in the name identify the location of the operands. These operands represent the parameters to the functions, and define how the operands are mapped to the function call. For example, a + b becomes ?++?(a, b).

In the example below, a new type, myComplex, is defined with an overloaded constructor, + operator, and string operator. These operators are called using the normal C syntax.

type Complex = struct { // define a Complex type
double real;
double imag;
}
// Constructor with default values
void ?{}(Complex &c, double real = 0.0, double imag = 0.0) {
    c.real = real;
    c.imag = imag;
}

Complex ?+(Complex lhs, Complex rhs) {
    Complex sum;
    sum.real = lhs.real + rhs.real;
    sum.imag = lhs.imag + rhs.imag;
    return sum;
}

String ()?(const Complex c) {
    // use the string conversions for the structure members
    return (String)c.real + . + . + (String)c.imag + .i.;
}

Complex a, b, c = {1.0}; // constructor for c w/ default imag
... c = a + b; // constructor for c w/ default imag
print(.sum = . + c);

26  Auto Type-Inferencing

Auto type-inferencing occurs in a declaration where a variable’s type is inferred from its initialization expression type.

C++

```
#define expr 3.0 * i
auto j = 3.0 * 4;    // use type of initialization expression
int i;
auto k = i;          // use type of primary variable
```

gcc

The two important capabilities are:

• not determining or writing long generic types,
• ensuring secondary variables, related to a primary variable, always have the same type.

In C++, typedef provides a mechanism to alias long type names with short ones, both globally and locally, but not eliminate the use of the short name. gcc provides typeof to declare a secondary variable from a primary variable. C++ also relies heavily on the specification of the left-hand side of assignment for type inferencing, so in many cases it is crucial to specify the type of the left-hand side to select the correct type of the right-hand expression. Only for overloaded routines with the same return type is variable type-inferencing possible. Finally, auto presents the programming problem of tracking down a type when the type is actually needed. For example, given

```
auto j = ...
```

and the need to write a routine to compute using j

```
void rtn( ... parm );
rtn( j );
```

A programmer must work backwards to determine the type of j’s initialization expression, reconstructing the possibly long generic type-name. In this situation, having the type name or a short alias is very useful.

There is also the conundrum in type inferencing of when to brand a type. That is, when is the type of the variable more important than the type of its initialization expression. For example, if a change is made in an initialization expression, it can cause cascading type changes and/or errors. At some point, a variable type needs to remain constant and the expression to be in error when it changes.

Given typedef and typeof in C++, and the strong need to use the type of left-hand side in inferencing, auto type-inferencing is not supported at this time. Should a significant need arise, this feature can be revisited.


```c
#include <fstream.hfa>
#include <coroutine.hfa>

coroutine Fibonacci {
    int fn; // used for communication
};

void main( Fibonacci & fib ) with( fib ) { // called on first resume
    int fn1, fn2; // retained between resumes
    fn = 0; fn1 = fn; // 1st case
    suspend; // restart last resume
    fn1 = 1; fn2 = fn1; fn1 = fn; // 2nd case
    suspend; // restart last resume
    for () {
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn; // general case
        suspend; // restart last resume
    }
}

int next( Fibonacci & fib ) with( fib ) { // restart last suspend
    resume( fib );
    return fn;
}

int main() {
    Fibonacci f1, f2;
    for ( 10 ) {
        sout | next( f1 ) | next( f2 );
    }
}
```

Figure 10: Fibonacci Coroutine

27 Concurrency

Concurrency support in CÀ is implemented on top of a highly efficient runtime system of light-weight, M:N, user level threads. The model integrates concurrency features into the language by making the structure type the core unit of concurrency. All communication occurs through method calls, where data is sent via method arguments, and received via the return value. This enables a very familiar interface to all programmers, even those with no parallel programming experience. It also allows the compiler to do static type checking of all communication, a very important safety feature. This controlled communication with type safety has some similarities with channels in Go, and can actually implement channels exactly, as well as create additional communication patterns that channels cannot. Mutex objects, monitors, are used to contain mutual exclusion within an object and synchronization across concurrent threads.

27.1 Coroutine

Coroutines are the precursor to threads. Figure 10 shows a coroutine that computes the Fibonacci numbers.

27.2 Monitors

A monitor is a structure in CÀ which includes implicit locking of its fields. Users of a monitor interact with it just like any structure, but the compiler handles code as needed to ensure mutual exclusion. An example of the definition of a monitor is shown here:

```c
type Account = monitor {
    const unsigned long number; // account number
    float balance; // account balance
};
```
27.3 Threads

C\textsuperscript{A} also provides a simple mechanism for creating and utilizing user level threads. A thread provides mutual exclusion like a monitor, and also has its own execution state and a thread of control. Similar to a monitor, a thread is defined like a structure:

28 Language Comparisons

C\textsuperscript{A} is one of many languages that attempts to improve upon C. In developing C\textsuperscript{A}, many other languages were consulted for ideas, constructs, and syntax. Therefore, it is important to show how these languages each compare with Do. In this section, C\textsuperscript{A} is compared with what the writers of this document consider to be the closest competitors of Do: \texttt{C++}, Go, Rust, and D.

28.1 C++

C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation. (Wikipedia)

The primary focus of C++ seems to be adding object-oriented programming to C, and this is the primary difference between C++ and Do. C++ uses classes to encapsulate data and the functions that operate on that data, and to hide the internal representation of the data. C\textsuperscript{A} uses modules instead to perform these same tasks. Classes in C++ also enable inheritance among types. Instead of inheritance, C\textsuperscript{A} embraces composition and interfaces to achieve the same goals with more flexibility. There are many studies and articles comparing inheritance and composition (or is-a versus has-a relationships), so we will not go into more detail here (Venners, 1998) (Pike, Go at Google: Language Design in the Service of Software Engineering, 2012).
#include <fstream>
#include <kernel>
#include <stdlib>
#include <thread>

thread First { signal_once * lock; };
thread Second { signal_once * lock; };

void ?{}( First * this, signal_once * lock ) { this->lock = lock; }
void ?{}( Second * this, signal_once * lock ) { this->lock = lock; }

void main( First * this ) {
    for ( int i = 0; i < 10; i += 1 ) {
        sout | "First : Suspend No." | i + 1;
        yield();
    }
    signal( this->lock );
}

void main( Second * this ) {
    wait( this->lock );
    for ( int i = 0; i < 10; i += 1 ) {
        sout | "Second : Suspend No." | i + 1;
        yield();
    }
}

int main( void ) {
    signal_once lock;
    sout | "User main begin";
    {
        processor p;
        {
            First f = { &lock };
            Second s = { &lock };
        }
        sout | "User main end";
    }
}

Figure 12: Simple Threads

Overloading in C# is very similar to overloading in C++, with the exception of the additional use, in C#, of the return type to differentiate between overloaded functions. References and exceptions in C# are heavily based on the same features from C++. The mechanism for interoperating with C code in C# is also borrowed from C++.

Both C# and C++ provide generics, and the syntax is quite similar. The key difference between the two, is that in C++ templates are expanded at compile time for each type for which the template is instantiated, while in C#, function pointers are used to make the generic fully compilable. This means that a generic function can be defined in a compiled library, and still be used as expected from source.

28.2 Go

Go, also commonly referred to as golang, is a programming language developed at Google in 2007 []. It is a statically typed language with syntax loosely derived from that of C, adding garbage collection, type safety, some structural typing capabilities, additional built-in types such as variable-length arrays and key-value maps, and a large standard library. (Wikipedia)

Go and C# differ significantly in syntax and implementation, but the underlying core concepts of the two languages
are aligned. Both Go and C\textvisibel{\textalpha} use composition and interfaces as opposed to inheritance to enable encapsulation and abstraction. Both languages (along with their tooling ecosystem) provide a simple packaging mechanism for building units of code for easy sharing and reuse. Both languages also include built-in lightweight, user-level threading concurrency features that attempt to simplify the effort and thought process required for writing parallel programs while maintaining high performance.

Go has a significant runtime which handles the scheduling of its lightweight threads, and performs garbage collection, among other tasks. C\textvisibel{\textalpha} uses a cooperative scheduling algorithm for its tasks, and uses automatic reference counting to enable advanced memory management without garbage collection. This results in Go requiring significant overhead to interface with C libraries while C\textvisibel{\textalpha} has no overhead.

28.3 Rust

Rust is a general-purpose, multi-paradigm, compiled programming language developed by Mozilla Research. It is designed to be a "safe, concurrent, practical language", supporting pure-functional, concurrent-actor\[dubious . discuss\][citation needed], imperative-procedural, and object-oriented styles.

The primary focus of Rust is in safety, especially in concurrent programs. To enforce a high level of safety, Rust has added ownership as a core feature of the language to guarantee memory safety. This safety comes at the cost of a difficult learning curve, a change in the thought model of the program, and often some runtime overhead.

Aside from those key differences, Rust and C\textvisibel{\textalpha} also have several similarities. Both languages support no overhead interoperability with C and have minimal runtimes. Both languages support inheritance and polymorphism through the use of interfaces (traits).

28.4 D

The D programming language is an object-oriented, imperative, multi-paradigm system programming language created by Walter Bright of Digital Mars and released in 2001. [.] Though it originated as a re-engineering of C++, D is a distinct language, having redesigned some core C++ features while also taking inspiration from other languages, notably Java, Python, Ruby, C#, and Eiffel.

D and C\textvisibel{\textalpha} both start with C and add productivity features. The obvious difference is that D uses classes and inheritance while C\textvisibel{\textalpha} uses composition and interfaces. D is closer to C\textvisibel{\textalpha} than C++ since it is limited to single inheritance and also supports interfaces. Like C++, and unlike C\textvisibel{\textalpha}, D uses garbage collection and has compile-time expanded templates. D does not have any built-in concurrency constructs in the language, though it does have a standard library for concurrency which includes the low-level primitives for concurrency.

A Syntax Ambiguities

C has a number of syntax ambiguities, which are resolved by taking the longest sequence of overlapping characters that constitute a token. For example, the program fragment x+++++y is parsed as \(x + + + + + y\) because operator tokens ++ and + overlap. Unfortunately, the longest sequence violates a constraint on increment operators, even though the parse \(x + + + + + + y\) might yield a correct expression. Hence, C programmers are aware that spaces have to added to disambiguate certain syntactic cases.

In C\textvisibel{\textalpha}, there are ambiguous cases with dereference and operator identifiers, e.g., int *?*?(), where the string *?*? can be interpreted as:

\[
\begin{align*}
\star ? \star ? & \quad \text{// dereference operator, dereference operator} \\
\star & \quad \text{// dereference, multiplication operator}
\end{align*}
\]

By default, the first interpretation is selected, which does not yield a meaningful parse. Therefore, C\textvisibel{\textalpha} does a lexical look-ahead for the second case, and backtracks to return the leading unary operator and reparses the trailing operator identifier. Otherwise a space is needed between the unary operator and operator identifier to disambiguate this common case.

A similar issue occurs with the dereference, *?(...), and routine-call, ?()(...) identifiers. The ambiguity occurs when the deference operator has no parameters:

\[
\begin{align*}
*?(&) & \\
*?(&,...) &
\end{align*}
\]


requiring arbitrary whitespace look-ahead for the routine-call parameter-list to disambiguate. However, the dereference operator must have a parameter/argument to dereference \( \ast \{ \ldots \} \). Hence, always interpreting the string \( \ast \{ \} \) as \( \ast \{ \} \) does not preclude any meaningful program.

The remaining cases are with the increment/decrement operators and conditional expression, e.g.:

1. \( i++ ? \ldots (\ldots) ; \)
2. \( i?++ \ldots (\ldots) ; \)

requiring arbitrary whitespace look-ahead for the operator parameter-list, even though that interpretation is an incorrect expression (juxtaposed identifiers). Therefore, it is necessary to disambiguate these cases with a space:

1. \( i++ ? i : 0 ; \)
2. \( i?++ i : 0 ; \)

### B C Incompatibles

The following incompatibles exist between C\& and C, and are similar to Annex C for C++ [22].

1. **Change:** add new keywords

   New keywords are added to C\& (see Section C, p. 65).

   **Rationale:** keywords added to implement new semantics of C\&.

   **Effect on original feature:** change to semantics of well-defined feature.

   Any C11 programs using these keywords as identifiers are invalid C\& programs.

   **Difficulty of converting:** keyword clashes are accommodated by syntactic transformations using the C\& backquote escape-mechanism (see Section 6, p. 5).

   **How widely used:** clashes among new C\& keywords and existing identifiers are rare.

2. **Change:** drop K&R C declarations

   K&R declarations allow an implicit base-type of int, if no type is specified, plus an alternate syntax for declaring parameters. e.g.:

   1. \( x ; \)  // int x
   2. \( \ast y ; \)  // int \( \ast \) y
   3. \( f( p1, p2 ) ; \)  // int f( int p1, int p2 ) ;
   4. \( g( p1, p2 ) \text{ int } p1, p2 ; \)  // int g( int p1, int p2 ) ;

   C\& continues to support K&R routine definitions:

   1. \( f( a, b, c ) ; \)  // default int return
   2. \( \text{ int a, b; char c } \)  // K&R parameter declarations
   3. \( \{ \ldots \} \)

   **Rationale:** dropped from C11 standard.\(^{19}\)

   **Effect on original feature:** original feature is deprecated.

   Any old C programs using these K&R declarations are invalid C\& programs.

   **Difficulty of converting:** trivial to convert to C\&.

   **How widely used:** existing usages are rare.

3. **Change:** type of character literal int to char to allow more intuitive overloading:

   1. \( \text{ int rtnt( int i ) ; } \)
   2. \( \text{ int rtnt( char c ) ; } \)
   3. \( \text{ rtnt( \text{ 'x' } ) ; } \)  // programmer expects 2nd rtnt to be called

   **Rationale:** it is more intuitive for the call to rtnt to match the second version of definition of rtnt rather than the first. In particular, output of char variable now print a character rather than the decimal ASCII value of the character.

\(^{19}\)At 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 [21, § 6.7.2(2)]
Having to cast `x` to char is non-intuitive.

**Effect on original feature:** change to semantics of well-defined feature that depend on:

```c
sizeof( 'x' ) == sizeof( int )
```

no long work the same in CV programs.

**Difficulty of converting:** simple

**How widely used:** programs that depend upon `sizeof( 'x' )` are rare and can be changed to `sizeof(char)`.

4. **Change:**
make string literals const:

```c
char * p = "abc";        // valid in C, deprecated in CV
char * q = expr ? "abc" : "de";       // valid in C, invalid in CV
```

The type of a string literal is changed from `[] char` to `const [] char`. Similarly, the type of a wide string literal is changed from `[] wchar_t` to `const [] wchar_t`.

**Rationale:**
This change is a safety issue:

```c
char * p = "abc";
p[0] = 'w';          // segment fault or change constant literal
```

The same problem occurs when passing a string literal to a routine that changes its argument.

**Effect on original feature:** change to semantics of well-defined feature.

**Difficulty of converting:** simple syntactic transformation, because string literals can be converted to `char *`.

**How widely used:** programs that have a legitimate reason to treat string literals as pointers to potentially modifiable memory are rare.

5. **Change:**
remove tentative definitions, which only occurs at file scope:

```c
int i;        // forward definition
int *j = &i;  // forward reference, valid in C, invalid in CV
int i = 0;    // definition
```

is valid in C, and invalid in CV because duplicate overloaded object definitions at the same scope level are disallowed. This change makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```c
struct X { int i; struct X *next; };  // forward definition
static struct X a;                   // forward definition
static struct X b = { 0, &a };       // forward reference, valid in C, invalid in CV
static struct X a = { 1, &b };       // definition
```

**Rationale:**
avoids having different initialization rules for builtin types and user-defined types.

**Effect on original feature:** change to semantics of well-defined feature.

**Difficulty of converting:** the initializer for one of a set of mutually-referential file-local static objects must invoke a routine call to achieve the initialization.

**How widely used:** seldom

6. **Change:**
have struct introduce a scope for nested types:

```c
enum Colour { R, G, B, Y, C, M };
struct Person {
    enum Colour { R, G, B };        // nested type
    struct Face {                   // nested type
        Colour Eyes, Hair;         // type defined outside (1 level)
    };
    .Colour shirt;                 // type defined outside (top level)
    Colour pants;                 // type defined same level
    Face looks[10];               // type defined same level
};
```

**Rationale:** avoids having different initialization rules for builtin types and user-defined types.

**Effect on original feature:** change to semantics of well-defined feature.

**Difficulty of converting:** the initializer for one of a set of mutually-referential file-local static objects must invoke a routine call to achieve the initialization.

**How widely used:** seldom
In C, the name of the nested types belongs to the same scope as the name of the outermost enclosing structure, i.e., the nested types are hoisted to the scope of the outer-most type, which is not useful and confusing. C++ is C incompatible on this issue, and provides semantics similar to C++. Nested types are not hoisted and can be referenced using the field selection operator ".", unlike the C++ scope-resolution operator "::".

**Rationale:** struct scope is crucial to C++ as an information structuring and hiding mechanism.

**Effect on original feature:** change to semantics of well-defined feature.

**Difficulty of converting:** Semantic transformation.

**How widely used:** C programs rarely have nest types because they are equivalent to the hoisted version.

7. **Change:** In C++, the name of a nested class is local to its enclosing class.

**Rationale:** C++ classes have member functions which require that classes establish scopes.

**Difficulty of converting:** Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```c
struct Y;  // struct Y and struct X are at the same scope
struct X {
    struct Y { /*...*/ y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.3.

**How widely used:** Seldom.

8. **Change:** remove implicit conversion of void * to or from any T * pointer:

```c
void foo() {
    int * b = malloc( sizeof(int) ); // implicitly convert void * to int *
    char * c = b;  // implicitly convert int * to void *, and then void * to char *
}
```

**Rationale:** increase type safety

**Effect on original feature:** deletion of semantically well-defined feature.

**Difficulty of converting:** requires adding a cast (see Section E.1 for better alternatives):

```c
int * b = (int *)malloc( sizeof(int) );
char * c = (char *)b;
```

**How widely used:** Significant. Some C translators already give a warning if the cast is not used.

9. **Change:** Types must be declared in declarations, not in expressions In C, a sizeof expression or cast expression may create a new type. For example,

```c
p = (void*){struct x {int i;} *};
```

declares a new type, struct x.

**Rationale:** This prohibition helps to clarify the location of declarations in the source code.

**Effect on original feature:** Deletion of a semantically well-defined feature.

**Difficulty of converting:** Syntactic transformation.

**How widely used:** Seldom.

10. **Change:** comma expression is disallowed as subscript

**Rationale:** safety issue to prevent subscripting error for multidimensional arrays: `x[i,j]` instead of `x[i][j]`, and this syntactic form then taken by C++ for new style arrays.

**Effect on original feature:** change to semantics of well-defined feature.

**Difficulty of converting:** semantic transformation of `x[i,j]` to `x[i][j]`

**How widely used:** Seldom.
C \textit{C\# Keywords}

\text{C\#} introduces the following new keywords, which cannot be used as identifiers.

\begin{center}
\begin{tabular}{llllllll}
basetypeof & enable & finally & int128 & report & trait & when \\
choose & exception & fixup & monitor & suspend & try & with \\
coroutine & fallthrough & forall & mutex & throw & virtual & zero_t \\
disable & fallthru & generator & one_t & throwResume & waitfor \\
\end{tabular}
\end{center}

\text{C\#} introduces the following new quasi-keywords, which can be used as identifiers.

\begin{center}
\begin{tabular}{llllll}
catch & fixup & \\
catchResume & or & \\
finally & timeout & \\
\end{tabular}
\end{center}

D \textit{Standard Headers}

\text{C\#11} prescribes the following standard header-files [21, § 7.1.2] and \text{C\#} adds to this list:

\begin{center}
\begin{tabular}{llllllllll}
assert.h & float.h & math.h & stdatomic.h & stdlib.h & time.h & gmp.h \\
complex.h & inttypes.h & setjmp.h & stdbool.h & stdnoreturn.h & uchar.h & malloc.h \\
ctype.h & iso646.h & signal.h & stddef.h & string.h & wchar.h & unistd.h \\
erro.h & limits.h & stdalign.h &stdint.h & tgmath.h & wctype.h \\

tenv.h & locale.h & stdarg.h & stdio.h & thresds.h & \\
\end{tabular}
\end{center}

For the prescribed head-files, \text{C\#} uses header interposition to wraps these includes in an extern "C"; hence, names in these include files are not mangled (see Section 4, p. 2). All other \text{C} header files must be explicitly wrapped in extern "C" to prevent name mangling. This approach is different from \text{C++} where the name-mangling issue is handled internally in \text{C} header-files through checks for preprocessor variable \texttt{cplusplus}, which adds appropriate extern "C" qualifiers.

E \textit{Standard Library}

The \text{C\#} standard-library wraps explicitly-polymorphic \text{C} routines into implicitly-polymorphic versions.

E.1 \textit{Storage Management}

The storage-management routines extend their \text{C} equivalents by overloading, alternate names, providing shallow type-safety, and removing the need to specify the allocation size for non-array types.

\text{C} storage management provides the following capabilities:

\begin{itemize}
\item \texttt{filled} after allocation with a specified character or value.
\item \texttt{resize} an existing allocation to decreased or increased its size. In either case, new storage may or may not be allocated and, if there is a new allocation, as much data from the existing allocation is copied into the new allocation. For an increase in storage size, new storage after the copied data may be filled.
\item \texttt{align} an allocation on a specified memory boundary, e.g., an address multiple of 64 or 128 for cache-line purposes.
\item \texttt{array} the allocation size is scaled to the specified number of array elements. An array may be filled, resized, or aligned.
\end{itemize}

Table 2 shows allocation routines supporting different combinations of storage-management capabilities.

\text{C\#} memory management extends the type safety of all allocations by using the type of the left-hand-side type to determine the allocation size and return a matching type for the new storage. Type-safe allocation is provided for all \text{C} allocation routines and new \text{C\#} allocation routines, e.g., in

```
int * ip = (int *)malloc( sizeof(int) ); // C
```

```
int * ip = malloc(); // C\# type-safe version of C malloc
```

```
int * ip = alloc(); // C\# type-safe uniform alloc
```
### E.1 Storage Management

<table>
<thead>
<tr>
<th>C</th>
<th>malloc</th>
<th>calloc</th>
<th>realloc</th>
<th>memalign</th>
<th>aligned_alloc(^a)</th>
<th>posix_memalign</th>
<th>valloc</th>
<th>pvalloc(^b)</th>
<th>cmemalign</th>
<th>realloc</th>
<th>alloc</th>
<th>alloc_set</th>
<th>alloc_align</th>
<th>alloc_align_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>no</td>
<td>yes (0 only)</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes (page size)</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>resize</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>alignment</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes (page size)</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>array</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\)Same as `memalign` but size is an integral multiple of alignment, which is universally ignored.
\(^b\)Same as `valloc` but rounds size to multiple of page size.

Table 2: Allocation Routines versus Storage-Management Capabilities

The latter two allocations determine the allocation size from the type of `p` (int) and cast the pointer to the allocated storage to `int *`.

CV memory management extends allocation safety by implicitly honouring all alignment requirements, e.g., in

```c
struct S { int i; } __attribute__(( aligned( 128 ) )); // cache-line alignment
S * sp = malloc(); // honour type alignment
```

the storage allocation is implicitly aligned to 128 rather than the default 16. The alignment check is performed at compile time so there is no runtime cost.

CV memory management extends the resize capability with the notion of **sticky properties**. Hence, initial allocation capabilities are remembered and maintained when resize requires copying. For example, an initial alignment and fill capability are preserved during a resize copy so the copy has the same alignment and extended storage is filled. Without sticky properties it is dangerous to use `realloc`, resulting in an idiom of manually performing the reallocation to maintain correctness.

CV memory management extends allocation to support constructors for initialization of allocated storage, e.g., in

```c
struct S { int i; }; // cache-line alignment
void ?( S & s, int i ) { s.i = i; }
// assume ? operator for printing an S
S & sp = new( 3 ); // call constructor after allocation
sout | sp.i;
delete( &sp );
S & spa = anew( 10, 5 ); // allocate array and initialize each array element
for ( i; 10 ) sout | spa[i] | nonl;
sout | nl;
adelete( 10, spa );
```

Allocation routines `new/anew` allocate a variable/array and initialize storage using the allocated type’s constructor.

Note, the matching deallocation routines `delete/adelete`.

```c
extern "C" {
    // C unsafe allocation
    void * malloc( size_t size );
    void * calloc( size_t dim, size_t size );
    void * realloc( void * ptr, size_t size );
}
```
void * memalign(size_t align, size_t size);
void * aligned_alloc(size_t align, size_t size);
int posix_memalign(void ** ptr, size_t align, size_t size);
void * cmemalign(size_t alignment, size_t noOfElems, size_t elemSize); // CFA

// C unsafe initialization/copy
void * memcpv(void * dest, const void * src, size_t size);
void * realloc(void * oaddr, size_t nalign, size_t size); // CFA heap

forall(dtype T | sized(T)) {
  // CV safe equivalents, i.e., implicit size specification
  T * malloc(void);
  T * alloc(size_t dim);
  T * realloc(T * ptr, size_t size);
  T * memalign(size_t align);
  T * cmemalign(size_t align, size_t dim);
  T * aligned_alloc(size_t align);
  int posix_memalign(T ** ptr, size_t align);

  // CV safe general allocation, fill, resize, alignment, array
  T * alloc(void); // variable, T size
  T * alloc(size_t dim); // array[dim], T size elements
  T * alloc(T ptr[], size_t dim); // realloc array[dim], T size elements

  T * alloc_set(char fill); // variable, T size, fill bytes with value
  T * alloc_set(size_t dim, char fill); // array[dim], T size elements, fill bytes with value
  T * alloc_set(size_t dim, T fill); // array[dim], T size elements, fill elements with value
  T * alloc_set(size_t dim, const T fill[]); // array[dim], T size elements, fill elements with array
  T * alloc_set(T ptr[], size_t dim, char fill); // realloc array[dim], T size elements, fill bytes with value

  T * alloc_align(size_t align); // aligned variable, T size
  T * alloc_align(size_t dim); // aligned array[dim], T size elements
  T * alloc_align(T ptr[], size_t align); // realloc new aligned array
  T * alloc_align(T ptr[], size_t align, size_t dim); // realloc new aligned array[dim]

  T * alloc_align_set(size_t align, char fill); // aligned variable, T size, fill bytes with value
  T * alloc_align_set(size_t align, size_t dim); // aligned array[dim], T size elements
  T * alloc_align_set(T ptr[], size_t align, size_t dim); // realloc new aligned array

  // CV safe initialization/copy, i.e., implicit size specification
  T * memset(T * dest, char fill);
  T * memcpy(T * dest, const T * src);

  // CV safe initialization/copy, i.e., implicit size specification, array types
  T * amemset(T dest[], char fill, size_t dim);
  T * amemcpy(T dest[], const T src[], size_t dim);
}

// CV allocation/deallocation and constructor/destructor, non-array types
forall(dtype T | sized(T), ttype Params | (void ?[](T & T, Params & Params); ) T * new(Params p);
forall(dtype T | sized(T) | (void ?[](T & T); ) void delete(T * ptr);
forall(dtype T, ttype Params | sized(T) | (void ^ ?[](T & T); void delete(Params & Params); )
void delete(T * ptr, Params rest);
E.2 String to Value Conversion

int ato( const char * ptr );
unsigned int ato( const char * ptr );
long int ato( const char * ptr );
unsigned long int ato( const char * ptr );
long long int ato( const char * ptr );
float ato( const char * ptr );
double ato( const char * ptr );
long double ato( const char * ptr );
float _Complex ato( const char * ptr );
double _Complex ato( const char * ptr );
long double _Complex ato( const char * ptr );

int strto( const char * ptr, char ** eptr, int base );
unsigned int strto( const char * ptr, char ** eptr, int base );
long int strto( const char * ptr, char ** eptr, int base );
unsigned long int strto( const char * ptr, char ** eptr, int base );
long long int strto( const char * ptr, char ** eptr, int base );
unsigned long long int strto( const char * ptr, char ** eptr, int base );
float strto( const char * ptr, char ** eptr );
double strto( const char * ptr, char ** eptr );
long double strto( const char * ptr, char ** eptr );
float _Complex strto( const char * ptr, char ** eptr );
double _Complex strto( const char * ptr, char ** eptr );
long double _Complex strto( const char * ptr, char ** eptr );

E.3 Search / Sort

forall( T | { int ?<?( T, T ); } )                      // location
    T * bsearch( T key, const T * arr, size_t dim );
forall( T | { int ?<?( T, T ); } )                      // position
    unsigned int bsearch( T key, const T * arr, size_t dim );
forall( T | { int ?<?( T, T ); } )
    void qsort( const T * arr, size_t dim );

forall( E | { int ?<?( E, E ); } )
    E * bsearch( E key, const E * vals, size_t dim );    // location
    size_t bsearch( E key, const E * vals, size_t dim );  // position

forall( K, E | { int ?<?( K, K ); Key get( const E & ); } )
    E * bsearch( K key, const E * vals, size_t dim );
    size_t bsearch( K key, const E * vals, size_t dim );

E.4 Absolute Value

E * bsearchu( K key, const E * vals, size_t dim );
size_t bsearchu( K key, const E * vals, size_t dim );
}

forall( E | { int ?<( E, E ); } ) {
  void qsort( E * vals, size_t dim );
}

E.4 Absolute Value

unsigned char abs( signed char );
int abs( int );
unsigned long int abs( long int );
unsigned long long int abs( long long int );
float abs( float );
double abs( double );
long double abs( long double );
float abs( float_Complex );
double abs( double_Complex );
long double abs( long double_Complex );
forall( T | { void ?{}( T * zero_t ); int ?<( T, T ); T ->?( T ); } )
T abs( T );

E.5 Random Numbers

void srand( unsigned int seed );
char random( void );
char random( char u ); // [0,u)
char random( char l, char u ); // [l,u]
int random( void );
int random( int u ); // [0,u)
int random( int l, int u ); // [l,u]
unsigned int random( void );
unsigned int random( unsigned int u ); // [0,u)
unsigned int random( unsigned int l, unsigned int u ); // [l,u]
long int random( void );
long int random( long int u ); // [0,u)
long int random( long int l, long int u ); // [l,u]
unsigned long int random( void );
unsigned long int random( unsigned long int u ); // [0,u)
unsigned long int random( unsigned long int l, unsigned long int u ); // [l,u]
float random( void ); // [0.0, 1.0)
double random( void ); // [0.0, 1.0]
float_Complex random( void ); // [0.0, 1.0]+[0.0, 1.0)i
double_Complex random( void ); // [0.0, 1.0]+[0.0, 1.0)i
long double_Complex random( void ); // [0.0, 1.0]+[0.0, 1.0)i

E.6 Algorithms

forall( T | { int ?<( T, T ); } ) T min( T t1, T t2 );
forall( T | { int ?>?( T, T ); } ) T max( T t1, T t2 );
forall( T | { T min( T, T ); T max( T, T ); } ) T clamp( T value, T min_val, T max_val );
forall( T ) void swap( T * t1, T * t2 );

F Math Library

The C\* math-library wraps explicitly-polymorphic C math-routines into implicitly-polymorphic versions.
F.1 General

float fmod( float, float );
double fmod( double, double );
long double fmod( long double, long double );

float remainder( float, float );
double remainder( double, double );
long double remainder( long double, long double );

float remquo( float, float, int *);
double remquo( double, double, int *);
long double remquo( long double, long double, int *);

float fma( float, float, float );
double fma( double, double, double );
long double fma( long double, long double, long double );

float fdim( float, float );
double fdim( double, double );
long double fdim( long double, long double );

float nan( const char *);
double nan( const char *);
long double nan( const char *);

F.2 Exponential

float exp( float );
double exp( double );
long double exp( long double );
float _Complex exp( float _Complex );
double _Complex exp( double _Complex );
long double _Complex exp( long double _Complex );

float exp2( float );
double exp2( double );
long double exp2( long double );
// float _Complex exp2( float _Complex );
// double _Complex exp2( double _Complex );
// long double _Complex exp2( long double _Complex );

float expm1( float );
double expm1( double );
long double expm1( long double );

float pow( float, float );
double pow( double, double );
long double pow( long double, long double );
float _Complex pow( float _Complex, float _Complex );
double _Complex pow( double _Complex, double _Complex );
F.3 Logarithm

```
long double Complex pow( long double Complex, long double Complex );
```

F.3 Logarithm

```
float log( float );
double log( double );
long double log( long double );
float Complex log( float Complex );
double Complex log( double Complex );
long double Complex log( long double Complex );
int log2( unsigned int );
long int log2( unsigned long int );
long long int log2( unsigned long long int );
float log2( float );
double log2( double );
long double log2( long double );
// float Complex log2( float Complex );
// double Complex log2( double Complex );
// long double Complex log2( long double Complex );
float log10( float );
double log10( double );
long double log10( long double );
// float Complex log10( float Complex );
// double Complex log10( double Complex );
// long double Complex log10( long double Complex );
float log1p( float );
double log1p( double );
long double log1p( long double );
int ilogb( float );
int ilogb( double );
int ilogb( long double );
float logb( float );
double logb( double );
long double logb( long double );
float sqrt( float );
double sqrt( double );
long double sqrt( long double );
float Complex sqrt( float Complex );
double Complex sqrt( double Complex );
long double Complex sqrt( long double Complex );
float cbrt( float );
double cbrt( double );
long double cbrt( long double );
float hypot( float, float );
double hypot( double, double );
long double hypot( long double, long double );
```

F.4 Trigonometric

```
float sin( float );
double sin( double );
long double sin( long double );
```
float __Complex sin( float __Complex );
double __Complex sin( double __Complex );
long double __Complex sin( long double __Complex );

float cos( float );
double cos( double );
long double cos( long double );
float __Complex cos( float __Complex );
double __Complex cos( double __Complex );
long double __Complex cos( long double __Complex );

float tan( float );
double tan( double );
long double tan( long double );
float __Complex tan( float __Complex );
double __Complex tan( double __Complex );
long double __Complex tan( long double __Complex );

float asin( float );
double asin( double );
long double asin( long double );
float __Complex asin( float __Complex );
double __Complex asin( double __Complex );
long double __Complex asin( long double __Complex );

float acos( float );
double acos( double );
long double acos( long double );
float __Complex acos( float __Complex );
double __Complex acos( double __Complex );
long double __Complex acos( long double __Complex );

float atan( float );
double atan( double );
long double atan( long double );
float __Complex atan( float __Complex );
double __Complex atan( double __Complex );
long double __Complex atan( long double __Complex );

float atan2( float, float );
double atan2( double, double );
long double atan2( long double, long double );

// alternative name for atan2

float sinh( float );
double sinh( double );
long double sinh( long double );
float __Complex sinh( float __Complex );
double __Complex sinh( double __Complex );
long double __Complex sinh( long double __Complex );

float cosh( float );
double cosh( double );
long double cosh( long double );
float __Complex cosh( float __Complex );


double _Complex cosh( double _Complex );
long double _Complex cosh( long double _Complex );

float tanh( float );
double tanh( double );
long double tanh( long double );
float _Complex tanh( float _Complex );
double _Complex tanh( double _Complex );
long double _Complex tanh( long double _Complex );

float asinh( float );
double asinh( double );
long double asinh( long double );
float _Complex asinh( float _Complex );
double _Complex asinh( double _Complex );
long double _Complex asinh( long double _Complex );

float acosh( float );
double acosh( double );
long double acosh( long double );
float _Complex acosh( float _Complex );
double _Complex acosh( double _Complex );
long double _Complex acosh( long double _Complex );

float atanh( float );
double atanh( double );
long double atanh( long double );
float _Complex atanh( float _Complex );
double _Complex atanh( double _Complex );
long double _Complex atanh( long double _Complex );

float erf( float );
double erf( double );
long double erf( long double );
float _Complex erf( float _Complex );
double _Complex erf( double _Complex );
long double _Complex erf( long double _Complex );

float erfc( float );
double erfc( double );
long double erfc( long double );
float _Complex erfc( float _Complex );
double _Complex erfc( double _Complex );
long double _Complex erfc( long double _Complex );

float lgamma( float );
double lgamma( double );
long double lgamma( long double );
float lgamma( float, int * );
double lgamma( double, int * );
long double lgamma( long double, int * );

float lgamma( float );
double lgamma( double );
long double lgamma( long double );

F.7 Nearest Integer
// n / align
signed char floor( signed char n, signed char align );
unsigned char floor( unsigned char n, unsigned char align );
short int floor( short int n, short int align );
unsigned short int floor( unsigned short int n, unsigned short int align );
int floor( int n, int align );
unsigned int floor( unsigned int n, unsigned int align );
long int floor( long int n, long int align );
unsigned long int floor( unsigned long int n, unsigned long int align );
long long int floor( long long int n, long long int align );
unsigned long long int floor( unsigned long long int n, unsigned long long int align );

// (n + (align – 1)) / align
signed char ceiling_div( signed char n, char align );
unsigned char ceiling_div( unsigned char n, unsigned char align );
short int ceiling_div( short int n, short int align );
unsigned short int ceiling_div( unsigned short int n, unsigned short int align );
int ceiling_div( int n, int align );
unsigned int ceiling_div( unsigned int n, unsigned int align );
long int ceiling_div( long int n, long int align );
unsigned long int ceiling_div( unsigned long int n, unsigned long int align );
long long int ceiling_div( long long int n, long long int align );
unsigned long long int ceiling_div( unsigned long long int n, unsigned long long int align );

// floor( n + (n % align != 0 ? align – 1 : 0), align )
signed char ceiling( signed char n, signed char align );
unsigned char ceiling( unsigned char n, unsigned char align );
short int ceiling( short int n, short int align );
unsigned short int ceiling( unsigned short int n, unsigned short int align );
int ceiling( int n, int align );
unsigned int ceiling( unsigned int n, unsigned int align );
long int ceiling( long int n, long int align );
unsigned long int ceiling( unsigned long int n, unsigned long int align );
long long int ceiling( long long int n, long long int align );
unsigned long long int ceiling( unsigned long long int n, unsigned long long int align );

float floor( float );
double floor( double );
long double floor( long double );

float ceil( float );
double ceil( double );
long double ceil( long double );

float trunc( float );
double trunc( double );
long double trunc( long double );

float rint( float );
long double rint( long double );
long int rint( float );
long int rint( double );
long int rint( long double );
long int rint( long int rint( float ) );
long int rint( long int rint( double ) );
long int rint( long int rint( long double ) );
long long int llrint( float );
long long int llrint( double );
long long int llrint( long double );

float nearbyint( float );
double nearbyint( double );
long double nearbyint( long double );

float round( float );
long double round( long double );
long int round( float );
long double round( long double );
long double round( long double );
long double round( float );
long int round( long double );
long double round( long double );
long double round( long double );

long int lround( float );
long int lround( double );
long int lround( long double );
long long int llround( float );
long long int llround( double );
long long int llround( long double );

float copysign( float, float );
double copysign( double, double );
long double copysign( long double, long double );

float frexp( float, int * );
double frexp( double, int * );
long double frexp( long double, int * );

float ldexp( float, int );
long double ldexp( float, int );
long double ldexp( double, int );

[ float, float ] modf( float );
float modf( float, float * );
[ double, double ] modf( double );
double modf( double, double * );
[ long double, long double ] modf( long double );
long double modf( long double, long double * );

float nextafter( float, float );
double nextafter( double, double );
long double nextafter( long double, long double );

float nexttoward( float, long double );
double nexttoward( double, long double );
long double nexttoward( long double, long double );

float scalbn( float, int );
double scalbn( double, int );
long double scalbn( long double, int );

float scalbln( float, long int );
double scalbln( double, long int );
long double scalbln( long double, long int );
G. Time Keeping

G.1 Duration

```c
struct Duration {
    int64_t tn; // nanoseconds
};

void ?{}( Duration & dur);
void ?{}( Duration & dur, zero_t);
void ?{}( Duration & dur, timeval t )
void ?{}( Duration & dur, timespec t )

Duration ?=?( Duration & dur, zero_t );
Duration ?=?( Duration & dur, timeval t )
Duration ?=?( Duration & dur, timespec t )

Duration +?( Duration rhs );
Duration ?+?( Duration & lhs, Duration rhs );
Duration ?+=?( Duration & lhs, Duration rhs );

Duration -?( Duration rhs );
Duration ?-?( Duration & lhs, Duration rhs );
Duration ?-=?( Duration & lhs, Duration rhs );

int64_t ?/?( Duration rhs );
Duration ?/?( Duration & lhs, int64_t rhs );
Duration /=?( Duration & lhs, int64_t rhs );

double div( Duration lhs, Duration rhs );

Duration ?%?( Duration rhs );
Duration %=?( Duration & lhs, Duration rhs );

bool ?==?( Duration lhs, zero_t );
bool ?!=?( Duration lhs, zero_t );
bool ?<?( Duration lhs, zero_t );
bool ?<=?( Duration lhs, zero_t );
bool ?>?( Duration lhs, zero_t );
bool ?>=?( Duration lhs, zero_t );

bool ?==?( Duration lhs, Duration rhs );
bool ?!=?( Duration lhs, Duration rhs );
bool ?<?( Duration lhs, Duration rhs );
bool ?<=?( Duration lhs, Duration rhs );
bool ?>?( Duration lhs, Duration rhs );
bool ?>=?( Duration lhs, Duration rhs );

Duration abs( Duration rhs );
Duration ?'ns( int64_t nsec );
Duration ?'us( int64_t usec );
Duration ?'ms( int64_t msec );
Duration ?'s( int64_t sec );
Duration ?'s( double sec );
Duration ?'m( int64_t min );
Duration ?'m( double min );
Duration ?'h( int64_t hours );
```
Duration ?`h`( double hours );
Duration ?`d`( int64_t days );
Duration ?`w`( int64_t weeks );

int64_t ?`ns`( Duration dur );
int64_t ?`us`( Duration dur );
int64_t ?`ms`( Duration dur );
int64_t ?`s`( Duration dur );
int64_t ?`m`( Duration dur );
int64_t ?`h`( Duration dur );
int64_t ?`d`( Duration dur );
int64_t ?`w`( Duration dur );

double ?`dns`( Duration dur );
double ?`dus`( Duration dur );
double ?`dms`( Duration dur );
double ?`ds`( Duration dur );
double ?`dm`( Duration dur );
double ?`dh`( Duration dur );
double ?`dd`( Duration dur );
double ?`dw`( Duration dur );

Duration max( Duration lhs, Duration rhs );
Duration min( Duration lhs, Duration rhs );

forall( ostype & | ostream( ostype ) ) ostype & ?|?( ostype & os, Duration dur );

timeval

void ?{}( timeval & t );
void ?{}( timeval & t, zero_t );
void ?{}( timeval & t, time_t sec, suseconds_t usec );
void ?{}( timeval & t, time_t sec );
void ?{}( timeval & t, Time time );
timeval ?=?( timeval & t, zero_t );
timeval ?+?( timeval & lhs, timeval rhs );
timeval ?-?( timeval & lhs, timeval rhs );
bool ?==?( timeval lhs, timeval rhs );
bool ?!=?( timeval lhs, timeval rhs );

timespec

void ?{}( timespec & t );
void ?{}( timespec & t, zero_t );
void ?{}( timespec & t, time_t sec, __syscall_slong_t nsec );
void ?{}( timespec & t, time_t sec );
void ?{}( timespec & t, Time time );
timespec ?=?( timespec & t, zero_t );
timespec ?+?( timespec & lhs, timespec rhs );
timespec ?-?( timespec & lhs, timespec rhs );
bool ?==?( timespec lhs, timespec rhs );
bool ?!=?( timespec lhs, timespec rhs );

itimerval

void ?{}( itimerval & itv, Duration alarm );
void ?{}( itimerval & itv, Duration alarm, Duration interval );
G.5  Time

struct Time {
    uint64_t tn;  // nanoseconds since UNIX epoch
};

void ?[]( Time & time );
void ?[]( Time & time, zero_t );
void ?[]( Time & time, timeval t );
void ?[]( Time & time, timespec t );

Time ?=( Time & time, zero_t );
Time ?=( Time & time, timeval t );
Time ?=( Time & time, timespec t );

Time ?+( Time & lhs, Duration rhs );
Time ?+( Duration lhs, Time rhs );
Time ?+=( Time & lhs, Duration rhs );

Duration ?-( Time lhs, Time rhs );
Time ?-( Time lhs, Duration rhs );
Time ?=( Time lhs, Duration rhs );
bool ?==( Time lhs, Time rhs );
bool ?==( Time lhs, Time rhs );
bool ?<=( Time lhs, Time rhs );
bool ?<==( Time lhs, Time rhs );
bool ?>=( Time lhs, Time rhs );
bool ?>=( Time lhs, Time rhs );

int64_t ?'ns( Time t );
char * yy_mm_dd( Time time, char * buf );
char * ?'ymd( Time time, char * buf );  // short form
char * mm_dd_yy( Time time, char * buf );
char * ?'mdy( Time time, char * buf );  // short form
char * dd_mm_yy( Time time, char * buf );
char * ?'dmy( Time time, char * buf );  // short form

size_t strftime( char * buf, size_t size, const char * fmt, Time time );

forall( ostype & | ostream( ostype ) ) ostype & ?[]( ostype & os, Time time );

H  Clock

H.1  C time

char * cltime( time_t tp );
char * cltime_r( time_t tp, char * buf );
tm * gmtime( time_t tp );
tm * gmtime_r( time_t tp, tm * result );
tm * localtime( time_t tp );
tm * localtime_r( time_t tp, tm * result );

H.2  Clock

struct Clock {
    Duration offset;
};  // virtual clock

// offset from computer real-time
I Pseudo Random Number Generator

Random numbers are values generated independently, i.e., new values do not depend on previous values (independent trials), e.g., lottery numbers, shuffled cards, dice roll, coin flip. While a primary goal of programming is computing values that are not random, random values are useful in simulation, cryptography, games, etc. A random-number generator is an algorithm that computes independent values. If the algorithm uses deterministic computation (a predictable sequence of values), it generates pseudo random numbers versus true random numbers.

All pseudo random-number generators (PRNG) involve some technique to scramble bits of a value, e.g., multiplicative recurrence:

```
rand = 36973 * (rand & 65535) + (rand >> 16);
// scramble bits
```

By dropping bits 63–32, bits 31–0 become scrambled after each multiply. The least-significant bits appear random but the same bits are always generated given a fixed starting value, called the seed (value 0x3e8e36 above). Hence, if a program uses the same seed, the same sequence of pseudo-random values is generated from the PRNG. Often the seed is set to another random value like a program’s process identifier (getpid) or time when the program is run; hence, one random value bootstraps another. Finally, a PRNG usually generates a range of large values, e.g., [0, UINT_MAX], which are scaled using the modulus operator, e.g., prng() % 5 produces random values in the range 0–4.

CV provides a sequential PRNG type only accessible by a single thread (not thread-safe) and a set of global and companion thread PRNG functions accessible by multiple threads without contention.

- The PRNG type is for sequential programs, like corouting:

```
struct PRNG { ... };  // opaque type
void ?{}( PRNG & prng );  // random seed
void ?{}( PRNG & prng, uint32_t seed );  // fixed seed
void set_seed( PRNG & prng, uint32_t seed );  // set seed
uint32_t get_seed( PRNG & prng );  // get seed
uint32_t prng( PRNG & prng );  // [0,UINT_MAX]
uint32_t prng( PRNG & prng, uint32_t u );  // [0,u]
uint32_t prng( PRNG & prng, uint32_t l, uint32_t u );  // [l,u]
uint32_t calls( PRNG & prng );  // number of calls
```

A PRNG object is used to randomize behaviour or values during execution, e.g., in games, a character makes a random move or an object takes on a random value. In this scenario, it is useful to have multiple PRNG objects,
PRNG prng1, prng2;
set_seed( prng1, 1009 ); set_seed( prng2, 1009 );

for ( 10 ) {
    // Do not cascade prng calls because side-effect functions called in arbitrary order.
    sout | nlOff | prng( prng1 ) ; sout | prng( prng1, 5 ) ; sout | prng( prng1, 0, 5 ) | '\t';
    sout | prng( prng2 ) ; sout | prng( prng2, 5 ) ; sout | prng( prng2, 0, 5 ) | nlOn;
}

Figure 13: Sequential PRNG

e.g., one per player or object. However, sequential execution is still repeatable given the same starting seeds for all
PRNGs. Figure 13 shows an example that creates two sequential PRNGs, sets both to the same seed (1009), and
illustrates the three forms for generating random values, where both PRNGs generate the same sequence of values.

- The PRNG global and companion thread functions are for concurrent programming, such as randomizing execution
in short-running programs, e.g., yield( prng() % 5 ).

  void set_seed( uint32_t seed ); // set global seed
  uint32_t get_seed(); // get global seed
  // SLOWER
  uint32_t prng(); // [0,UINT_MAX]
  uint32_t prng( uint32_t u ); // [0,u]
  uint32_t prng( uint32_t l, uint32_t u ); // [l,u]
  // FASTER
  uint32_t prng( thread$ & th ); // [0,UINT_MAX]
  uint32_t prng( thread$ & th, uint32_t u ); // [0,u]
  uint32_t prng( thread$ & th, uint32_t l, uint32_t u ); // [l,u]

The only difference between the two sets of prng routines is performance.

Because concurrent execution is non-deterministic, seeding the concurrent PRNG is less important, as repeat-
able execution is impossible. Hence, there is one system-wide PRNG (global seed) but each CV thread has its own
non-contended PRNG state. If the global seed is set, threads start with this seed, until it is reset and then threads
start with the reset seed. Hence, these threads generate the same sequence of random numbers from their specific
starting seed. If the global seed is not set, threads start with a random seed, until the global seed is set. Hence,
these threads generate different sequences of random numbers. If each thread needs its own seed, use a sequen-
tial PRNG in each thread. The slower prng functions without a thread argument call active_thread internally to
indirectly access the current thread’s PRNG state, while the faster prng functions with a thread argument directly
access the thread through the thread parameter. If a thread pointer is available, e.g., in thread main, eliminating
the call to active_thread significantly reduces the cost of accessing the thread’s PRNG state. Figure 14 shows an
example using the slower/faster concurrent PRNG in the program main and a thread.

CV has an interface to the GMP multi-precision signed-integers [17], similar to the C++ interface provided by GMP. The
CV interface wraps GMP routines into operator routines to make programming with multi-precision integers identical
to using fixed-sized integers. The CV type name for multi-precision signed-integers is Int and the header file is gmp.

void ?{}( Int * this ); // constructor/destructor
thread T {};  
void main( T & th ) { // thread address  
   for ( i; 10 ) {  
      sout | nlOff | prng(); sout | prng( 5 ); sout | prng( 0, 5 ) | ' 	 '; // SLOWER  
      sout | nlOff | prng( th ); sout | prng( th, 5 ); sout | prng( th, 0, 5 ) | nlOn; // FASTER  
   }
}

int main() {  
   set_seed( 1009 );  
   thread $ & th = *active_thread(); // program_main thread_address  
   for ( i; 10 ) {  
      sout | nlOff | prng(); sout | prng( 5 ); sout | prng( 0, 5 ) | ' 	 '; // SLOWER  
      sout | nlOff | prng( th ); sout | prng( th, 5 ); sout | prng( th, 0, 5 ) | nlOn; // FASTER  
   }
   sout | nl;  
   T t; // run thread
}

37301721 2 2 1681308562 1 3
290112364 3 2 1852700364 4 3
732221210 1 3 1775396023 2 3
123981445 2 3 2062557687 2 0
283934808 1 0 672325890 1 3
1414344101 1 3 873424536 3 4
871831898 3 4 866783532 0 1
2142057611 4 4 17310256 2 5
802117363 0 4 492964499 0 0
2346353643 1 3 2143013105 3 2
// same output as above from thread t

Figure 14: Concurrent PRNG

1 void ?( Int * this, Int init );  
2 void ?( Int * this, zero_t );  
3 void ?( Int * this, one_t );  
4 void ?( Int * this, signed long int init );  
5 void ?( Int * this, unsigned long int init );  
6 void ?( Int * this, const char * val );  
7 void ?( Int * this );  
8 Int ?=( Int * lhs, Int rhs ); // assignment  
9 Int ?=( Int * lhs, long int rhs );  
10 Int ?=( Int * lhs, unsigned long int rhs );  
11 Int ?=( Int * lhs, const char * rhs );  
12 char ?=( char * lhs, Int rhs );  
13 short int ?=( short int * lhs, Int rhs );  
14 int ?=( int * lhs, Int rhs );  
15 long int ?=( long int * lhs, Int rhs );  
16 unsigned char ?=( unsigned char * lhs, Int rhs );  
17 unsigned short int ?=( unsigned short int * lhs, Int rhs );  
18 unsigned int ?=( unsigned int * lhs, Int rhs );  
19 unsigned long int ?=( unsigned long int * lhs, Int rhs );  
20 long int narrow( Int val );  
21 unsigned long int narrow( Int val );  
22 int ?=( Int oper1, Int oper2 ); // comparison
int ?==?( Int oper1, long int oper2 );
int ?==?( long int oper2, Int oper1 );
int ?==?( Int oper1, unsigned long int oper2 );
int ?==?( unsigned long int oper2, Int oper1 );
int ?!=?( Int oper1, Int oper2 );
int ?!=?( Int oper1, long int oper2 );
int ?!=?( long int oper1, Int oper2 );
int ?!=?( Int oper1, unsigned long int oper2 );
int ?!=?( unsigned long int oper1, Int oper2 );
int ?<?( Int oper1, Int oper2 );
int ?<?( Int oper1, long int oper2 );
int ?<?( long int oper2, Int oper1 );
int ?<?( Int oper1, unsigned long int oper2 );
int ?<?( unsigned long int oper2, Int oper1 );
int ?<=?( Int oper1, Int oper2 );
int ?<=?( Int oper1, long int oper2 );
int ?<=?( long int oper2, Int oper1 );
int ?<=?( Int oper1, unsigned long int oper2 );
int ?<=?( unsigned long int oper2, Int oper1 );
int ?>>?( Int oper1, Int oper2 );
int ?>>?( Int oper1, long int oper2 );
int ?>>?( long int oper1, Int oper2 );
int ?>>?( Int oper1, unsigned long int oper2 );
int ?>>?( unsigned long int oper2, Int oper1 );

int +?( Int oper ); // arithmetic
int -?( Int oper );
int ~?( Int oper );
int ?&?( Int oper1, Int oper2 );
int ?&?( Int oper1, long int oper2 );
int ?&?( long int oper1, Int oper2 );
int ?&?( Int oper1, unsigned long int oper2 );
int ?&?( unsigned long int oper1, Int oper2 );
int ?&=?( Int * lhs, Int rhs );
int ?|?( Int oper1, Int oper2 );
int ?|?( Int oper1, long int oper2 );
int ?|?( long int oper1, Int oper2 );
int ?|?( Int oper1, unsigned long int oper2 );
int ?|?( unsigned long int oper1, Int oper2 );
int ?|=?( Int * lhs, Int rhs );
int ^?( Int oper1, Int oper2 );
int ^?( Int oper1, long int oper2 );
int ^?( long int oper1, Int oper2 );
int ^?( Int oper1, unsigned long int oper2 );
int ^?( unsigned long int oper1, Int oper2 );
int ?^=?( Int * lhs, Int rhs );
J Multi-precision Integers

Int \oplus? (Int addend1, Int addend2);
Int \oplus? (Int addend1, long int addend2);
Int \oplus? (long int addend2, Int addend1);
Int \oplus? (Int addend1, unsigned long int addend2);
Int \oplus? (unsigned long int addend2, Int addend1);
Int \equiv? (Int * lhs, Int rhs);
Int \equiv? (Int * lhs, long int rhs);
Int \equiv? (Int * lhs, unsigned long int rhs);
Int \equiv? (Int * lhs);
Int \equiv? (Int * lhs);
Int \equiv? (Int * lhs, long int rhs);
Int \equiv? (Int * lhs, unsigned long int rhs);
Int \equiv? (Int * lhs);
Int \equiv? (Int * lhs);
Int \equiv? (Int * lhs, long int rhs);
Int \equiv? (Int * lhs, unsigned long int rhs);
Int \equiv? (Int * lhs);
Int \equiv? (Int * lhs);

Int \ominus? (Int minuend, Int subtrahend);
Int \ominus? (Int minuend, long int subtrahend);
Int \ominus? (long int minuend, Int subtrahend);
Int \ominus? (Int minuend, unsigned long int subtrahend);
Int \ominus? (unsigned long int minuend, Int subtrahend);
Int \ominus? (Int * lhs, Int rhs);
Int \ominus? (Int * lhs, long int rhs);
Int \ominus? (Int * lhs, unsigned long int rhs);
Int \ominus? (Int * lhs);
Int \ominus? (Int * lhs);

Int \otimes? (Int multiplicator, Int multiplicand);
Int \otimes? (Int multiplicator, long int multiplicand);
Int \otimes? (long int multiplicator, Int multiplicand);
Int \otimes? (Int multiplicator, unsigned long int multiplicand);
Int \otimes? (unsigned long int multiplicator, Int multiplicand);
Int \otimes? (Int * lhs, Int rhs);
Int \otimes? (Int * lhs, long int rhs);
Int \otimes? (Int * lhs, unsigned long int rhs);
Int \otimes? (Int * lhs);
Int \otimes? (Int * lhs);

[ Int, Int ] div( Int dividend, Int divisor);
[ Int, Int ] div( Int dividend, unsigned long int divisor);

Int \%? (Int dividend, Int divisor);
Int \%? (Int dividend, unsigned long int divisor);
Int \%? (unsigned long int dividend, Int divisor);
Int \%? (Int dividend, long int divisor);
Int \%? (long int dividend, Int divisor);
Int \%? (Int * lhs, Int rhs);
Int \%? (Int * lhs, long int rhs);
Int \%? (Int * lhs, unsigned long int rhs);

Int \ll\ll? (Int shiften, mp_bitcnt_t shift);
Int \ll\ll? (Int * lhs, mp_bitcnt_t shift);
Int \ll\ll? (Int shiften, mp_bitcnt_t shift);
Int \ll\ll? (Int * lhs, mp_bitcnt_t shift);

Int abs( Int oper );
Int fact( unsigned long int N );
Int gcd( Int oper1, Int oper2 );

// number functions
K  Rational Numbers

Rational numbers are numbers written as a ratio, i.e., as a fraction, where the numerator (top number) and the denominator (bottom number) are whole numbers. When creating and computing with rational numbers, results are constantly reduced to keep the numerator and denominator as small as possible.

// implementation
struct Rational {
  long int numerator, denominator;    // invariant: denominator > 0
}; // Rational

Rational rational();                     // constructors
Rational rational( long int n );
Rational rational( long int n, long int d );
void ?!( Rational * r, zero_t );
void ?!( Rational * r, one_t );
long int numerator( Rational r );       // numerator/denominator getter/setter
long int numerator( Rational r, long int n );
long int denominator( Rational r );
long int denominator( Rational r, long int d );

int ?==?( Rational l, Rational r );      // comparison
int ?!=?( Rational l, Rational r );
int ?<?( Rational l, Rational r );
int ?<=?( Rational l, Rational r );
int ?>?( Rational l, Rational r );
int ?>=?( Rational l, Rational r );

Rational −?( Rational r );              // arithmetic
Rational ?+?( Rational l, Rational r );
Rational ?−?( Rational l, Rational r );
Rational ?•?( Rational l, Rational r );
Rational ?÷?( Rational l, Rational r );

double widen( Rational r );            // conversion
Rational narrow( double f, long int md );

References
#include <gmp.hfa>
int main( void ) {
    sout | "Factorial Numbers";
    Int fact = 1;
    sout | 0 | fact;
    for ( i; 40 ) {
        fact *= i;
        sout | i | fact;
    }
}

Factorial Numbers
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
21 51090942171709440000
22 112400072777676800000
23 2358520167388849766400000
24 62044840173323943936000000
25 155112100433309859840000000
26 4032914611266056355840000000
27 108888694504183521607680000000
28 30488834461171386050150400000000
29 884176199379701954543616000000000
30 265252859812191058636308480000000000
31 82228386541779228177255628800000000000
32 263130836536935016721801216000000000000
33 86833176188118864955181944012800000000000
34 29523279903960414084761860964352000000000000
35 10333147966386144929666651337523200000000000000
36 37199332678990121746799944815083520000000000000000
37 137637530912263450463159795815809024000000000000000000
38 5230226174666011176000722410074291200000000000000000000
39 2039782081197443586402817399028973568000000000000000000000
40 81591528324789773434561126959611589427200000000000000000000000000

Figure 15: Multi-precision Factorials


Index

Italic page numbers give the location of the main entry for
the referenced term. Plain page numbers denote uses of
the indexed term. Entries for grammar non-terminals are
italicized. A typewriter font is used for grammar terminals
and program identifiers.

!? 56
&& 56
*?= 56
++?= 56
—? 56
—colors 4
—deterministic-out 4
—gdb 4
—help 4
—libcfa 4
—linemarks 4
—no-linemarks 4
—no-main 4
—no-prelude 4
—prelude-dir 5
—print 4
—prototypes 4
—statistics 5
—tree 5
—? 56
—CFA 3
—L 4
—N 4
—P 4
—S 5
—XCFA 3 4
—c 4
—d 4
—debug 3
—fenvעים inline 3
—g 4
—h 4
—help 3
—l 4
—lgmp 84
—m 4
—n 4
—nodebug 3
—nohelp 3
—noquiet 4
—p 4
—quiet 3
—std=gnu11 3
—t 5
—~ 12

abs 69
acos 72
acosh 73
Ada 2 6 29
address 18
duality 19
address duality 20
addressing errors 20
| adelete, 68 | cancellation |
| aggregate, 13 | pointer/reference, 19 |
| aggregation, 31, 33 | cascade assignment, 42 |
| Algol68, 19 | case, 7, 10 |
| aliasing, 31 | catch, 65 |
| aligned_alloc, 67 | catchResume, 65 |
| alloc, 67 | cbrt, 71 |
| alloc_set, 67 | ceil, 74 |
| altexpr, 4 | cfa, 3, 4 |
| anew, 68 | CV, 1 |
| arguments | choose, 9, 10, 65 |
| default, 27 | clamp, 69 |
| named, 27 | clear, 43, 45 |
| array, 20 | close, 43, 45 |
| reference, 20 | closing coercion, 40 |
| ASCII, 44, 45 | cmemalign, 67 |
| extended, 44 | Cobol, 2 |
| ascodegen, 4 | codegen, 5 |
| asin, 72 | coercion, 20 |
| asinh, 73 | comma expression, 31 |
| assert.h, 65 | compilation |
| assignment, 20 | cfa, 3 |
| cascade, 42 | compilation option |
| pointer, 18 | –CFA, 3 |
| ast, 4 | –XCFA, 3 |
| astdecl, 4 | –debug, 3 |
| asterr, 4 | –fignuz89-inline, 3 |
| astexpr, 4 | –help, 3 |
| astgen, 5 | –fnodebug, 3 |
| atan, 72 | –fnohelp, 3 |
| atan2, 72 | –fnoquiet, 4 |
| atanh, 73 | –quiet, 3 |
| ato, 68 | –std=gnu11, 3 |
| backward-compatible, 3 | complex, 6 |
| basetypeof, 65 | complex constant, 6 |
| Bilson, Richard, 2 | complex.h, 65 |
| bin, 47 | const hell, 21 |
| bit field, 17 | constant |
| box, 5 | underscore, 5 |
| brand, 57 | continue, 12 |
| break, 7, 12 | continue |
| break | labelled, 12 |
| labelled, 12 | copysign, 75 |
| bsearch, 68 | coroutine, 65 |
| bsearchl, 68 | coroutines, 58 |
| bsearchu, 68 | cos, 72 |
| C linkage, 3 | cosh, 72 |
| C++, –3, 20, 29, 59, 64, 65 | coroutine, 4 |
| C-for-all, 1 | ctype.h, 65 |
| C11, 2 | D, 2, 59, 61 |
| calling convention, 2 | dangling else, 13 |
| calloc, 66 | Dave Till, 2 |
| deadlock, 51 |
INDEX

1 declstats, 4
2 default, 10
3 default arguments, 27
4 delete, 67
5 describe not prescribe, 1
6 disable, 65
7 Ditchfield, Glen, 2
8 div, 31
9 do, 6, 10, 12, 13
10 Doug Lea, 25
11 duality, 19
12 Duff's device, 8–10
13 enable, 65
14 eng, 47
15 enum, 23
16 enumeration, 23
17 enumeration constant, 23
18 eot, 43, 45
19 erf, 73
20 erfc, 73
21 errno.h, 65
22 Esteves, Rodolfo, 2
23 exception, 65
24 excl, 50
25 exp, 70
26 exp2, 70
27 expm1, 70
28 exponentiation
29 floating, 6
30 logarithm, 6
31 operator, 6
32 unsigned integral, 6
33 extended else, 12
34 eye candy, 12
35 fail, 43, 45
36 fallthrough, 9, 10, 65
37 fallthru, 9, 65
38 fdim, 70
39 fenv.h, 65
40 Fibonacci, 58
41 finally, 65
42 fixup, 65
43 flattening coercion, 40
44 float.h, 65
45 floor, 74
46 flush, 43, 45
47 fma, 70
48 fmod, 70
49 for, 6, 10, 12, 13
50 forall, 65
51 Fortran, 2
52 frexp, 75
53 fstream, 1, 4
54 fstream.hfa, 42
55 garbage-collection, 1
56 regional, 1
57 gcc, 3, 4, 57
58 generator, 65
59 getpid, 79
60 Glen Ditchfield, 2
61 gmp, 80, 85
62 gmp.h, 65, 85
63 gnu, 3
64 Go, 2, 58–60
65 goto, 12
66 goto restricted, 12
67 hex, 47
68 hypot, 71
70 I/O
71 common case, 42
72 separator, 43
73 if, 6, 12, 13
74 ifstream
75 ?{}, 45
76 ∨?{}, 45
77 clear, 45
78 close, 45
79 eot, 45
80 fail, 45
81 open, 45
82 read, 45
83 unget, 45
84 ignore, 50
85 ilogb, 71
86 immutable, 18
87 implicit output separation, 45
88 implicit referencing, 21
89 implicit separator, 43
90 incl, 50
91 inheritance
92 enumeration, 24
93 initialization, 20
94 input, 42
95 instruction decoding, 18
96 Int, 80
97 int128, 65
98 interoperability, 2
99 interposition, 5
100 inttypes.h, 65
101 iostream, 1
102 isacquire, 50
103 iso646.h, 65
104 istream
INDEX

1 isacquire, 50
2 Java, 1, 2, 6, 20, 30, 61
3 K-W C, 2
4 keyword, 65
5 labelled
6 break, 12
7 continue, 12
8 lambda, 31
9 ldexp, 75
10 Lea, Doug, 25
11 left, 49
12 level number, 4
13 lexical list, 39
14 lgamma, 73
15 library
16 stream, 42
17 library support, 2
18 limits.h, 65
19 locale, 51
20 stream, 51
21 locale.h, 65
22 log, 71
23 log10, 71
24 log1p, 71
25 log2, 71
26 logarithm, 6
27 logb, 71
28 lrint, 74
29 lround, 75
30 lvalue, 19, 21, 22, 41
31 malloc, 66
32 malloc.h, 65
33 mangling
34 name, 3, 65
35 manipulator, 45, 47, 49
36 bin, 47
37 eng, 47
38 excl, 50
39 hex, 47
40 ignore, 50
41 incl, 50
42 left, 49
43 nl, 46, 47
44 nlOff, 46, 47
45 nlOn, 46, 47
46 nobase, 48
47 nodp, 48
48 oct, 47
49 pad0, 49
50 sci, 47
51 sep, 45
52 sepDisable, 46
53 sepEnable, 46
54 sepGet, 45
55 sepGetTuple, 46
56 sepOff, 46
57 sepOn, 46
58 sepSet, 45
59 sepSetTuple, 46
60 sepTuple, 46
61 sign, 48
62 skip, 49
63 unit, 48
64 upcase, 48
65 wd, 48
66 wdi, 50
67 ws, 49
68 manipulators, 46
69 mass, 34
70 math.h, 65
71 max, 69
72 memalign, 67
73 memcpy, 67
74 memory fault, 18
75 memset, 67
76 Michael Tiemann, 25
77 min, 69
78 modf, 31, 75
79 monitor, 65
80 multi-level exit, 12
81 multi-precision, 80
82 multiple assignment, 41
83 multiple, 34
84 multiple derivation, 16
85 mutex, 65
86 name clash, 3
87 name hiding, 30
88 name mangling, 3
89 named arguments, 27
90 named return values, 26
91 naming, 31
92 nan, 70
93 nearbyint, 75
94 nested control-structure, 12
95 nested locking, 51
96 new, 67
97 newline separation, 46
98 nextafter, 75
99 nexttoward, 75
100 nl, 46, 47
101 nlOff, 46, 47
102 nlOn, 46, 47
103 nobase, 48
104 nodp, 48
105 mangling
null-pointer constant, 18
object, 18
object-oriented, 1
oct, 47
ostream
?{}, 45
^?{}, 45
clear, 45
close, 45
fail, 45
flush, 45
open, 45
write, 45
one, 65
onion, 17
open, 43, 45
opening coercion, 40
operator
  exponentiation, 6
or, 65
ostream
  mutex, 50
  osacquire, 50
out parameter, 31
output, 42
overload, 2, 3
pad0, 49
parametric-polymorphic, 2
parse, 4
particle and wave, 21
plan-9, 24
pointer, 16–18
  assignment, 18
  cancellation, 19
pointer type, 18
posix_memalign, 67
pow, 70
pow(x,y), 6
power of a name, 3
prelude, 3
preprocessor, 23
preprocessor variables
  __CFA_MINOR__, 4
  __CFA_PATCH__, 4
  __CFA__, 4
  __CFA_PATCH__, 4
  __CFORALL__, 4
  __cfallall, 4
prescribing, 17
pretty, 4
prng, 79
productivity, 1, 16
pseudo random-number generators, 79
Python, 1, 43, 61
qsort, 68, 69
quasi-keyword, 65
raii, 50
random, 69
Rational, 84
read, 43, 45
realloc, 66
recursive lock, 50
reference, 18
reference type, 18, 19
referenced type, 18
regional garbage-collection, 1
remainder, 70
remquo, 70
report, 65
resolver, 4
return list, 25
Richard Bilson, 2
rint, 74
Rodolfo Esteves, 2
round, 75
routine object, 21
rproto, 4
rsteps, 4
Rust, 2, 59, 61
rvalue, 19, 21, 22
safety, 1, 16
scala, 1
scalbln, 75
scalbn, 75
sci, 47
seed, 79
sentinel value, 18
sep, 45
sepDisable, 46
sepEnable, 46
sepGet, 45
sepGetTuple, 46
sepOff, 46
sepOn, 46
sepSet, 45
sepSetTuple, 46
sepTuple, 46
setjmp.h, 65
shell pipe-operator, 43
sign, 48
signal.h, 65
sin, 71
sinh, 72
skip, 49
try, 65
tuple, 31, 43
tuple, 4
tuple assignment, 34
tuple expression, 32
tuple type, 32, 39
tuple variable, 33
tuple-returning functions, 33
type hoisting, 30
type nesting, 30
uchar.h, 65
undefined, 18, 26
underscore, 5
unget, 45
ungetc, 43
unistd.h, 65
unit, 48
unsound, 18
upcase, 48
version number, 4
virtual, 65
waitfor, 13, 65
warning, 23
wchar.h, 65
wctype.h, 65
wd, 48
wdi, 50
What goes around, comes around., 2
when, 65
while, 6, 10, 12, 13
with, 65
write, 43, 45, 51
ws, 49
zero, 65