The arithmetic operators are used to compute the values of integer and real (float/double) expressions. The arithmetic operators in C are:

- addition (+)
- subtraction (-)
- multiplication (*)
- division (/)
- negation (-)
- modulus (%)

The modulus operator only takes integer operands. It always returns an integer result. Division of integers truncates the fractional part. For all operations except %, result is of type double if either or both operands are of type double. If one of both operands is of type integer the result is of type integer.

For all operations except %, the result is of type double if either or both operands are of type double. If both operands are of type integer the result is of type integer. The modulus operation A % B returns the remainder of A divided by B. The integer result of A % B is always less than B.

For all operations except %, the result is of type double if either or both operands are of type double. If both operands are of type integer the result is of type integer.

The modulus operation A % B returns the remainder of A divided by B. The integer result of A % B is always less than B.

Logical expressions are used to compare the values of integer and real (floating-point) expressions. The arithmetic operators are used to compute the values of integer and real (floating-point) expressions.

Supplemental Reading

K.N. King, K. Ng, Chapter 8, Section 7.4, 7.5, 7.6
K.N. King, Chapter 4

Reading Assignment

Chapter 12
S. McConnell

Copyright for purposes other than this use and all forms of distribution are expressly prohibited.

University of Toronto

These lecture notes are provided for the personal use of students taking CSC 181F in the Fall term 1999/2000 at the University of Toronto.

CSC 181F Lecture Notes
Increment and Decrement Operators

The increment (++) and decrement (--) operators can be used to efficiently add or subtract one from a variable. Both of these unary operators take an lValue as an operand. They add or subtract one from the value of the variable in memory. The result of the operation is an rValue which can be used like any other rValue.

If the operator occurs before the lValue, then the rValue is the value of the variable after it is incremented or decremented.

If the operator occurs after the lValue, then the rValue is the value of the variable before it is incremented or decremented.

Example:

```c
k = 1;
printf("k is %d\n", ++k); /* prints "k is 2" */
printf("k is %d\n", k); /* prints "k is 2" */
k = 1;
printf("k is %d\n", k--); /* prints "k is 1" */
printf("k is %d\n", k); /* prints "k is 0" */
```

A unary operator has one operand.

Assignment Operator

Assignment is an operator in C.

**Assignment Operator**

lValue = rValue

Assignment is the act of computing the value of some expression and making

that expression the value of some variable.

Assignment is the act of computing the value of some expression and making

that expression the value of some variable.

Example:

```c
lValue = rValue
```

Other kinds of values will be discussed later in the term.

In particular, an lValue is required on the right side of an assignment.

Values can be used anywhere that an expression is allowed.

An lValue is another name for an arbitrary expression.

Expressions can be stored in memory.

Expressions are values.

Values are lValues.
Logical Operators

Logical Operators are used to combine simple logical values into more complicated logical values. The logical operators are:

- `&&` (logical and)
- `||` (logical or)
- `!` (logical not)

The definition for these operators is:

| A | B | A && B | A || B |
|---|---|--------|--------|
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

The relational (comparison) operators in C are used to combine more complicated logical values into logical expressions. The relational operators are:

- `==` (equal)
- `!=` (not equal)
- `>` (greater than)
- `>=` (greater than or equal)
- `<` (less than)
- `<=` (less than or equal)

The relational operators can be used to combine logical values to produce a logical expression. The relational operators compare the values of two expressions and produce a logical expression.
Try to minimize the number of operators in a logical expression to make it easier to understand.

DeMorgan Laws

\[ A \lor B \] replace with \[ \neg (A \land B) \] 
\[ A \land B \] replace with \[ \neg (A \lor B) \] 

Invert Relations

\[ (X = Y) \] replace with \[ X \neq Y \] 
\[ (X < Y) \] replace with \[ X \geq Y \] 

Cancellation

\[ X \land X \land \] replace with \[ X \] 
\[ X \lor X \lor \] replace with \[ X \] 

How to use logical operators

Variables (usually char or int) can store logical values for later use.

For efficiency order operands to produce an early result.

For formatting order operands to produce an easy read.

For order of operands to guarantee safe evaluation of the right operand.

Example: Relational and Logical Operators

Assume that \[ A = 1, B = 2, C = 3 \] and \[ X < Y < Z \].

\[
\text{Example:}
\begin{align*}
\text{if} & \ \neg X \text{ then } \neg Y \\
\text{if } & X \text{ then } \neg Y \\
\text{else if } & Y \\
\text{else} &
\end{align*}
\]

\[
\text{Example:}
\begin{align*}
\text{if} & \ X \lor Y \\
\text{else if } & \neg X \\
\text{else} &
\end{align*}
\]

\[
\text{Example:}
\begin{align*}
\text{if} & \ X \land Y \\
\text{else if } & \neg X \\
\text{else} &
\end{align*}
\]
Operator Precedence

The precedence rules in C are mostly intuitive and sensible.

Arithmetic operators have higher precedence than relational operators, so $A + A - A + A - A + A$ means $(A + A) - (A + A) - (A + A)$ and not $A + (A - A) - (A + A)$.

The precedence rules in C are mostly intuitive and sensible.

Examples: The higher precedence than $+$ and $\times$.

The precedence determines the order in which operators in an expression are evaluated.

**Operator Precedence**

Examples: *hashigherprecedencethan + so $A * B + C$ means $(A * B) + C$ and not $A * (B + C)$.

Arithmetic operators have higher precedence than relational operators so $A + B$ means $(A + B)$ and not $A + (B + C)$.

**Examples:**

A comma operator is used to put several expressions in places where

**Good Style:** Always enclose conditional expressions in parentheses for readability and to avoid operator precedence problems.

**Examples:**

Good Style: The comma operator is used to put several expressions in places where

**Operator**

Good Style: Use the comma operator sparingly when you really need a list of

**Size of Operator**

The size of operator returns the size in bytes of the object.

**Examples:**

**Conditional Expression**

The value of the boolean expression selects one of expressions when

**Good Style:** Always enclose conditional expressions in parentheses for readability and to avoid operator precedence problems.

**Examples:**

The value of the boolean expression 

**Conditional Expression**

Good Style: Use the comma operator sparingly when you really need a list of

**Expression**

expression is evaluated and then becomes the value of the entire

**Conditional Expression**

the comma operator is used to put several expressions in places where

**Conditional Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire

**Expression**

expression is evaluated and then becomes the value of the entire
Reading Assignment

See King, Section 7.5 for full details.

Chapter 15
S. McConnell

Chapter 14
S. McConnell

Supplemental Reading

K.N. King, Section 24.1
K.N. King, Chapter 6
K.N. King, Chapter 5

Operators Precedence

### Type Conversions

C does reasonable automatic type conversions when information is not lost from narrower operand to wider operand.

Examples:
- char -> int
- short -> int or long
- float -> double
- int -> float or double

See King, Section 7.5 for full details.

### Operator Precedence

Here are some of the operators with their precedence levels:

```
Operator       Precedence
--------------
->             1
.              2
/ =           3
< == >        4
- +           5
%             6
&             7
^             8
|             9
&&            10
||            11
? :           12
<< >=        13
== !=        14
&            15
^            16
|            17
<< =         18
|            19
== !=        20
&            21
^            22
|            23
<< =         24
```

Use parentheses if the default associativity isn’t what you want.

**Left-to-right associativity** means the operators will be evaluated from left to right.

```
A + B * C
A + (B * C)
```

**Right-to-left associativity** means the operators will be evaluated from right to left.

```
A / (B / C)
(A / B) / C
```

Precedence will be evaluated in an expression. Operator associativity determines the order in which operations of equal precedence will be evaluated.
is obvious by anyone reading the program.

**Good Style:** Always place matching \{ and \} so that structure of the program is not need to be terminated by a semicolon.

not normally overridden. The grouping behavior is a single statement but does look complicated and confusing. The grouping symbols introduce a new scope where only one statement is allowed. The \{ and \} introduce a new scope where declarations and statements can occur.

| Grouping | { | \{ | Grouping | | Grouping | \} | | | { | \{ | Grouping | \} | | | \} | | | \} |

---

**Scopes of Declaration**

The term scope refers to a place in a program where variables, constants, and types can be declared. Scopes can nest. A scope can be contained in a larger scope.

The inner scope refers to a place in a program where variables, constants, and types can be declared. Scopes can nest. A scope can be contained in a larger scope.

- **Grouping:** \{ and \}
- **Decision making:** if, switch
- **Loops:** while, for, do
- **Control Flow Statements:** break, continue

**Scopes and Visibility**

The scope visibility rule for a programming language determines what names (variables, constants, types, etc.) can be legally used at any given point in a program.

In order for a name to be used at a given point, it must be visible at that point.

The normal scope visibility rule is that names declared in a scope are visible within that scope. They are not visible outside that scope. Variables, constants, types, etc., can be legally used at any given point in a program if they were declared in a scope at or above that point.

**Good Style:**
- Always place matching \{ and \} so that structure of the program is obvious by anyone reading the program.
- Don't declare items with file scope unless they are used to share information between different functions.
- Declare variables, constants, and types in the smallest scope (most local) scope that contains all uses of the item.
- Declare variables, constants, and types in the smallest scope.
- Don't declare items with file scope unless they are used to share information between different functions.

---

**Control Flow Statements**

- **Grouping:** \{ and \}
- **Decision making:** if, switch
- **Loops:** while, for, do
- **Control Flow Statements:** break, continue
assert

The assertion function

The logical expression is evaluated.

assert (logical expression)

If the logical expression is not true, the program is terminated.

The logical expression is evaluated.

assert (logical expression)
Nested if example

```c
if (A < B)
    if (C > D)
        X = C;
    else
        X = D;
else if (A == B)
    {
        X = A;
        B = B + 3;
    }
else if (X != B)
    X = B;
else
    X = A;
```

If statement Examples

```c
if (A == 0)
    printf("THE IMPOSSIBLE HAPPENED\n");
if (X <= Y){
    double T; /* local temporary variable */
    T = X; /* Interchange X and Y */
    X = Y;
    Y = T;
} else
    X -= 1.0;
```

Nested if-else statement

```c
if (logical-expression)
    if (logical-expression)
        ( ? @ ? % A % + ? )
    else
        ( ? @ ? % A % + ? )
else
    ( ? @ ? % A % + ? )
```

**WARNING:** You must use `if { }` and `else { }` more than one statement is required in the true

Logical expression is false if its value is zero, otherwise it is true.

The optional `else` associates with nearest `if`

Statement Example:

```c
if (logical-expression)
    ( ? @ ? % A % + ? )
else
    ( ? @ ? % A % + ? )
```

```c
X = 1.0
else
    X = Y
else
    X = X
```

```
/* Interchange x and y */
T = x;
/* Local temporary variable */
double t = ( x => x )
```

```c
printf("THE IMPOSSIBLE HAPPENED\n")
if ( A == 0 )
    ...
```
for loop

Definition of for loop

(for (expression) while (expression) statement)
(statement) while (expression) statement
for (expression)
(expression)
(statement)

Examples:

for (J=0; J < N; J++)
A[J] = 0;
A[J] = 0;

for (J=N; J > 0; J--)
A[J] = 0;
A[J] = 0;

while and do iteration statements

while (logical-expression) do
statement;
statement;
while (logical-expression);
logical-expression
(required around logical-expression.

Use:
and
if more than one statement is required

Both loop expressions are separated by semicolons.

Expressions are separated by semicolons.

end (logical-expression)

End loop initialization

End loop initialization

End loop initialization

Good Technique:
Use nested ifs as an alternative to complicated logical expressions.

Deeply nested if statements are covered in a nested if statement.

Good Technique: Be sure that all possible cases are covered in a nested if.

WARNING: Be very careful that logical expressions in statements are expressed as a simple logical expression.

Use if statements for controlling program flow when control flow condition can
Iteration Templates

1. Counting Up from $M$ to $N$ by $P$
   - $I = M$
   - for $(I = M; I <= N; I += P)$
     - statement
   - while $(I <= N)$
   - $I += P$

2. Counting Down from $R$ to $S$ by $T$
   - $I = R$
   - for $(I = R; I >= S; I -= T)$
     - statement
   - while $(I >= S)$
   - $I -= T$

Loop control
- `break` causes an immediate exit from the nearest enclosing while or for loop.
- `continue` causes an immediate exit from the nearest enclosing while or for loop.
- `while` statement
  - Evaluate an expression (any of the loop) when the loop continues.
  - `expression` should be a single logical expression. The loop will
    continue if the value of the expression is nonzero.

Examples:
- $I = 0; J = 0; while (I < 100)$
  - $I += 1$
  - $J += 2$

Examples:
- $1 + 2 + 3 + 4$  (result $= 10$)
- $10 * (1 + 2 + 3)$  (result $= 60$)
- $expression$ should be a single logical expression. The loop will
  continue if the value of the expression is nonzero.

Example:
- $sum = 0; i = 0; while (i < 100)$
  - $sum += i$
  - $i += 1$

How to use the for loop

Initialization
- Initialize all variables needed in the loop in expression.

Condition
- The condition used in the expression `expression` to be a logical expression.

Increment
- Increment expression

Examples:
- $i = 0; N = 100$
  - $i += 1$
  - $i -= 1$

Examples:
- $J = 0; N = 100$
  - $J += 2$
  - $J -= 2$

Examples:
- $I = 0; N = 100$
  - $I += 1$
  - $I -= 1$

Examples:
- $J = 0; N = 100$
  - $J *= 2$
  - $J /= 2$

Examples:
- $I = 0; N = 100$
  - $I %= 2$
  - $I **= 2$
switch statement

switch (expression):
    case constExpn: statements...
    default: statements;

Each constExpn is a single constant expression.
The case construct can be repeated as necessary.

WARNING: case clauses FALlTHROUGH from one to the next unless a break
statement is used to exit the switch statement.

The case: construct can be repeated as necessary.
Each case expression is a single constant expression.

Good Style: the last line in every case alternative should always be one of:

- break
- /*FALLTHROUGH ONEXTCASE*/
- return

...
Arrays

type-name identifier[size];

An array is a data structure containing a number of data values, all of which have the same type. The identifier is the name of the elements in the array. size is the number of elements in the array. size can be any positive integer constant. 

WARNING: Valid array subscripts run from 0 to (size-1). identifier[size] is NOT an element of the array.

A particular element in an array can be accessed by specifying a subscript:

identifier[expression];

Each value stored in the array is called an element of the array.

HOW TO USE THE SWITCH STATEMENT

A switch statement is often better than deep-nested if statements. A complex controlling expression in a switch statement is often easier to read and less error prone.

A good switch: Switch is chosen before then deeply nested if statements.

A bad switch: When you need a multi-way decision and the decision can be made on the value of some expression.

Use switch statement when you need a multi-way decision and the decision

Reading Assignment

K.N. King, Chapter 8

switch statement example

switch (1 + 7)
{
    case 4:
        printf("Case 4 
");
        return;
    case 7:
        printf("Case 7 
");
        return;
    case 12:
        printf("Case 12 
");
        return;
    case 19:
        printf("Case 19 
");
        return;
    default:
        printf("Default 
");
        return;
}

GOOD TECHNIQUE: Use a default case that crashes to catch logic errors, e.g.

default: printf("Case statement logic error 
");

GOOD FORMATTING: Use a default case that causes a catch logic errors, e.g.

assert (false);

GOOD DESIGN: A complex controlling expression in a switch statement is clean in an indication of bad program design.
Array Storage Layout

```c
int A [10] ;
```

```c
```

```c
float B [100]  ;
```

```c
```

In C, the name of an array is equivalent to the address of the first element in the array. Subscripts are used to access elements of the array. Note the special case of size determined by initialization list.

- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.
- The `sizeof` operator can be used to determine the size of an array.
- The `sizeof` operator can be used to determine the number of elements in the array.

### Array Declaration Examples

```c
int A[100], B[200];
char message[128+1];
#define B_SIZE(200)
int buffer1[B_SIZE], buffer2[B_SIZE];
/* Use this array with subscripts -128..+127 */
/* Example: S[I+SBIAS] with -128 <= I <= 127 */
#define SBIAS(128)
#define S_SIZE(256)
longint S[S_SIZE];
```

### Warning
- C does not check subscript bounds.
- The size of an array cannot be modified during program execution.

### An array subscript can be any integer expression
- The `A[J+10*N]` subscript is valid.

### An array can be initialized in the definition
- An array can be initialized in the definition.
- An array can be initialized in the definition.

### An array subscript may be any integer expression
- An array subscript may be any integer expression.

```c
long int S[size];
#define S_SIZE(256)

/* Example: S[I+SBIAS] with -128 <= I <= 127 */
/* Use this array with subscripts -128..+127 */

int buffer1[B_SIZE], buffer2[B_SIZE];
char message[128+1];
#define B_SIZE(200)

/* Use this array with subscripts -128..+127 */
/* Example: S[I+SBIAS] with -128 <= I <= 127 */
#define SBIAS(128)
#define S_SIZE(256)
longint S[S_SIZE];
```

### An array subscript may be any integer expression
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
- An array subscript may be any integer expression.
Multidimensional Arrays

A multidimensional array can also be initialized in the definition

Example:

```c
int a[2][3];
```

An array can also be initialized in two stages.

Example:

```c
int M[2][3] = < 1,0,0 > , < 0,1,0 > ;
```

**Good Style:** Always use `=` and `[]` to completely specify multidimensional array initialization.

**WARNING:** You must use separate `[]` for each subscript.

**Initialization:**

- Using a named constant for the size of the array
- Using named constants for the size of the array
- Defined a named type for type parameter

For each array

```c
typedef name parameter size_t size;
```

**HOW TO PARAMETERIZE ARRAYS**

- Defined a named type using `typedef` and `#define` makes it much easier to modify

Parameterized Arrays Example

```c
#define ASIZE(175)
typedef float AElement;
typedef AElement[ASIZE] AType;
```

```c
AType X, Y, Z;
AElement tempSum; /* temp variable used with array */
X[ASIZE-1]=-1.0; /* Mark end of array */

/* Form sum of two arrays */
for (J=0; J<ASIZE; J++)
  tempSum=X[J]+Y[J];
...
Z[J]=tempSum;
```

**WARNING:** You must use separate `[]` for each subscript.

**Initialization:**

- Using a named constant for the size of the array
- Using named constants for the size of the array
- Defined a named type for type parameter

For each array

```c
typedef name parameter size_t size;
```

**HOW TO PARAMETERIZE ARRAYS**

- Defined a named type using `typedef` and `#define` makes it much easier to modify

Parameterized Arrays Example

```c
#define SIZE(100)
double X[SIZE], Y[SIZE], maxY;
```

```c
/* Initialize X and Y */
for (K=0; K<SIZE; K++)
  X[K]=0.0;
  Y[K]=K+1.0;
```

```c
/* Normalize Y into X */
for (maxY=Y[0], K=1; K<SIZE; K++)
  if (Y[K] > maxY)
    maxY=Y[K];
for (K=0; K<SIZE; K++)
  X[K]=Y[K]/maxY;
```
Table Lookup

is a powerful technique for writing compact, efficient, correct programs. In general, a table of constant information is used to implement a mapping function between some input and a fixed corresponding output. Let \( f(x) = [\text{function} = \text{value}] \), where the function between some input \( x \) and a fixed corresponding output \( y \) is defined. In general, a table of constant information is used to implement a mapping function. Table lookup is a powerful technique for writing compact, efficient, correct programs. Examples of table lookup usage include:

- Character classification.
- Data format conversion.
- Unit conversion.
- Data packing and unpacking.

Switch statements:

- Think of table lookup as an alternative to writing a complicated set of if or switch statements.
- Data packing and unpacking.
- Unit conversion.
- Data format conversion.
- Character classification.

Explanations of lookup usage include:

- The mapping function might be one-to-one or many-to-one.
- Indicating that function arguments should not be modified.
- Keeping lookup tables that are used.

Table Lookup

const objects behave exactly like variables except that they can't be assigned:

- const is used to declare objects that resemble variables but are "read-only".
- const is a form of documentation. It says the programmer doesn't intend to change the value.
- The compiler can check that the program doesn't attempt to change the value.
- const is a form of documentation. It says the programmer doesn't intend to change the object.

Examples:

- const is used to declare objects that resemble variables but are "read-only".
- const is a form of documentation. It says the programmer doesn't intend to change the value.

const objects behave exactly like variables except that they can't be assigned:

- const is used to declare objects that resemble variables but are "read-only".
- const is a form of documentation. It says the programmer doesn't intend to change the value.

Multidimensional Array Example

```c
#define ARRAY_SIZE 200

int J, K;
double A[ARRAY_SIZE][ARRAY_SIZE], sum = 0.0;
/* Assume A is given a value here */

/* sum the elements of the array A */
for (J = 0; J < ARRAY_SIZE; J++)
for (K = 0; K < ARRAY_SIZE; K++)
    sum += A[J][K];
```
/* ArabictoRomanNumeralConversion */

const char *unitsDigits[10] =
{
"","I","II","III","IV","V","VI","VII","VIII","IX"
};

const char *tensDigits[10] =
{
"","X","XX","XXX","XL","L","LX","LXX","LXXX","XC"
};

const char *hundredsDigits[10] =
{
"","C","CC","CCC","CD","D","DC","DCC","DCCC","CM"
};

int number;

assert (0 <= number && number <= 1000);

printf "%dasaRomannumeralis%s%s%s\n",number,
hundredsDigits[number/100],
tensDigits[(number/10)%10],
unitsDigits[number%10]);

/* Character Classes */
typedef enum <
illegal, whitespace, newline, letter, digit, special
>= CharClasses;

CharClasses classify[256];

unsigned char ch, ch1;

/* Initialize classify table at runtime */
for (ch=0; ch < 256; ch++)
    classify[ch] = illegal;

classify[\'\'\'] = whitespace;
classify[\'\t\'] = whitespace;
classify[\'\n\'] = newline;

for (ch='a', ch1='A'; ch <= 'z'; ch++, ch1++)
    classify[ch] = classify[ch1] = letter;

for (ch='0'; ch <= '9'; ch++)
    classify[ch] = digit;

classify[\'+\'] = special;

...// more special characters here...

classify[\';\'] = special;

HOW TO Use Table Lookup

Describe the mapping function as a set of augmented, truth tables.

Describe the mapping function as a set of augmented, truth tables.

Describe the mapping function as a set of augmented, truth tables.
If a declaration is given for a function, the declaration must be consistent with the definition of the function.

Good Technique: Always provide a function prototype at the start of a file for any functions that must be used before they are defined so that the compiler has complete information about the function at the point where it is used.

WARNING: If you fail to provide a function declaration before a function is used, the compiler will guess default types for the value returned by the function and the types of function parameters. If the compiler's guess is wrong, you have an ERROR in your program.

Header files are used in C to provide function prototypes and related declarations for functions that are defined and compiled separately. Header files are traditionally named file-name.h.

How to Design Functions

1. Identify computation of an expression that occurs at several places in the program.
2. Identify a group of statements that occur at several places in the program or
   identify computation of an expression that occurs at several places in the program.
3. Function parameters are the link between the function and each place that the function is used.
4. The parameters of a function are also used to identify the arguments of statements.
5. Identify the input values that are required to compute the expression.
6. Return a single result of the solution.
7. Function prototypes should be provided in separate files.

Examples

Good Design - small number of parameters

1. String functions: change case, reverse, blank.

Good Style: Every function should be declared or defined before it is used.

Function

The function declaration provides all the information that is needed to use the function.

\{ ... \}

A function definition is a function header followed by the body of the function.

\{ ... \}

A function declaration is also called a function prototype.

\{ ... \}

The prototype of a function is defined by a semicolon.

\{ ... \}

The name of the function.

\{ ... \}

A function header specifies

- A function declaration and definition.

Functions – Declaration and Definition

A function header specifies

- The name of the function.
- The type of value returned by the function.
- The type and name of the parameter(s) that the function accepts.
- The link between the function and each place that the function is used.

The function declaration provides all the information that is needed to use the function.

A function definition is a function header followed by the body of the function.

A function declaration is also called a function prototype.

The prototype of a function is defined by a semicolon.

The name of the function.

A function header specifies

- A function declaration and definition.

Functions
Function Call & Function Arguments

A function is called by writing the name of the function followed by a list of arguments enclosed in parentheses. If the function has no parameters, you must use an empty set of parentheses ( ).

WARNING: F is not the same as F(). F does not call the function.

The order in which the arguments are written is used to match the arguments to the parameters of the function. Each argument must be of a type that is compatible with the type of the corresponding function parameter.

WARNING: F is not the same as F(). F does not call the function.

The order in which the arguments are written is used to match the arguments to the parameters of the function. Each argument must be of a type that is compatible with the type of the corresponding function parameter.

Variables, types, and constants declared within a function are local to the function. A function is called by writing the name of the function followed by a list of arguments enclosed in parentheses. If the function has no parameters, you must use an empty set of parentheses ( ).

Function Definition

A function definition has the same form as a function declaration except that the body of the function is provided.

Function Parameters

Function parameters are comma-separated lists of declarations of the form type-name identifier. Example: int K, double X, short A[]. The function parameter declarations specify
- The order in which the function expects to receive its parameters
- The type of value associated with each parameter
- The name that will be used to refer to the parameter in the body of the function

The parameters of a function are a comma-separated list of declarations of the form type-name identifier. The function parameter declarations specify
- The order in which the function expects to receive its parameters
- The type of value associated with each parameter
- The name that will be used to refer to the parameter in the body of the function

Function Declaration

The parameters are optional, but the left and right parentheses are required. Good Style: use an explicit void to indicate a function takes no parameters.

Examples: float random(); int maxmum(int X, int Y); double innerProduct(double A[], double B[], int size); void printTable(float table[], short tableSize);
Array Argument Example

```c
double xArray[1000];

int xCount;
...

/* Counting the number of negative values in an array */
int count
negatives( const double A[], const int aSize)
{
    int count = 0, J;
    for (J = 0; J < aSize; J++)
    if (A[J] < 0.0)
        count++;
    return count;
}

xCount = count
negatives(xArray, 1000);
```

Parameter and Argument Example

```c
int K = 3, J = 17;
float Y = 3.1, Z = 123.45;

testFunc( int I, float X);
...
I = 17;
...

void testFunc( int I, float X)
{
    I = 17;
    ...
}
```

Arguments to Functions

```
Array Arguments

As a special mechanism in C, making it easy to pass arrays as arguments to functions

Arguments are Passed By Value

In C, the parameters of a function behave like variables that are local to the

Later we'll see other forms of parameter passing.

The const qualifier can be used to indicate that the function is not intended to

Change the value of the parameter variable.

The corresponding argument, even if it is a variable,

Change the corresponding parameters in the parameter variable do not affect the

A special mechanism in C, makes it easy to pass arrays as arguments to functions.

If the parameter is a multidimensional array, the size in the first dimension may be

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.

The corresponding parameters in the parameter variable do not affect the

In C, the parameters of a function behave like variables that are local to the

Later we'll see other forms of parameter passing.

The const qualifier can be used to indicate that the function is not intended to

Change the value of the parameter variable.

The corresponding argument, even if it is a variable,

Change the corresponding parameters in the parameter variable do not affect the

A special mechanism in C, makes it easy to pass arrays as arguments to functions.

If the parameter is a multidimensional array, the size in the first dimension may be

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.

The corresponding parameters in the parameter variable do not affect the

In C, the parameters of a function behave like variables that are local to the

Later we'll see other forms of parameter passing.

The const qualifier can be used to indicate that the function is not intended to

Change the value of the parameter variable.

The corresponding argument, even if it is a variable,

Change the corresponding parameters in the parameter variable do not affect the

A special mechanism in C, makes it easy to pass arrays as arguments to functions.

If the parameter is a multidimensional array, the size in the first dimension may be

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.

The corresponding parameters in the parameter variable do not affect the

In C, the parameters of a function behave like variables that are local to the

Later we'll see other forms of parameter passing.

The const qualifier can be used to indicate that the function is not intended to

Change the value of the parameter variable.

The corresponding argument, even if it is a variable,

Change the corresponding parameters in the parameter variable do not affect the

A special mechanism in C, makes it easy to pass arrays as arguments to functions.

If the parameter is a multidimensional array, the size in the first dimension may be

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.

The corresponding parameters in the parameter variable do not affect the

In C, the parameters of a function behave like variables that are local to the

Later we'll see other forms of parameter passing.

The const qualifier can be used to indicate that the function is not intended to

Change the value of the parameter variable.

The corresponding argument, even if it is a variable,

Change the corresponding parameters in the parameter variable do not affect the

A special mechanism in C, makes it easy to pass arrays as arguments to functions.

If the parameter is a multidimensional array, the size in the first dimension may be

If the parameter is an array, the name of an array without

May be passed as the corresponding argument.

Can be the size of an any parameter is specified. C shows a compatible any size of any

The function can determine the size of an any argument. So, the size of the array can be

A special mechanism in C makes it easy to pass arrays as arguments to functions.
The return statement is used to return a value from a function.

Good Style: declare a function as returning void if it is not intended to return a useful value.

void declaration. The value returned by a function can be discarded.

| WARNING: Many C compilers do not verify this compatibility. |

The type of the expression should be compatible with the return type.

1. The return statement is used to return a value from a function.

2. If a function returns by running off the end of the function body it: returning a value.

3. The expression is optional, if it is omitted the function returns 

GARBAGE

Function example

```c
int power(int x, int n);
...
main()
{
  int i = 2, j = 10, k;
  ...
k = power(i, j);
  ...
}
int power(int x, int n)
{
  int result = 1;
  while (n-- > 0)
    result = result * x;
  return result;
}
```

More on Scopes in C

The unit of compilation in C is a single source file.

The extern declaration prefix can be used to share declarations across source files. 

The static declaration prefix can be used to limit the scope of a globally declared item.

Each source file introduces a file scope containing all the types, data and functions declared in that file.

Items declared in a local scope of a local block scope are only visible in that scope.

A local block is introduced by the block opened in and the body of each function introduces a distinct local scope.

The body of each function introduces a distinct local scope.

The unit of compilation in C is a single source file.

```c
int K = 5;
int K = 4;
int K = 2;
int K = 1;
int K = 0;

void f(int K)
{
    X
    K = 1;
    Y
    void g(void)
    {
        X
        int K = 2;
        Y
        if (K == 0) 
        {
            X
            int K;
            K = 3;
            Y
            K = 4;
            Y
            void h(void)
            {
                X
                int K = 5;
                Y
            }
            return;
        }
    }
    return;
}
```
The `#include` directive causes the named file to be automatically included in the source program at the point of the directive.

The first form is used to include files from the system libraries. The second form is used to include files from the user's directory. The `#include` statement at the top of the file will include the file named `fname` to be automatically included in the source program at the point of the directive.

The `#include` statement at the top of the file will include the file named `fname` to be automatically included in the source program at the point of the directive.

**WARNING:** Directory path specifications are **not portable**.

These second form is used to include files from the user's directory.

The first form is used to include files from the system libraries.

The source program at the point of the `#include` directive causes the named file to be automatically included in the source program at the point of the directive.

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

The first form is used to include files from the system libraries. The second form is used to include files from the user's directory.

The `#include` statement at the top of the file will include the file named `fname` to be automatically included in the source program at the point of the directive.

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```

**Examples:**

```
#include<stdio.h>
#include"myInterface.h"
#include"C:\no\one\else\can\find\this\file.h"
```