CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 1999/2000 at the University of Toronto

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

Reading Assignment

K.N. King	Chapter 4
K.N. King	Section 7.4, 7.5, 7.6
Supplemental Reading	
S. McConnell	Chapter 8
	Chapter 12

Preview

- Arithmetic Expressions
- Assignment, lValues and rValues
- Logical Expressions
- Conditional expressions
- Operator associativity
- Operator precedence
- Type conversions

Arithmetic Operators

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

$-$	negation	$*$	multiplication
$+$	addition	$/$	division
$-$	subtraction	%	modulus

- For all operators except %, result is of type double if either or both operands are of type double; if both operands are of type integer then the result is of type integer.
  The modulus operator only takes integer operands % always returns a integer result. Division of integers truncates the fractional part.
  The modulus operation A % B returns the remainder of A divided by B.
  It is always true that $0 <= A \% B < B$

Assignment
lValues and rValues

- Assignment is the act of computing the value of some expression and making that expression the *value* of some variable.
- An *lValue* represents an object stored in the memory of the computer. Variables are lValues[a]
- An *lValue* is used in an assignment to indicate where the value of the expression is to be stored in memory.
- An *rValue* is another name for an arbitrary expression.
  rValues can be used anywhere that an expression is allowed.
  In particular an *rValue* is required on the right side of an assignment.

[a] Other kinds of lValues will be discussed later in the term.

Assignment Operator

lValue = *rValue*

- Assignment is an **operator** in C.
  The left operand of this operator must be an *lValue* (usually a variable).
- The right operand of this operator must be an *rValue* .
  The *rValue* can be *any* arbitrary *expression* involving constants, variables, operators and function calls. The result of the assignment *operator* is also an *rValue* .
- Examples:

```
height = 8;
volume = height * length * width;
i = j = k = 0;    /* Equivalent: i = ( j = ( k = 0 )) ; */
12 = i;           /*** WRONG, lValue can't be  a constant ***/
I + j = 0;        /*** WRONG, lValue can't be an expression **
```


Combined Operate and Assign

- For (almost) any binary operators[a]
  lValue **binaryOp=** *rValue*
  is equivalent to:
  lValue = *lValue* **binaryOp** *rValue*
- Examples

```
x += 1 ;        /* x = x + 1 */
y *= x + z;     /* y = y * (x + z) */
i %= 7         /*  i = i % 7  */
```

- **Good Style: Don't** use operate and assign if it makes your program difficult to understand.

[a] *binary* operators are arithmetic operators that require two (left and right) operands, e,g, +

Increment and Decrement Operators ++ and −−

- The increment (++) and decrement (−−) operators can be used to efficiently add or subtract one from a *variable*
  Both of these unary[a] operators take an *lValue* as an operand.
  They add or subtract one from value of the variable *in memory*.
  The result of the operation is a *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:

```
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] A *unary* operator has one operand.

Logical Expression in C

- The C language contains a number of operators that work on *logical values*, i.e. **true** and **false** . Logical values are often called *Boolean* values.
- The *relational operators* compare the values of two expressions and produce a logical value.
- The *logical operators* can be used to combine logical values to produce a logic valued result.
- Historically **false** is represented internally by the value zero and **true** is represented by **any** non-zero value.
- Many programmers use the definitions

```
#define   FALSE   ( 0 )
#define   TRUE    ( 1 )
typedef int Bool ;
```


Relational Operators

- The relational (comparison) operators in C are

==	equal	!=	not equal
>	greater than	>=	greater than or equal
<	less than	<=	less than or equal

  The operators are typically used as

  $expression_{left}$ relationalOperator $expression_{right}$

- The relational operators compare the values of $expression_{left}$ and $expression_{right}$ and produce a logic value describing the result of the comparison.
- All scalar types can be compared. *Only* scalar types can be compared.
- **WARNING: BE REALLY REALLY CAREFUL**

  don't confuse assignment (=) with equality compare (==)

  K = 3 is always true AND CHANGES K !!!

Logical Operators

- The logical operators in C are used to combine simple logical values into more complicated logical values. The logical operators are

&&	logical and
||	logical or
!	logical not

- The definitions for these operators are

A	B	! A	A || B	A && B
false	**false**	**true**	**false**	**false**
false	**true**	**true**	**true**	**false**
true	**false**	**false**	**true**	**false**
true	**true**	**false**	**true**	**true**

Logical Operators are CONDITIONAL in C

Operator	Meaning				
A && B	if A is **true** then	B	else	**false**
A || B	if A is **true** then	**true**	else	B

Note that the right expression (B) is *only* evaluated if it is needed.

Many C programs use the conditional nature of the logical operators.

Example: $0 \leq$ K && K $<$ ARRAY_SIZE && A[K] == 0

WARNING: Be VERY careful to not use the bitwise operators[a] when you want the logical operators.

Operator	Logical	Bitwise
Not	!	~
And	&&	&
Or	||	|

[a] To be discussed later, See King Section 20.1

Example: Relational and Logical Operators

Assume that $x = 1, y = 4, z = 14$.

x <= 1 && y == 3	**false**
x <= 1 || y == 3	**true**
!(x > 1)	**true**
! x > 1	**false** (! !)
!(x <= 1 || y == 3)	**false**
x >= 1 && y == 3 || z < 14	**false**

HOW TO Use Logical Operators

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

  int xTooLow **;**
  xTooLow = X $<$ 0.75 **;**

- Use && , || and ! to combine logical results.
  - ! inverts the logical sense of the expression
  - && produces **true** if BOTH operands are **true**
  - || produces **true** if EITHER operand is **true**

- For efficiency order operands to produce an early result

  put term most likely to be **false** first for &&
  put term most likely to be **true** first for ||

- Use order of operands to *guarantee* safe evaluation of the right operand.

  0 $\leq$ K && K $<$ ARRAY_SIZE && A[K] == 0

- Use parentheses generously to make logical expressions easy to read.
- Use formatting to make long logical expressions easier to read

```
    ( A < 3   ||  !B          ||  C != 17  )
&&  (   !D    ||  E <= 1.375  ||       F   )
&&  (  X < Y  ||  Z >= 13.8                )
```

- Long logical expressions *MAY* indicate *BAD THINKING* make sure each long expression is really necessary. Sometimes computing the logical inverse of an expression and using ! is simpler.

- Try to minimize the number of ! operators in a logical expression to make it easier to understand.

DeMorgan Laws	! A || ! B	replace with	! (A && B)
	! A && ! B	replace with	! (A || B)
Invert Relations	! (X == Y)	replace with	X ! = Y
	! (X $<$ Y)	replace with	X $>$ = Y
Cancellation	! ! X	replace with	X

Conditional Expression

($boolExpn$? $expn_{true}$: $expn_{false}$)

The value of the boolean expression $boolExpn$ selects one of $expn_{true}$ or $expn_{false}$ as the value of the entire construct

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

Examples:

(X $>$ Y ? X : Y) /* max(X, Y) */

(1 $\leq$ N && N $\leq$ LIMIT ? N : 1) /* Bounded N */

Comma Operator

$expression_{left}$ **,** $expression_{right}$

- The comma operator is used to put several expressions in places where normally only a single expression is allowed.
- $expression_{left}$ is evaluated and its value is discarded.
- $expression_{right}$ is then evaluated and then becomes the value of the entire expression (an *rValue*).
- **Good Style:** Use the comma operator sparingly when you really need a list of expressions. Do not use it to write hard to understand programs.

Sizeof Operator

sizeof (object)

- The *sizeof operator* returns the size in bytes of the object.
- In the most common case object is a *type-name* , but object can also be a constant, variable or expression.
- Example:

```
int I ;

sizeof ( int ) ;
sizeof ( I ) ;
sizeof ( 23 ) ;
sizeof ( I + 32768 ) ;
```


Operator Precedence

- Operator Precedence determines the order in which operators in an expression are evaluated. An operator with higher precedence will be evaluated *before* an operator of lower precedence.
- **Examples:** $*$ has higher precedence than $+$ so
  A * B + C means (A * B) + C and not A * (B + C)
  Arithmetic operators have higher precedence than relational operators so
  A + B $<$ C * D means (A + B) $<$ (C * D) and not A + (B $<$ C) * D
- The precedence rules in C are mostly intuitive and sensible.
  Use parentheses when in doubt or to force a particular order of evaluation.
- **WARNING:** Be careful when mixing operators from different precedence classes in an expression.

Operator Associativity

- Operator Associativity determines the order in which operators *of equal precedence* will be evaluated in an expression.
- left −> right associativity means the operators will be evaluated *from left to right* as the occur in the expression so

  A * B / C   means (A * B) / C   and not A * (B / C)

  A - C + 3   means (A - C) + 3   and not A - (C + 3)
- right −> left associativity means the operators will be evaluated *from right to left* as they occur in the expression, so

  I = J = K   means I = (J = K)   and not (I = J) = K
- Use parentheses if the default associativity isn't what you want.

Operator Precedence[a]

Operators	Associativity
`() [] -> .`	left −> right
`! ~ ++ -- + - * & (type) sizeof`	right −> left
`* / %`	left −> right
`+ -`	left −> right
`<< >>`	left −> right
`< <= > >=`	left −> right
`== !=`	left −> right
`&`	left −> right
`^`	left −> right
`|`	left −> right
`&&`	left −> right
`||`	left −> right
`?:`	right −> left
`= += -= *= /= %= &= ^= |= <<= >>=`	right −> left
`,`	left −> right

[a] See King Appendix B. Some of these operators will be discussed later.

Type Conversions

- C does *reasonable* automatic type conversions

  narrower operand −> wider operand

  when information is not lost
- Examples:

  char −> int

  short −> int or long

  float −> double

  int −> float or double
- See King Section 7.5 for full details

Reading Assignment

```
      K.N. King,          Chapter 5

      K.N. King,          Chapter 6

      K.N. King,          Section 24.1

Supplemental Reading

      S. McConnell        Chapter 14

      S. MCConnell        Chapter 15
```

Control Flow Statements

- Scopes of Declaration
- Assert function
- Grouping: { and }
- Decision making: **if**, **switch**
- Loop building: **while**, **do**, **for**
- Loop ending: **break**, **continue**

Scopes of Declaration

- The term *scope* refers to a *place* in a program where variables, constants and types can be declared. Scopes can *nest*, i.e. a scope can be contained in a larger scope.
- Scopes in C include
  - *Grouping scope* - The { and } grouping symbols introduce a new scope where declarations can be made.
  - *File scope* - Each source file defines a scope. Declarations in a source file but *not* in a function are visible to all functions defined in the same file.
  - *Function scope* - The body of a function introduces a new scope. The parameters of the function are automatically included in this scope.

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* for C is that names declared in a scope are only visible within that scope. They are undefined and unavailable outside that scope[a]
- **Good Style:** Declare variables, constants and types in the **smallest scope** (most local) scope that contains all uses of the item.
  Good Style: Don't declare items with file scope unless they are used to share information between different functions.

[a] This rule can be modified using the **extern** and **static** declaration qualifiers that will be discussed later.

Grouping

{
  declarations
  statements
}

- The { and } introduce a new scope where declarations and statements can occur.
- Use { and } to write multiple statements where only one statement is normally allowed. The grouping behaves like a single statement but it does *not* need to be terminated by a semicolon.
- **Good Style: always** place matching { and } so that structure of the program is *obvious* to anyone reading the program.

WHERE TO Locate Variables

- In C you can declare a new variable
  - At the start of any { } grouping. This includes the bodies of functions and inside loops.
  - In a program file outside of any { } grouping.

  Variables declared inside { } can only be used inside the grouping (and any contained groupings).
- Variables declared in a program file (*global variables*) can be used by all functions declared in the same file. this rule can be modified using the **extern** and **static** qualifiers in ways that will be discussed later.
- **Good Style:** Variables should be declared in the **smallest** grouping that contains all *necessary* uses of the variable.
  Reuse of variables (other than obvious temporary variables) should be avoided.
- **Good Style:** do **not** use global variables unless there is *no* simpler alternative.

The assert function

#include < assert.h >

assert(*logical-expression*) **;**

- The *logical-expression* is evaluated.
  - if its value is **true** the assert function does nothing.
  - if its value is **false** the assert function causes program execution to HALT.
    An error message is produced describing the location at which the program halted.
- The assert function is a very efficient and compact way to verify the correct operation of a program *during execution of the program*
  Using assert costs almost nothing extra in time or space.
- assert IS THE PROGRAMMERS FRIEND.

HOW TO Use assert

- Add calls to the assert function *generously* in your program
  - To check for unlikely error conditions.
  - To verify that assumptions that you made in the design of your program are correct.
  - To check that your program is not being used outside of its design limits.
  - To catch programming errors near where they occurred.
  - To verify the integrity of complicated data structures.
  - To detect bad input data before it crashes your program.
- **Good Style:** Each use of assert should be accompanied by a comment describing the purpose of the assert.
- **Good Technique:** Use a lot of asserts to verify and validate your program.

assert Examples

```
/* Algorithm won't converge if R > 1.0 */
assert ( R <= 1.0 ) ;

/* Read three input values */
assert ( scanf("%f%f%f" , & X , & Y , & Z ) == 3 )

/* DEBUG - Is memory getting trashed here ??? */
assert ( 0 <= K && K < ASIZE ) ;
A[ K ] = 3 * K + J ;

/* The IMPOSSIBLE HAS HAPPENED. Should NEVER reach here. */
assert ( false ) ;      /* Beam me up Scotty */
```

if statement

if (*logical-expression*)
    $statement_{true}$ **;**
	if (*logical-expression*)
	    $statement_{true}$ **;**
	else
	    $statement_{false}$ **;**

- The ( ) are **required** around logical-expression.
  Logical expression is **false** if its value is ZERO, otherwise it is **true**
  The optional **else** associates with *nearest* **if**
- **WARNING:** You **must** use $\{$ and $\}$ if more than one statement is required in the true or false parts.

if Statement Examples

```
if ( A == 0 )
      printf("THE IMPROBABLE HAPPENED\n");

if ( X <= Y ) {
      double T ;    /* local temporary variable */
      T = X ;       /* Interchange X and Y */
      X = Y ;
      Y = X ;
 } else
      X -= 1.0 ;
```


Nested if-else statement

if $(expression_1)$
    $statement_1$;
else if $(expression_2)$
    $statement_2$;
else if $(expression_3)$
    $statement_3$;
$\ldots$
else if $(expression_n)$
    $statement_n$;
else
    $statement$;

Nested if example

```
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 ;
```

HOW TO Use the if statement

- Use if statement for controlling program flow when control flow condition can be expressed as a simple logical expression.
- **WARNING:** be very careful that logical expressions in **if** statements are expressed properly. (e.g don't use $<$ if you mean $<=$).
- **Good Technique:** Use nested ifs as an alternative to complicated logical expressions.
- Deeply nested ifs are often an indication of bad program design
- **Good Technique:** Be sure that *all* possible cases are covered in a nested if. You should be able to explain in English the purpose of a nested if statement.

while and do iteration statements

while (*logical-expression*) statement ;	**do** statement ; **while** (*logical-expression*);

- ( ) are required around logical-expression
- Use $\{$ and $\}$ if more than one statement is required
- Both loops execute as long as logical-expression is **true** .

Examples:

J = 0 ; **while** (J $<$ N) A[J++] = 0 ;	J = N - 1 ; **do** A[J$--$] = 0 ; **while** (J $>=$ 0) ;

for iteration statement

for ($expression_{init}$ **;** $expression_{test}$ **;** $expression_{incr}$)
statement ;

- $expression_{init}$ is pre-loop initialization
  $expression_{test}$ is the loop termination test
  $expression_{incr}$ is the per-iteration increment
- ( ) required around the three expressions.
  Expressions are separated by semicolons.

Examples:

for (J = 0 ; J $<$ N ; J++) A[J] = 0 ;	**for** (J = N ; J $>=$ 0 ; J$--$) A[J] = 0 ;

Definition of for loop

for ($expression_{init}$ **;** $expression_{test}$ **;** $expression_{incr}$)
statement

is equivalent to:

$expression_{init}$ **;**
while ($expression_{test}$)
$\{$
statement **;**
$expression_{incr}$ **;**
$\}$ **;**

HOW TO Use the for Loop

- Initialize all variables needed in the loop in $expression_{init}$
  The **,** (comma) operator allows $expression_{init}$ to be a *list* of expressions.
  Example: sum = 0.0 **,** I = 0 **,** limit = 100
- $expression_{test}$ should be a single logical expression. The for loop will continue to iterate as long as this expression has an non-false non-zero value.
- **All** variables that need to be modified from one iteration of the loop to the next should be included in $expression_{incr}$
  The comma operator allows more than one variable to be modified.
  Example: I++ **,** J−− **,** X += 2.5

Loop control

break
continue

---

break causes an immediate exit from the nearest enclosing **while do** or **for** loop

continue causes an immediate start of the next iteration (if any) of the *nearest* enclosing **while** , **do** or **for** loop

Examples:

```
for ( J = 0 ; J < N ; J++ )
    if ( A[J] == X )
        break ;
```

```
for ( J = N ; J >= 0 ; J−− )
    if ( A[J] < 0.0 )
        continue ;
    else
        A[J] -= 0.5 ;
```


HOW TO Iterate

- Iteration is the repeated execution of some sequence of statements
  - Counted iteration is based on some variable taking on a succession of increasing or decreasing values until some final value is reached.
  - Logical iteration is based on the truth of some logical expression
  - More general iteration can combine counting and logical expression testing.
- In C the **while** and **do while** statements are usually used for logical iteration. The **for** statement is usually used for counted iteration and more general iterations.
- There are many iteration patterns (e.g. counting up, counting down that occur repeatedly in programs. You should learn a fixed *template* for each kind of iteration and always use the template when required.

Iteration Templates

Counting Up - from M to N by P

```
I = M ;
while ( I <= N ) {
    statement ;
    I + = P ;
} ;
```

```
for ( I = M ; I <= N ; I + = P )
    statement ;
```

Counting Down - from R to S by T

```
I = R ;
while ( I >= S ) {
    statement ;
    I − = T ;
} ;
```

```
for ( I = R ; I >= S ; I − = T )
    statement ;
```

Logical Iteration on Expression U

```
while ( U )                    for ( ; U ; )
    statement ;                    statement ;
```

Test after iteration / $N\frac{1}{2}$ Loop

```
do                             for ( ... ) {
    statement ;                    ...
while ( U ) ;                      if ( ! U )
                                       break
                                   ...
                               }
```

Infinite Loops

```
while ( true )                 for ( ; true ; )
    statement ;                    statement ;
```


HOW TO Iterate Safely and Sanely

- Except for intentionally infinite loops, each execution of a loop should make some progress toward reaching its limit or termination condition.
- **WARNING:** Beware of **off by one errors** in iteration termination test,
  For example using $<$ instead of <= .
  The iteration will be done once too often or not quite enough.
- **WARNING:** be sure that an iteration and the program that follows it **does nothing gracefully**, i.e. is correct even if the loop executes zero times.
- It should be possible to describe in a simple sentence what each loop does.
  It should be possible to state an *invariant condition*, a logical expression that is true for all iterations of the loop.

- If you're in doubt about a loop terminating successfully, build in a loop sanity test

Paranoid Iteration

```
#define ITER_LIMIT (1000)                      #define ITER_LIMIT (1000)
int iterCount ;                                int iterCount ;
iterCount = 0 ;
while ( ... && iterCount++ < ITER_LIMIT ) {    for (iterCount = 0 , ... ) {
    statement ;                                    ...
} ;                                                assert ( iterCount++ < ITER_LIMIT );
                                                   ...
if ( iterCount >= ITER_LIMIT )                 }
    ...
```


switch statement

```
switch ( expression ) {
        case constExpn : statements
        ...
        default : statements
}
```

- Each constExpn is a single constant expression
- The **case** : construct can be repeated as necessary
  default identifies the optional default statement
- **WARNING: case clauses FALL THROUGH from one to the next unless a break statement is used to exit the switch statement.**
- **Good Style:** the last line in **every** case alternative should **always** be one of:
  - **break** ;
  - /* FALL THROUGH TO NEXT CASE */
  - **return** ;

switch statement example

```
switch ( i + j -7 ) {
    case  3:   k = 1 ;
               break ;
    case  4:
    case  7:
    case 11:
               k = 9 ;
               return ;
    case  12:  k *= 6 ;
          /* FALL THROUGH TO NEXT CASE */
    case  19:  k++ ;
               break ;
    default:   k = 0 ;
               break ;
} ;
```


HOW TO Use the switch statement

- Use switch statement when you need a multi-way decision and the decision can be made on the value of some expression.
- **Good Style:** switch is often better than deeply nested ifs
- A complicated controlling expression in a switch statement is often an indication of bad program design
- **WARNING:** be sure all cases are properly covered in a **switch** statement and that the default does the correct thing for all default cases.
- **Good Technique:** Use a default case that crashes to catch logic errors, e.g.
  default : **printf** ("Case statement logic error\n ");
  assert (**false**) ;

Reading Assignment

K.N. King, Chapter 8

Arrays

type-name identifier [*size*] ;

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

[a] Each value stored in the array is called an *element* of the array

- An array subscript may be any integer expression

```
A[J + 10 * N]
A[J++]
```

- **WARNING: C does NOT check subscript bounds**

```
int A[10], J;
for ( J = 1 ; J <= 10 ; J++ )
    A[J] = 0 ;      /* ERROR A[10] does not exist !! */
```

- An array can be initialized in the definition

```
int A[4] = { 45, 2, 800, 81 };
int B[10] = { 1 , 2 , 3 };      /* the remaining elements are given the value 0 */
int C[10] = { 0 };
int D[] = { 6, 0, 1, 7, 3 };      /* the size may be omitted if an initializer is present */
```


- In C the name of an array is equivalent to the address of the first element in the array.
- Later we'll see how to allocate storage for arrays dynamically (i.e. during program execution).
- The **sizeof** operator can be used to determine the number of elements in the array

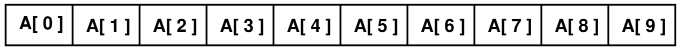
  sizeof (A) / **sizeof** (A [0]);
- Note special case of size determined by initialization list.
- Subscripts with other ranges (e.g. -128 **..** 128) **must** be mapped into 0 **..** size - 1 by adding or subtracting a constant from *all* subscripts.

Array Declaration Examples

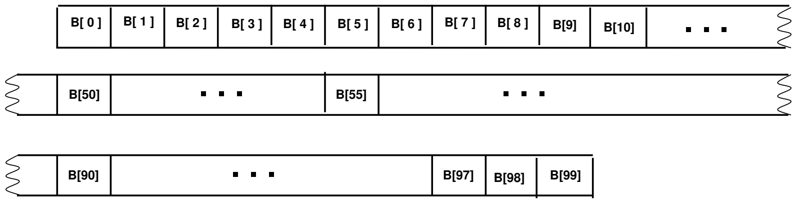
```
int A[100] , B[200] ;
char message[ 128 + 1 ] ;

#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 )
long int S[ S_SIZE ] ;
```


Array Storage Layout

int A [10] ;



float B[100] ;



Array Examples

```
#define SIZE ( 100 )
double X[ SIZE ] , Y[ SIZE ] , maxY ;
/* Initialize X and Y /*
for ( K = 0 ; K < SIZE ; K++ ) {
    X[ K ] = 0.0 ;
    Y[ K ] = K + 1.0 ;
}
/* 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 ;
```


HOW TO Parameterize Arrays

- Defining arrays using **typedef** and **#define** makes it much easier to modify and maintain a program.
- For each array
  - Define a named constant for the size of the array using **#define**
  - Define a named type for the array element using **typedef** .
  - Define a named type for the array.
- All declarations related to the array (including temporary variables) should use the named types defined above.
- All use of the array, especially loops should use the named constant defined above to determine the size of the array.

Parameterized Array Example

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

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 ;
}
```


Multidimensional Arrays

type-name identifier [*size_1*][*size_2*]. . . [*size_n*];

- C stores arrays in **row-major** order, *i.e.,* row 0 first, then row 1, and so forth.

  Example: **int** a[2][3];

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

- An multidimensional array can also be initialized in the definition

  Example: **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* [and] for each subscript.

  M [J , K] is **not** the same as M [J] [K]

Multidimensional Array Example

```
#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 ];
```


Const Qualifier

- **const** is used to declare objects that resemble variables but are "read-only"
- A program can access the value of a **const** object, but can't change
  Example:
  ```
  const int n = 100;
  const int days_per_month[] =
      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  ```
- **const** is a form of documentation. It says the programmer doesn't intend to change the object.
- The compiler can check that the program doesn't attempt to change the value of a **const** object.

- **const** objects behave exactly like variables *except* that they can't be assigned to.
- Use **#define** to create *compile time* constants and **const** to create *run-time* tables of constants.
- The most common uses of **const** are
  - creating table of constants
  - indicating that function arguments should not be modified.

Table Lookup

- *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 $argument$ and a fixed corresponding $value$, i.e.
  $table[argument] = value$
  The mapping function might be one-to-one or many-to-one.
- Examples of table lookup usage include
  - Character classification.
  - Data format conversion.
  - Unit conversion.
  - Data packing and unpacking.
- Think of table lookup as an alternative to writing a complicated set of **if** or **switch** statements.

HOW TO Use Table Lookup

- Describe the mapping function as a set of $argument, value$ pairs.
- If the table is to be stored in an array (we'll see other alternatives later), the $argument$ should be an integer value in the range 0 .. MAX_TABLE . Otherwise all subscripts must be biased by a constant.
- Declare an array of some type compatible with possible $values$ to hold the table.
- Usually the table array is initialized at the point where it is declared, but it could be initialized by the program.

Table Lookup Example

```
/* Arabic to Roman Numeral Conversion */
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 ("%d as a Roman numeral is %s%s%s\n", number,
    hundredsDigits[ number / 100 ] ,
    tensDigits[ ( number / 10 ) % 10 ] ,
    unitsDigits[ number % 10 ] ) ;
```


Example Initialize Character Classification Table

```
/* Character Classes */
typedef enum { illegal, whitespace, newline, letter, digit, special } CharClasses ;
CharClasses classify[ 256 ] ;
unsigned char ch, ch1 ;
/* Initialize classify table at run time */
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 ;
```


Reading Assignment

K.N. King	Chapter	9, 10, 15
K.N. King	Sections	18.2

Functions

- Functions are a mechanism that allows a large program to be subdivided into smaller and more manageable pieces.
- Functions can be developed and tested separately.
- A function abstracts code that is used many places in the program
- It is easier to debug and maintain one copy of code in a function than to debug and maintain many copies of the same code spread throughout the program
- The parameters of a function are the *link* between the function and each place that the function is used.

How to Design Functions

- Identify computation of an expression that occurs at several places in the program
  Identify a group of statements that occur at several places in the program or represent a sub*part of the solution
- Identify the input values that are required to compute the expression
  Identify the variables that a modified by the group of statements
- Good Design - small number of parameters
- Examples
  math functions: sin, cos, sqrt, atan, log
  vector functions: inner product, vector sum
  string functions: change case, remove blanks

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 parameters that the function accepts.
- A function can be specified in a program in two ways.
  - A *function declaration* is a function header followed by a semicolon.
    A function declaration is also called a *function prototype*
  - A *function definition* is a function header followed by the *body* of the function enclosed in { and }
- The function declaration provides all the information that is required to *use* the function.
- **Good Style:** Every function should be declared or defined before it is used.

- 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 functions 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`

Function Declaration[a]

type-name functionName (*parameters*) **;**

- *functionName* is the name of the function.
  type-name is the type of value returned by the function. Use **void** to indicate that a function returns *no* value.
- 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.
- This *declaration* is a **promise** that somewhere else there will be a consistent *definition* for the function.

Examples: **float** random (**void**) **;**

int maxmum(**int** X , **int** Y) **;**

double innerProduct(**double** A[], **double** B[], **int** size) **;**

void printTable(**float** table[], **short** tableSize) **;**

[a] Function declarations are sometimes called *function prototypes*

Function Definition

type-name functionName (*parameters*)
{
declarations
statements
}

- A function definition has the same form as a function declaration except that the body of code that implements the function is supplied.
- Variables, types and constants declared within a function are *local* to the function[a]
- The variables local to a function are created at the instant a function is called, exist until the function returns, at which point they are destroyed.

[a] Except for constants declared using **#define**

Function Parameters

- The parameters of a function are a comma-separated list of declarations of the form *type-name identifer*
  Example: **int** K, **double** X, **short** A[]
- The function parameter declaration specifies
  - 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 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:** Many C compilers do very little checking for correct parameter passing when a function is called. gcc is better than most
  Be **very careful** about
  - The type of each argument.
  - The order of arguments
  - That exactly the right number of arguments has been supplied

Arguments are Passed By Value

- In C, the parameters of a function behave like variables that are local to the function.
- When the function is called, space is allocated for the parameters of the function. Each argument is evaluated and the value of the argument is assigned to the local parameter variable.
- Changes (i.e. assignments) to the parameter variable *do not* affect the corresponding argument, even if it is a variable.
- The **const** qualifier can be used to indicate that the function is not intended to change the value of the parameter variable.
- Later we'll see other forms of parameter passing.

Parameter and Argument Example

```
int K = 3, J = 17 ;
float Y = 3.1, Z =123.45 ;
void testFunc( int I, float X ) {
    ...
    I = 17 ;
}
```

	I	X
testFunc(7, 14.5) ;	7	14.5
testFunc(K, Y + Z) ;	3	126.55
testFunc(J - K , Y*Y) ;	14	9.61

Array Arguments to Functions

- A special mechanism in C makes it easy to pass arrays as arguments to functions
- An array parameter is declared like an array, except that the size of the array can be omitted.
  Example: **int** A[], **int** B[100] , **double** xCoords[]
- The function can't determine the size of an array argument[a] so the size of the array *must* be passed as an additional argument to the function.
- Even is the size of an array parameter is specified, C allows a compatible array of *any* size to be passed as the corresponding argument.
- The argument corresponding to an array parameter is the name of an array *without any subscripts*
- If the parameter is a multidimensional array, the size in the *first* dimension may be omitted, but all the size in all other dimensions *must* be specified.

[a] **sizeof** won't give the right answer in this case

Array Argument Example

```
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 ) ;
```

Function example

```
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;
}
```


return statement

return *expression*

- The return statement is used to return a value from a function
- *expression* is the value returned by the function.
  The expression is optional, **if it is omitted the function returns GARBAGE**
  If a function returns by running off the end of the function body **it returns GARBAGE**
- The type of the expression **should be** compatible with the return type declared for the function.
  WARNING: Many C compilers do not verify this compatability.
- Using an expression statement, the value returned by a function can be discarded.
- **Good Style:** declare a function as returning **void** if it is not intended to return a useful value

More on Scopes in C

- The *unit of compilation* in C is a single source file
- The body of each function introduces a distinct *Local scope*.
  A *local block scope* is corresponds to the text enclosed in { and }
  Items declared in a local scope or a local block scope are only visible in that scope.
- Each source file introduces a *file scope* containing all the types, data and functions declared in that source file. Items declared in a source file outside of a function are visible to all functions declared in the 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 to the source file in which it occurs
- **Good Style:** Use **extern only** when there is no other alternative for sharing variables between files.
  WARNING: Variables shared between files can lead to bad program structure and are a major cause of errors.

Scope Example

```
int K;
void f( int K )
{
    K = 1;
}
void g(void)
{
    int K = 2;
    if ( K > 0 ) {
        int K;
        K = 3;
    }
    K = 4;
}
void h(void)
{
    K = 5;
}
```

extern & static

extern *declaration*

static *declaration*

The **extern** prefix on a declaration declares that the declared items exist in some other file that is a part of the program
Normal usage: declare something in one source file and use **extern** in all other files that need to access it

The **static** prefix on a declaration makes the declaration invisible outside of the file in which it is declared. This can be used to hide declarations including function declarations.
The **static** prefix also causes data items to have a lifetime that is the same as the main program. Variables in a function declared with the **static** prefix retain their values between calls of the function.

Scope example

```
/* File baz.c */

char D ;
int I ;

extern void f(void ) ;
   :

char S ;
   :
```

```
/* File foo.c */

extern char D ;
static int J;

void f(void ) {
    int M ;
    static int K ;
       :
}

extern char S ;
```


Structuring C programs - `.h` and `.c`

- A small C program is contained in a single source file.
  Larger C programs are contained in several source files.
- Technique: each logically separate part of the program should be represented as two distinct source files:
  `fileName.h` should define the interface to the part.
  `fileName.c` should contain the implementation of the part
- Typically the `.h` file contains *only* declarations of data items and functions that are needed to use the part.
- Typically the `.c` file contains private data declarations and the definitions of the functions declared in the `.h` file
- To use the part, only the `.h` file is required.
  The `.c` file can be separately compiled.

Preview: #include

#include < systemFileName>

#include "localFileName"

- 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 users directory.
  localFileName can include directory path specifications.
  WARNING: directory path specifications are **not portable**.
- Examples:

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

Useful system library include files

assert.h	Diagnostic functions
stdio.h	All input and output functions
ctype.h	Character classification functions
string.h	All string processing functions
math.h	Mathematical functions, sin, sqrt, etc.
stdlib.h	Utility functions, conversion, storage allocation
stdarg.h	Variable argument list functions
setjmp.h	Non-local jumps
signal.h	Signals
time.h	Date and time functions
limits.h	Implementation defined limits
float.h	Implementation defined floating point

Include using **#include** `<fileName.h>`

Program Structure Example

```
/* File foo.h */

extern char D ;
extern int I ;

extern void f(void ) ;
:

extern char S ;

:
```

```
/* File foo.c */

#include "foo.h"

char D ;
static int J;
int I ;
char S ;
void f(void ) {
int M ;
static int K ;

:
}
static int g( int N ) {
:
}
```