Review of the C Programming Language

Prof. James L. Frankel
Harvard University

Version of 11:55 AM 22-Apr-2018
Copyright © 2018, 2016, 2015 James L. Frankel. All rights reserved.
Reference Manual for the Language

• Required textbook
  • C: A Reference Manual, Fifth Edition by Harbison & Steele

• Has all details of the language
• This is a necessary reference to be able to implement a compiler for the language
Concepts of Language Design and Usage

• This presentation contains many concepts of computer language design and usage
• These concepts are very important in understanding computer languages and also how compilers work
• Feeling at ease with many of these concepts is important to be able to successfully implement a compiler
• This presentation basically follows C89, but may contain some language features that are not present in C89
• Not all C89 language features described in this presentation are part of the language accepted by our class project
Language: Lexical Elements (§2)

• Character Set
• Comments
• Tokens
• Operators and Separators
• Identifiers
• Keywords
• Constants
Language: Operators

• Operators perform operations
  • For example, the “=” (simple assignment) operator stores the value obtained by evaluating the right-hand-side operand into the location designated by the left-hand-side operand

• Operators produce results
  • For example, the result of the “=” (simple assignment) operator is the value obtained by evaluating the right-hand-side operand after being converted to the type of the left-hand-side operand

• Observation: both unary prefix ++ and unary postfix ++ will increment their operand; the difference is the value of the result of the operator
Unary, Binary, and Ternary Operators

• Operators may take one, two, or three operands

• Unary operators take one operand
• Binary operators take two operands
• The ternary operator takes three operands
Prefix, Infix, and Postfix Operators

• Operators that precede their operand are prefix operators
• Operators that appear between their operands are infix operators
• Operators that follow their operand are postfix operators
Expression Parsing

- Expressions are parsed into a tree-like structure
- For example, the following expression $a = b / c$ results in the following tree

```
  =
 /   \
/     \
/      \
/       \
/        \
/         \
/          \
/           \
/             \
/               \
/                 \
/                   \
b               c
```


Expression Evaluation

- When an expression is evaluated, the top-level operator is executed.
- Nested operators are evaluated as determined by the top-level operator in the expression.
  - Most operators evaluate all their operands.
  - Some operators evaluate some of their operands as determined by other operands.
    - Take a look at the “short-circuit” operators: && and ||
    - Also, look at the ternary conditional operator.
“As If” Behavior

• The language stipulates how operators, expressions, statements, etc. behave
• The compiler must implement those rules
• However, the compiler is free to not strictly follow those rules if the program cannot determine whether the rule was followed or not. This is called “as if” behavior because the compiler acts as if all the rules were followed.
• For example, if a program cannot determine if an expression was evaluated or not (i.e., the evaluation of the expression does not effect the state of the program – that is, the result is not needed and there are no side effects), then the compiler does not need to evaluate that expression
Unary Prefix Operators

• Increment and decrement    ++  --
• Size    sizeof
  • Yes, sizeof is an operator!
  • When are parentheses needed?
  • What is the value of    sizeof array    when array is declared as    int array[20]    and int’s are four bytes in length?
• Bitwise not    ~
• Logical not    !
• Arithmetic negation and plus    -  +
• Address of    &
• Indirection    *
• Casting    ( type-name )
  • Because of the matching parentheses, this is defined as a unary operator
Unary Postfix Operators

- Subscripting \( a[k] \)
  - Because of the matching brackets, this is defined as a unary operator
- Function Call \( f(...) \)
  - Because of the matching parentheses, this is defined as a unary operator
- Direct Selection .
- Indirect Selection ->
- Increment and decrement ++ --
Binary Infix Operators

• Multiplicative    *  /  %
• Additive    +  -
• Left and Right Shift    <<  >>
• Relational    <  >  <=  >=
• Equality/Inequality    ==  !=
• Bitwise    &  ^  |
• Logical    &&  ||
• Assignment    =  +=  -=  *=  /=  %=  <<=  >>=  &=  ^=  |=
• Sequential Evaluation    ,
Logical Operands and Results

- Operators that evaluate **operands** for logical values accept 0 to signify false or any non-zero value to signify true
  - These operators are !, &&, ||, and the first operand of ? :

- Operators that produce a logical **result** always result in either 0 (for false) or 1 (for true)
  - These operators are !, &&, ||, <, >, <=, >=, ==, !=
  - No value other than 0 or 1 will be the result of these operators
  - Following the “as if” rule, if the result of these operators is not used as a numeric value, but is used directly in another way (say, as the condition in an if statement), then the true or false result may result in conditional branching but not in a 0 or 1 value
Details of the Binary Infix Logical Operators

• The && and || operators always evaluate their left-hand operand, but only evaluate their right-hand operand when needed to determine the result of the operator; this is sometimes referred to as short-circuit behavior.

• To be clear, these operators are not allowed to evaluate their right-hand operands when not needed to determine the result of the operator.

• That is, the && operator will evaluate its right-hand operand only when its left-hand operand evaluates to true (non-zero).

• The || operator will evaluate its right-hand operand only when its left-hand operand evaluates to false (zero).

• As for all operators, the “as if” rule applies.
Ternary Infix Operator

• Conditional operator  ? :

• Example
  • a ? b : c
  • If a is true, b is evaluated and returned as the result of the conditional operator
  • If a is false, c is evaluated and returned as the result of the conditional operator

• To be clear, this operator is *not allowed* to evaluate the operand that is not required in the description above

• As for all operators, the “as if” rule applies
Associativity (§7.2.1)

• Operators of degree greater than one (*i.e.*, with more than one operand) may be either left- or right-associative

• Associativity determines how operators of the same precedence level are grouped when parentheses are not present

• In the C Programming Language, all binary operator are left-associative except for the assignment operators (includes both simple and compound assignment operators)

• In the C Programming Language, the ternary operator (the conditional operator) is right-associative

• See Table 7-3 on page 205 in Harbison & Steele

• Of course, it is possible to specify associativity by using parentheses
Associativity Examples

• Left-associativity examples
  • $a - b - c$ is equivalent to $((a - b) - c)$

• Right-associativity examples
  • $a = b = c$ is equivalent to $(a = (b = c))$
  • $a ? b : c ? d : e$ is equivalent to $(a ? b : (c ? d : e))$
Precedence(§7.2.1)

• Precedence determines how operators of different precedence levels are grouped when parentheses are not present.
• For example, because multiplicative operators have higher precedence than additive operators and because they both have higher precedence than assignment operators (and because additive operators are left-associative),
  • $a = b + c \times d + e$
    is evaluated as if fully parenthesized as follows
    $(a = ((b + (c \times d)) + e))$

• See Table 7-3 on page 205 in Harbison & Steele.
Overloading (§4.2.4)

• Overloading is the principle that the same symbol (including operators and identifiers) may have more than one meaning

• For example, the – operator is used both as a unary prefix operator and also as a binary infix operator

• Overloading may also be determined by type
  • For example, the – operator is used both for integral subtraction and for floating-point subtraction. These operations are very different even though they have similar mathematical principles that serve as their inspiration

• Overloading may also be determined by context
  • void as a pointer target type means pointer to anything
  • void as the return value in a function declaration means no return value
  • void as the sole type in a cast means discard the value of the expression
Computer Language Operators are Not the Same as Mathematical Operators

• Keep in mind that operators in computer languages are not the same as the similar operator in mathematics

• Several reasons for dissimilarity
  • In mathematics, the number of integral values is infinite – that is, the range of positive and negative integers is unlimited
    • Computers’ integers are constrained in range
  • In mathematics, real numbers are used to represent any real value – that is, they have unlimited range and precision (accuracy to any number of decimal places)
    • Computers’ floating-point numbers are constrained in both range and precision and, in addition because of their internal representation, computers may not be able exactly represent a real value
Type

• Each constant, identifier, sub-expression, and expression has a type
• A type describes the kind of values that are able to be represented
• Taxonomy of types
  • Scalar types
  • Arithmetic types
    • Integral types: char, short, int, long
    • Floating-point types: float, double, long double
  • Pointer types
• Aggregate types
  • Array types
  • Structure types
• Union types
• Function types
• Void types

• The language has a means to declare a type – this is a declaration
  • The type description in a declaration is called a declarator
• The language has rules to describe how types are used
Use of Types

• Some operators may accept operands of a limited subset of types
• The function of an operator may be determined by the type(s) of the operands
  • For example, binary addition is very different for integral values and for floating-point values because they have very different internal representations
• The type of the result of an operator may be determined by the type(s) of the operands
• Overall, we call this the “type calculus”
Lvalues vs. Rvalues (part 1 of 2)

• Some expressions can be used to refer to locations in memory
  • Examples
    • a
    • array[i]
    • node.field
  • These are lvalues

• Other expressions represent values
  • Examples (in addition to those above)
    • a * b
    • funct(a, b, c)
  • These are rvalues
  • All lvalues can represent rvalues, but not all rvalues can represent lvalues

• \( a = 1 + 2; \)
  • Because of associativity and precedence, this is the same as \( (a = (1 + 2)); \)
  • The result of the + operator (adding 1 and 2 in this expression) is not an lvalue
    • It cannot be used to refer to the location of an operand, therefore...
    • For example, it cannot appear on the left-hand-side of an assignment operator
      \( (1 + 2) = a; \)
Lvalues vs. Rvalues (part 2 of 2)

- The description of each operator in Harbison & Steele includes information as to...
  - Whether each operand must be an lvalue (or if an rvalue is acceptable)
  - Whether the result of the operator may be used as an lvalue (or if it may be used solely in those instances that require an rvalue)

- Examples
  - Assignment operators
    - The lhs (left-hand side) must be a modifiable lvalue
    - The result is never an lvalue
  - Address-of operator
    - The operand must be either a function designator or an lvalue designating an object
    - The result is never an lvalue
  - Indirection operator
    - The operand must be a pointer
    - If the pointer points to an object, then the result is an lvalue referring to the object
Layout of Multidimensional Arrays in Memory

• In C, multidimensional arrays are stored in row-major order (i.e., adjacent elements in memory differ by one in their last subscript)

• Thus, a 2-by-3 array of int (two rows, three columns) declared as
  
  int matrix[2][3];

• would be laid out in memory as
  
  matrix[0][0]
  matrix[0][1]
  matrix[0][2]
  matrix[1][0]
  matrix[1][1]
  matrix[1][2]
Language: Declarations (§4)

• Restriction on Where Declarations Can Appear
• Storage Class and Function Specifiers
  • Storage class: auto, extern, register, static, typedef
• Type Specifiers and Qualifiers
  • Qualifiers: const, volatile, restrict (C99)
• Declarators
• Initializers
• External Names
Scope (§4.2.1)

• Identifiers are declared in nested scopes
• Scopes exist in different levels
  • File scope (Top-level identifiers)
    • From declaration point to the end of the program file
  • Procedure scope (Formal parameters in function definitions)
    • From declaration point to end of the function body
  • Block scope (Local identifiers)
    • From declaration point in block to end of the block
  • Entire procedure body (Statement labels)
    • Forward reference to a statement label is allowed
  • Source file (Preprocessor macros)
    • From #define through end of source file or until the first #undef that cancels the definition
Order of Declarations and Statements

• In C89, within any block, all declarations must appear before all statements
• As stated in Compound Statements (§8.4), in C99, declarations and statements may be intermixed
  • In previous versions of C, declarations must precede statements
• In our language which is based most-closely on C89, we will require all declarations to precede statements
Scope Example

```c
int global;

int main(int argc, char *argv[]) { 
    int local;
    {
        int nested_local;
        ...
        if(error_occurred) {
            goto symbol_length_exceeded;
        }
        ...
    } 
    symbol_length_exceeded:
    exit(EXIT_FAILURE);
}
```
Overloading Classes for Names (§4.2.4)

• Preprocessor macro names
• Statement labels
• Structure, union, and enumeration tags
  • Always follow struct, union, or enum
• Component names (“members”)
  • Associated with each structure or union
  • Use always follows either . or ->
• Other names
  • Includes variables, functions, typedef names, and enumeration constants
Visibility (§4.2.2)

• A declaration of an identifier is visible if the use of that identifier will be bound to that declaration
• Declarations may be hidden by successive declarations

• Example:

```c
int i;
int main(void) {
  int i;
  i = 17;
}
```
Extent (or Lifetime or Storage Duration) (§4.2.7)

• In C, procedures and variables occupy storage (memory) during some or all of the time a program is executing
  • Procedures have code in memory
  • Variables have location(s) in memory where their value(s) are stored
• Static storage duration denotes that memory is allocated at or before the program begins execution and remains allocated until program termination
• Local storage duration denotes that memory is allocated at entry to a procedure or block and deallocated at exit from that procedure or block
• Dynamic storage duration denotes that memory is allocated and freed explicitly by the user under program control (e.g., by using malloc and free)
Static and Local Storage Duration

- Procedures have static storage duration
- Global (top-level) variables have static storage duration
- Some variables in blocks may have static storage duration
  - These are declared with the `static` storage class specifier
- Formal parameters have local storage duration
- Some variables in blocks may have local storage duration
  - Automatic variables have local storage duration
  - These either do not have the `static` storage class specifier or they have the `auto` class specifier

- Notes: when the `static` storage class specifier is applied to a procedure, it means that the function name is not externally visible (i.e., not visible outside the current program file)
Storage Class Specifiers (§4.3)

- auto
- extern
- register
- static
- typedef
- Defaults
  - Top-level declarations default to extern
  - Function declarations within blocks default to extern
  - Non-function declarations within blocks default to auto
Type Qualifiers (§4.4)

• **const**
  • A const-qualified lvalue cannot be used to modify an object

• **volatile**
  • An object accessed through a volatile-qualified lvalue can have its value accessed through means not under the compiler’s/run-time’s control

• **restrict**
  • Let’s the compiler know that the object accessed through a restrict-qualified lvalue does not currently have any aliases through which the object can be accessed in the compiler
Position of Type Qualifiers

• const int i; /* means i is a const int */
  /* i cannot be modified */
  /* a value can be assigned to i by using an initializer */

• const int *p1; /* means p1 is a pointer to a const int */
  /* p1 can be modified, but the int pointed to by p1 cannot be modified */

• int *const p2; /* means p2 is a const pointer to an int */
  /* p2 cannot be modified, but the int pointed to by p2 can be modified */

• const int *const p3; /* means p3 is a const pointer to a const int */
  /* neither p3 nor the int pointed to by p3 can be modified */
Modifiable lvalue(§7.1)

• An lvalue that permits modification to its designated object is referred to as a *modifiable lvalue*

• An lvalue that does not permit modification to the object it designates has
  • array type
  • incomplete type
  • a const-qualified type
  • structure or union type one of whose members (applied recursively to nested structures and unions) has a const-qualified type
Declaration vs. Definition

• A declaration of an identifier determines the type of the identifier
• A definition of an identifier sets aside storage for that identifier
• If a procedure/function is being declared or defined...
  • A declaration determines the number and type of parameters and the type of the return value
  • A definition includes the body (i.e., implementation or code) of the function
Language: Types (§5)

- Integer Types
- Floating-Point Types
- Pointer Types
- Array Types
- Enumerated Types
- Structure Types
- Union Types
- Function Types
- The Void Type
- Typedef Names

- Many of the types listed above were explained in the preceding slide labeled “Type”
Side Effects

• For a function, a side effect is any modification to a program’s state that is exhibited other than through the function’s return value
  • Includes: input or output operations, modification of global variables, modification of data structures
Logical Values

• When used as a logical operand,
  • A true value is represented by any non-zero value
  • A false value is represented by a zero value

• When a logical type is produced as a result of an operator,
  • A true value is one (1)
  • A false value is zero (0)
Language: Expressions (§7)

- Objects, Lvalues, and Designators
- Expressions and Precedence
- Primary Expressions
- Postfix Expressions
- Unary Expressions
- Binary Operator Expressions
- Logical Operator Expressions
- Conditional Expressions
- Assignment Expressions
- Sequential Expressions
- Constant Expressions
  - Can be evaluated at compile-time (rather than run-time)

- See preceding slides beginning with the slide labeled “Language: Operators”
Sequence Points (§4.4.5, 7.12.1)

• All previous side effects must have taken place before reaching a sequence point
• No subsequent side effects may have occurred when reaching a sequence point
• Sequence points exist:
  • At the end of a full expression
    • An initializer
    • An expression statement
    • The expression in a return statement
    • The control expressions in a conditional, iterative, or switch statement (incl. each expr. in a for statement)
  • After the first operand of &&, ||, ?, or comma operator
  • After evaluation of arguments and function expr. in a function call
  • At the end of a full declarator
• In Standard C, if a single object is modified more than once between sequence points, the result is undefined
Language: Statements (§8)

• Expression Statements
• Labeled Statements
• Compound Statements
• Conditional Statements
• Iterative Statements
• Switch Statements
• Break and Continue Statements
• Return Statements
• Goto Statements
• Null Statements
Expression Statements (§8.2)

• Treat an expression as a statement
• Discard the result of evaluating the expression

• Expression statements are used when the evaluation of the expression causes one or more desired side effects
Labeled Statements (§8.3)

• A label may be affixed to any statement in order to allow control to be transferred to that statement via a `goto` or `switch` statement.
• There are three kinds of labels:
  • Named labels
  • `case` label (see the Switch Statements (§8.7) slide below)
  • `default` label (see the Switch Statements (§8.7) slide below)

• Example of a named label:

```c
int main(void) {
    if (erroneous_behavior) {
        goto error_occurred;
    }
    ...
    error_occurred:
    ...
}
```
Compound Statements (§8.4)

• Where a single statement could appear, a brace-enclosed list of statements may be used

• Example:

```c
if(expr)
    return;

if(expr2) {
    a = 73;
    b++;
}
```
Conditional Statements (§8.5)

• Allow control flow to be altered based on the value of an expression
• if(expression)
  statement
• if(expression)
  statement
 else
  statement
Iterative Statements (§8.6)

• Allow control flow to loop based on the value of an expression

  • `while(control-expression)`
    
    `statement`

  • `do`
    
    `statement`

  `while(control-expression);`

• `for(initial-clause_{opt}; control-expression_{opt}; iteration-expression_{opt})`

  `statement`
Switch Statements (§8.7)

- Allow control flow to follow a multiway branch based on the value of an expression

- `switch(integral-expression)`
  `switch-statement-body`

- Within the `switch-statement-body`, `case` and `default` labels may appear
  - `case integral-constant-expression`:
  - `default`:

- `case` and `default` labels are bound to the innermost containing `switch` statement

- Control flow will proceed directly through case and default labels
  - A `break` statement is needed to cause a branch to the end of a `switch` statement
Break and Continue Statements (§8.8)

• Cause control flow to branch to a defined location
  • break;
  • continue;

• **break** and **continue** can appear within loops

• **break** can appear within a switch statement

• **break** causes control flow to be transferred just past the closest enclosing loop or switch statement

• **continue** causes control flow to be transferred to the end of the body of the closest enclosing loop (i.e., **while**, **do**, or **for**)
  • From that point, any and all control-expression and loop iteration-expression are reevaluated
Return Statements (§8.9)

• Cause the current procedure or function to return to the caller
• Returns a value, if specified by the declaration of the function

• `return expression_{opt};`
Goto Statements (§8.10)

• Cause control to be transferred to the specified labeled statement
• `goto named-label;`
Functions (§9)

• Function Definitions
• Function Prototypes
• Formal Parameter Declarations
Parameter-Passing Conventions

- Call-by-value (C, Ada)
- Call-by-reference (C++)
- Call-by-result (Pascal, Ada)
- Call-by-value-result (Pascal, Ada)
- Call-by-name (largely archaic) (Algol 60)
  - As if by textual substitution into the function body, but avoiding becoming bound to a declaration within that function
  - Reevaluated – often by using a thunk – each time the parameter is referenced
- Call-by-need (Haskell, R)
  - Similar to call-by-name, but parameter is evaluated only once and memoized
- Call-by-future (C++11 std::future & std::promise)
  - Concurrent evaluation which blocks when value is needed
Function Prototype Declarations and Function Definitions

• Parameter names are optional in function prototype declarations (§9.2)
  • It is better style to include the names to document the purpose of the parameters

• Clearly, parameter names are required in function definitions

• Actual function arguments are converted, as if by assignment, to the type of the formal parameters (see §9)
Array as a Formal Parameter

• If an array is declared as a \textit{formal} parameter, the \textit{leftmost} dimension need not be specified and, if specified, it is ignored (see §4.5.3 & §5.4.3)
  • All other bounds are required for the compiler to properly subscript into the array

• If an array is declared as a \textit{formal} parameter, the \textit{top-level} “array of T” is rewritten to have type “pointer to T” (see §9.3) and that array dimension (if it is specified) is ignored
Array as a Actual Argument

• As mentioned above, when a prototype declaration is present, actual arguments are converted to the formal parameter type, *as if by assignment* (§6.3.2), and if possible

• If an array is passed as an actual argument, the top-level “array of T” is converted to have type “pointer to T” using the same rules as for simple assignment (§7.9.1)