Scalar Data Types

- Escape Sequences
  - We have already used the `\n` escape sequence, which represents a newline. In addition, C also supports escape character sequences of the form:

    \octal-number and \hex-number

    These translate into the character represented by the octal or hexadecimal number.

`\a`	alert
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab

Scalar Data Types

- Floating-Point Types
  - To declare a variable to hold a real number, you use *float* or *double:*

```
float pi;
double pi_squared;
pi = 3.141;
pi_squared = pi * pi;
```

  - The word *double* stands for *double-precision,* because on many machines it is capable of representing about twice as much *precision* as a *float.*
  - Read the documentation for your particular compiler to discover the range and precision of floats and doubles (these limits are also listed in the `<limits.h>` header file that comes with the ANSI runtime library).

Scalar Data Types

ANSI FEATURE --- long double Type

Intended to provide a greater range and precision than doubles. On many machines, however, long double and double are synonymous.

- Scientific Notation
  - Useful shorthand for writing lengthy floating-point values

Legal	Illegal	Reason
3.141	35	No decimal point or Exponent
.333333	3,500.45	Commas are illegal
0.3	4e	Exponent sign must be followed by a number
3e2		
5E-5	4e3.6	Exponent must be an integer
-3.7e-12		

Scalar Data Types

- Initialisation
  - A declaration allocates memory for a variable, but does not necessarily store an initial value at that location.
  - To initialise a variable, include an assignment expression after the variable name in the declaration:

```
char ch = 'A';
```

- Finding the Address of an Object

```
int main(){
   int j = 1;
   printf("the value of j is : %d\n",j);
   printf("the address of j is: %p\n", &j);
   exit(0);
}
```

Scalar Data Types

- Introduction to Pointers
  - To store address, you need a special type of variable called a *pointer* variable.
  - To declare a *pointer* variable, you precede the variable name with an asterisk:

```
long *ptr; /* A pointer to a long int */
```

```
long *ptr;              long *ptr;
long long_var;          float float_var;
ptr = &long_var;        ptr = &float_var; /* Illegal*/
```

Scalar Data Types

```
int main(){
   int j = 1;
   int *pj;
   pj = &j;
   printf("The value of j is %d\n",j);
   printf("The address of j is: %p\n",pj);
  exit(0);
}
```

The result is:
The value of j is: 1
The address of j is: 3634264

Note it is illegal to try and set the address of an object:

```
&x = 1000; /* is Illegal */
```

Scalar Data Types

- Dereferencing a Pointer
  - The asterisk, in addition to being used in pointer declarations, is also used to *dereference* a pointer (i.e., get the value stored at the pointer address).
  - The following is a roundabout and contrived example that assigns the character 'A' to both `ch1` and `ch2`.

```
int main(){
  char *p_ch;
  char ch1 = 'A', ch2;
  printf("The address of p_ch is %p\n", &p_ch);
  p_ch = &ch1;
  printf("The value stored at p_ch is %p\n", p_ch);
  printf("The dereferenced value of p_ch is %c\n", *p_ch);
  ch_2 = *p_ch;
  exit(0);
}
```

Scalar Data Types

- The expression `*p_ch` is interpreted as: "take the address value stored in p_`ch` and get the value stored at that address."
- This gives us a new way of looking at pointer declarations.

```
float *fp;
```

- means that when `*fp` appears as an expression, the result will be a *float* value.
- The expression `*fp` can also appear on the left hand side of an expression:

```
*fp = 3.15
```

- In this case, we are storing a value (3.15) at the location designated by the pointer `fp`. This is different from

```
fp = 3.15
```

which attempts to store the address 3.15 in `fp`. This, by the way, is illegal since addresses are not the same as integers or floating-point values.

Scalar Data Types

- Initialising Pointers
  - You can initialise a pointer just as you would any type of variable. The initialisation value, however, must be an address.
  - You cannot reference a variable before it is declared.

Legal	Illegal
`int j;`	`int *ptr_to_j = &j;`
`int *ptr_to_j = &j;`	`int j;`

Pointer variables are used frequently with aggregate types, such as arrays and structures.

Scalar Data Types

- typedef
  - C allows you to create your own names for data types with the *typedef* keyword.
  - Syntactically, a typedef is exactly like a variable declaration but preceded by the *typedef* keyword.
  - Semantically, the variable name becomes a synonym for the data type rather than a variable that has memory allocated for it.

```
typedef  long int FOUR_BYTE_INT;
```

There after, `long` int j; and FOUR_BYTE_INT j; are synonyms

Scalar Data Types

Bug Alert --- Confusing typedef with #define

```
#define USHORT unsigned int
typedef unsigned int USHORT;
```

are similar in effect. However, consider trying to declare two pointers to *int* with:

```
#define PT_TO_INT int *
PT_TO_INT p1,p2;
```

This expands to: `int *p1,p2;`

Only `p1` is a pointer, `p2` is an `int`!!

On the other hand:

```
typedef int *PT_TO_INT;
PT_TO_INT p1,p2;
```

defines both `p1` and `p2` as pointers to `int`.

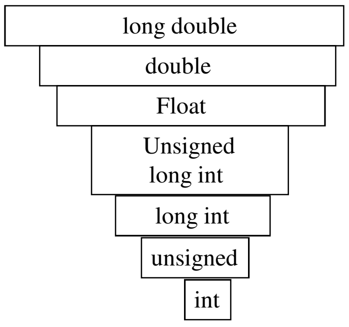
Scalar Data Types

- Mixing Types
  - `num = 3*2.1;` /* num can be any scalar data type except pointer. */
  - `3.0 + 1/2` evaluates to `3.0`! Not `3.5` as expected.
- Implicit (quite, automatic) conversions
  - These occur under 4 circumstances:
    - In assignment statements, the value on the r.h.s. of the assignment is converted to the data type of the variable on the l.h.s. Assignment Conversions.
    - Whenever a `char` or `short int` appears in an expression, it is converted to an `int`. `unsigned chars` and `unsigned shorts` are converted to `int` if the `int` can represent their value; otherwise they are converted to `unsigned int`. These are called integral widening conversions.
    - In an arithmetic expression, objects are converted to conform to the conversion rules of the operator.
    - In certain situations, arguments to functions are converted. This type of conversion will be described later.

Scalar Data Types

- Implicit Conversions in Expressions
  - Each operator has its own rules for operand type agreement, but most binary operators require both operands to have the same type. If the types differ, the compiler converts one of the operands to agree with the other one. To decide which operand to convert, the compiler resorts to the hierarchy of data types and converts the "lower" type to the "higher" type.

Scalar Data Types



If a pair of opnds contains a *long double* the other is converted to a *long double***.**

otherwise, if one of the opnds is a *double* the other is converted to a *double*

otherwise, ...*float*........

otherwise, ...*unsigned long int* ....

otherwise, ...*long int* ....

otherwise, ...*unsigned int* ......

These conversions occur *after* integral widening has taken place.
That is why *int* is at the bottom of the hierarchy.

Scalar Data Types

- Mixing signed and unsigned Types
  - The only difference between *signed* and *unsigned* integer types is the way they are interpreted. They occupy the same amount of storage.
  - A *signed char* with the bit pattern *11101010* has a decimal value of -*22* (2's compliment assumed). An *unsigned char* with the same bit pattern has a decimal value of *234*.
  - Consider *10u -15.The* ANSI standard states that if one of the operands of a binary expression has type *unsigned int* and the other operand has type *int*, the *int* object is converted to *unsigned int*, and the result is *unsigned*.
  - Using this rule *10u-15* has the value *4,294,967,291* (assuming the machine has 4-byte *ints* and uses 2's compliment notation).

Scalar Data Types

- In most cases, the conversion from *signed* to *unsigned* does not cause any problems and goes unnoticed. You do need to be careful though when you use an *unsigned* expression to control program flow.

```
int main(){
  unsigned jj;
  int k;
  if (jj-k < 0) /* Almost certainly a bug since this expression
                 * will never be less than zero
                          */
    foo();
exit(0);
}
```

Scalar Data Types

- Mixing Floating-Point Values
  - No difficulty mixing `float`, `double`, `long double` in expressions. The compiler widens the smaller object of each binary pair to match the wider object.
  - Aside: arithmetic with `float` is much faster than with `doubles`, `long doubles`.
  - Problems with floating-point conversions occur when you assign a larger type to a smaller type. One is loss of precision due to rounding, the other is an overflow condition, which usually leads to a runtime error. No particular behaviour is defined by ANSI standard.

Scalar Data Types

- Mixing Integers with Floating-Point Values

```
#include<stdio.h>

int main(){
   long int j = 2146754677;
   float x;
   x = j;
   printf("j is %d\nx is %10f\n", j, x);
   exit(0);
}
```

int j = 2.5; /* j get value 2 */
int j = -5.8; /* j gets value -5 */
int j = 999999999999.888888; /* overflow */

Scalar Data Types

- Explicit Conversions - Casts

```
j = (float) 2;
```

```
int j = 2, k =3;
float f;
f = k/j; /* value of f is 1.0 */
f = (float) k /j;  /* value of  f is 1.5 */
```

Scalar Data Types

- Enumeration Types
  - Useful when you want to create a unique set of values that may be associated with a variable.
  - Enum types were not part of the original K&R standard.
  - ANSI standard prohibits compilers from halting compilation due to `enum` type conflicts.

```
enum { red, blue, green, yellow } colour;
enum { bright, medium, dark } intensity;
colour = yellow; /* OK */
colour = bright; /* type conflict */
```

```
enum { apples, oranges = 10, lemons, grapes = -5, melons } fruit;
apples = 0, oranges = 10, lemons = 11, grapes = -5, melons = -4
```

Scalar Data Types

- void Data Type
  - Not part of the K&R standard
  - Two purposes:
    - to indicate that a function does not return a value;
    - to declare a generic pointer.

```
void func (int a, int b){
..…
}
```

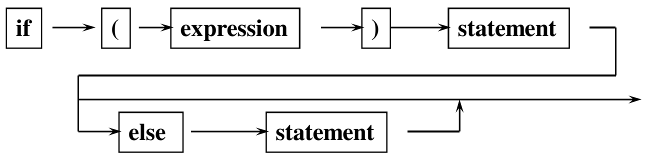
```
extern void func (int, int);
```

```
num = func (x, y); /* should produce an error */
```

Flow of Control

"Begin at then beginning," the King said, very gravely, " and go on till you come to the end: then stop."
- Lewis Carroll, Alice n Wonderland

- Conditional Branching



```
if (x)
  statement1;  /* Executed only if x is non-zero */
else
  statement2; /* Executed only if x is zero */
statement; /* Always executed */
```

Flow of Control

- Testing the validity of input data

```
#include <stdio.h>
#include<stdlib>
#include<math.h>

int main(){
  double num;
  printf( "Enter a non-negative number: ");
/* The %lf conversion specifier indicates a
 * data object of type double
*/
  scanf( "%lf", &num);
  if (num < 0)
    printf( "Input Error: Number is negative. \n");
  else
    printf( "%s%f\n","The square root is: ", sqrt(num));
  exit(0);
}
```

Flow of Control

- Comparison Expressions
  - The are 6 comparison operator (relational operators)

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

  - The value of a relational expression is either 0 or 1. C represents the Boolean values TRUE and FALSE with integers. Zero is equivalent to FALSE, and any non-zero value is considered TRUE.

Flow of Control

Try these: -1 < 0;   0 > 1;   0 == 0;   1 != -1;   1 >= -1;   1 > 10

Bug Alert: Confusing = with == ==

A common mistake made by beginners and experts alike is to use the assignment operator (=) instead of the equality operator (==). For instance:

```
if (j=5)
    do_something();
```

This is syntactically legal since all expressions have a value. The value of the expression `j=5` is 5. Since this is a non-zero value the if expression will always evaluate to true and `do-something`() will always be invoked.

Because Boolean values are represented as integers, it is perfectly legal to write:

```
if (j)
 statement;
```

Flow of Control

A program that reads a character and prints it out if it is a letter of the alphabet and ignores it otherwise. The runtime library function `isalpha`*()* returns a non-zero value if its argument is not a letter

```
#include <ctype.h> /* included for isalpha() */
int main(){
  char ch;
  printf("Enter a character: ");
  scanf("%c", &ch);
  if (isalpha(ch))  /* same  as (isalpha(ch) != 0) */
    printf("%c",ch);
  else
    printf("%c is not an alphabetic char.\n",ch);
  exit(0);
}
```

Common C idiom

```
if (func())
   proceed;
else
   error handler;
```

Flow of Control

Any statement can be replaced by a block of statements, sometimes called a compound statement. A compound statement must begin with a left brace { and end with a right brace }.

```
int main(){
  double num;
  printf(“Enter a non-negative number: ” );
  scanf(“%lf ”, &num);
  if (num< 0 )
    printf(“Input Error: Number is negative.\n”);
  else{
    printf(“%f  squared is %f\n", num, num*num);
    printf(“%f  cubed is %f\n", num, num*num*num);
  }
  exit(0);
}
```

Bug Alert: Missing Braces can lead to syntactically correct programs with undesired meaning

Flow of Control

Nested if Statements

```
int min (int a, int b, int c){
  if (a<b)
    if(a<c) return a;
    else return c;
  else
    if (b<c) return b;
    else return c;
}
```

Bug Alert - The Dangling else

Nested *if* statements create the problem of matching each *else* phrase to the right *if* statement. This is often called the dangling *else* problem. In the *min*() function, for example, note that the first *else* is associated with the second *if***.** The general rule is:

an else is always associated with the nearest previous if.

Each *if* statement, however, can have only one **else** phrase. The next *else* phrase in *min*(), therefore, corresponds to the first *if* because the second *if* has already been matched up.

The final phrase corresponds to the third *if* (written as **else** *if*).

An *else* phrase should always be at the same indentation level as its associated *if*.

Flow of Control

```
int switch_example(char input_arg){
  switch(input_arg){
    case 'A' : return 1;
    case 'B' : return 2;
    case 'B' : return 3;
    case 'B' : return 3;
    default : return -1;
  }
}
```

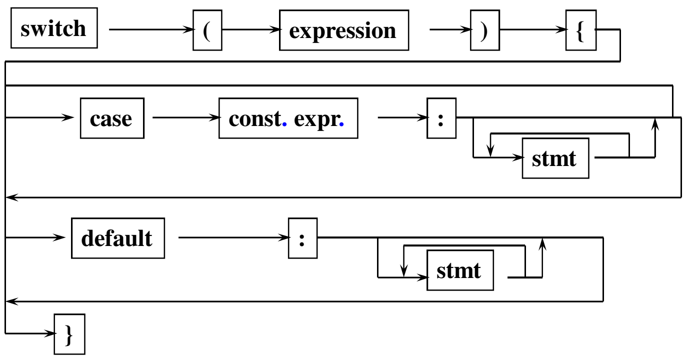
The expression after the switch keyword must be an integral expression: *char*, `short int`, or `long`. (K&R - `int` only).

The expression following the case must be an constant integral expression- that is, it cannot contain a variable.

No two case labels may have the same value.

`default` need not be the last label but it is good style to put it last.

Flow of Control



Program flow continues from the selected case label until another control-flow statement is encountered or the end of the `switch` statement is reached.

`break`, `goto`, `return`.

Flow of Control

The *break* statement explicitly exits the `switch` construct, passing control to the statement following the `switch` statement. Since this is usually what you want, you should almost always include a `break` statement at the end of the statement list following each `case` label

```
void print_error(int error_code){
  switch (error_code){
    case ERR_INPUT_VAL:  printf("Illegal Input\n");
             break;
    case ERR_OP: printf("Illegal Operator\n");
            break;
    default: printf("unknown error\n");
                 break;  /*for consistency */
  }
}
```

To associate a group of statements with more than one case label:

```
switch (arg){
 case '.':
 case ',':
 case ':':
 case ';': return 1;
 default  : return 0;
}
```

Flow of Control

```
#include "err.h"
double evaluate (double op1, char operator, double op2){
  extern void print_error();
  switch (operator){
     case '+' : return op1 + op2;
     case '-'  : return op1 - op2;
     case '*' : return op1 * op2;
     case '/' : return op1 / op2;
     default : print_error (ERR_OP);
                     exit(1);
  }
}
```

The exit`()` function ends the program and returns control to the O.S.
You should have a normal `exit(0)` in your `main`() function.
The way the O.S. reacts to different values returned from `exit`() varies from one implementation to another.

Flow of Control

- Looping
  - `while` statement
  - `do` .. `while` statement
  - `for` statement
- while Statement
  - First the expression is evaluated, if it is a non-zero value, statement is executed. After statement is executed, program control returns to the top of the while statement and the process is repeated.
  - If the expression evaluates to zero (FALSE), the program flow jumps to a point immediately following statement.

$$\begin{tikzcd} \boxed{\textbf{while}} \arrow[r] & \boxed{(} \arrow[r] & \boxed{\textbf{expression}} \arrow[r] & \boxed{)} \arrow[r] & \boxed{\textbf{statement}} \end{tikzcd}$$

Flow of Control

- To read data when you don't know the data type, you can use the `getchar`() function, which read a single character from your keyboard and returns it as an `int`.
- When `getchar`() reaches the end of the input, it returns a special value called `EOF`. `EOF` is a constant name defined in the header file <`stdio.h`>.

```
#include <stdio.h>
main(){
   int ch, num_of_spaces = 0;
   printf("Enter a sentence:\n");
   ch = getchar();
   while (ch != '\n'){
      if (ch ==' ')
         num_of_spaces++;
      ch = getchar();
   }
   printf("The number of spaces is %d.\n",num_of_spaces);
   exit(0);
}
```

Because the operation of adding 1 to a variable occurs so frequently, the C language has a special increment operator called ++.

Flow of Control

$$\begin{tikzcd} \boxed{\textbf{do}} \arrow[r] & \boxed{\textbf{statement}} \arrow[r] & \boxed{\textbf{while}} \arrow[r] & \boxed{(} \arrow[r] & \boxed{\textbf{expression}} \arrow[r] & \boxed{)} \arrow[r] & \boxed{;} \end{tikzcd}$$

```
#include <stdio.h>
int main(){
   int ch, num_of_spaces = 0;
   printf(“Enter a sentence:\n”);
   do{
      ch = getchar();
      if (ch ==‘ ’)
         num_of_spaces++;
  } while (ch != ‘\n’);
   printf(“The number of spaces is %d.\n”, num_of_spaces);
   exit(0);
}
```

Initial assignment to *ch* before the loop is not necessary in this version.

Flow of Control

$$\begin{tikzcd} \boxed{\textbf{for}} \arrow[r] & \boxed{(} \arrow[r] \arrow[rr, bend left] & \boxed{\textbf{expr1}} \arrow[r] & \boxed{;} \arrow[r] \arrow[rr, bend left] & \boxed{\textbf{expr2}} \arrow[r] & \boxed{;} \arrow[r] \arrow[rr, bend left] & \boxed{\textbf{expr3}} \arrow[r] & \boxed{)} \arrow[r] & \boxed{\textbf{stmt}} \end{tikzcd}$$

First , *expr1* is evaluated. This is usually an assignment expression that initialises one or more variables.
Then *expr2* is evaluated. This is the conditional part of the statement

If *expr2* is FALSE, program control exits the for statement and flows to the next statement in the program. If *expr2* is TRUE, *statement* is executed.

After *statement* is executed *expr3* is evaluated. Then the statement loops back to test *expr2* again.

NOTE: *expr1* is evaluated only once, whereas *expr2* and *expr3* are evaluated on each iteration.

Flow of Control

```
for (expr1; expr2; expr3)
    statement;
```

$\Downarrow$

```
  expr1;
  while (expr2){
     statement;
     expr3;
  }
```

```
long int factorial (int val){
int j, fact =1;
for (j=2; j<=val; j++)
   fact = fact * j;
return fact;
}
```

Flow of Control

This function reads a string of digits from the keyboard and produces the string's integer value.

```
#include <stdio.h>
#include<ctype.h>
int make_int(){
   int num = 0, digit;
   digit = getchar();
   for( ; isdigit(digit); digit = getchar()){
      num = num * 10;
      num = num + (digit - '0');
   }
   return num;
}
```

Note: `getchar()` is called twice, which is unfortunate.

Flow of Control

```
#include <stdio.h>
#include<ctype.h>
int make_int(){
   int num = 0, digit;
   for( ; isdigit(digit= getchar());){
      num = num * 10;
      num = num + (digit - ‘0’);
   }
   return num;
}
```

Here `getchar()` is called only once. The assignment to digit and the test of digit are combined in a single expression.

Control Flow

Bug Alert - Off-By-One Errors

A common programming error is to iterate through a loop the wrong number of times. These errors usually do not cause compile-time or runtime errors. Instead the program runs smoothly and produces erroneous results.

Flow of Control

- Null Statements
  - It is possible to omit the body of a loop. This is useful when the loop's work is being performed by the expressions

```
#include <stdio.h>
#include <ctype.h>
void skip_spaces(){
   int c;
   for (c = getchar(); isspace(c); c = getchar())
      ; /* null statement */
   ungetc (c, stdin); /* put the non-space character
                       * back in the buffer
                         */
}
```

It is a good idea to put the semicolon on a separate line to make it more visible since it is potentially misleading.

```
for (;isspace(c=getchar());)
   ;
ungetc(c,stdin);


while (isspace(c=getchar()))
   ;
ungetc(c, stdin);
```

Control Flow

Bug Alert - Misplaced semicolons

A common mistake is to place a semicolon immediately after a control flow statement:

```
if (j ==1);
   j =0;
instead of
if (j==1)
   j = 0;
```

Flow of Control

- Nested Loops:
  - A program to print a multiplication table

```
int main(){
  int j, k;
  printf("     1  2  3  4  5  6  7  8  9  10\n");
  printf("    ------------------------\n");
  for(j=1; j<=10; j++){
    printf("%5d|", j);
    for(k=; k <=10; k++)
      printf("%5d", j*k);
    printf("\n");
  }
  exit(0);
}
```

The `%5d` conversion specifier forces `printf()` to output 5 characters for each number. If the number requires fewer characters, it is preceded with padding spaces.

```
       1    2    3    4    5    6    7    8    9   10
     --------------------------------------------------
 1 |   1    2    3    4    5    6    7    8    9   10
 2 |   2    4    6    8   10   12   14   16   18   20
 3 |   3    6    9   12   15   18   21   24   27   30
 4 |   4    8   12   16   20   24   28   32   36   40
 5 |   5   10   15   20   25   30   35   40   45   50
 6 |   6   12   18   24   30   36   42   48   54   60
 7 |   7   14   21   28   35   42   49   56   63   70
 8 |   8   16   24   32   40   48   56   64   72   80
 9 |   9   18   27   36   45   54   63   72   81   90
10|   10   20   30   40   50   60   70   80   90  100
```

Flow of Control

- The break Statement

```
for (cnt = 0; cnt < 50; cnt++){
   c = getchar();
   if (c == '\n')
      break;
   else /*process character */
.....
}
/* program continues here after
 * break statement
*/
```

`break` causes program control to flow to the next statement after the loop (or `case`).

Use with caution since it causes program control to jump discontinuously to a new place.

There is usually another way to write the code without using `break`.

Too many `break` statements can make the program difficult to follow.

Control Flow: continue

- The continue Statement

```
#include <stdio.h>
#include <ctype.h>
int mod_make_int(){
   int num = 0, digit;
   while ((digit = getchar()) != '\n'){
      if (isdigit(digit) ==0)
         continue;
      num = num * 10;
      num = num + (digit - '0');
   }
   return num;
}
```

The `continue` statement provides a means for returning to the top of a loop earlier than normal.

The `mod_make_int()` function skips non-digit characters. If the input is `A3b-45C,` for example, the function would return `345.`

`continue` statements should be used judiciously since they break up the natural control flow.

Flow of Control

- The `goto` Statement

```
main(){
   int num;
   scanf("%d", &num);
   if(num<0)
      goto bad_val;
   else{
     printf("%f\n", sqrt(num));
     goto end;
   }
   bad_val: printf("Error negative value\n");
   exit(1);
   end: exit(0);
}
```

Few programming statements have produced as much debate as the `goto` *statement***.** Its use in high-level languages is frowned upon.

The purpose of the `goto` *statement* is to enable the program control to jump to some other spot. The destination spot is identified by a *statement label*, which is a name followed by a colon. The *label* must be in the same function as the `goto` statement that references it.

There are specific instances where the `goto` statement makes the code more efficient or enhances readability.

Nevertheless you should not use `goto` unless you have a very good reason for doing so.

Flow of Control

- Infinite Loops
  - Can represent a bug
  - Can be deliberate

```
while (1)
  statement;

for (;;)
  statement
```

These loops cause the program to run continuously until you abort it. On most systems this can be done by typing CTRL-C.

Operators and Expressions

We must either institute conventional forms of expression or else pretend that we have nothing to express -- George Santayana, Soliloquies in England

- There are 4 important types of expression:
  - constant expressions
    - These contain only constant values such as 5; 5+6*13/3.0; 'a'
  - integral expressions
    - After all automatic and explicit conversions these produce a result which has one of the integral types. E.g., k - 'a'; `3+(int)5.0`
  - float expressions
    - After all automatic and explicit conversions these produce a result which has one of the floating-point types. E.g., `3.0;` `3+(float) 4; x/y*5`
  - pointer expressions
    - Expressions that evaluate to an address. These include expressions containing pointer variables, the "address of" operator (&), string literals, and array names. E.g., p; &j; p+1; "abc"; `(char *) 0x000fffff`

Operators and Expressions

Class of operator	Operators in that class	Associativity	Precedence
primary	() ,[] ,->, .	L-to-R	Highest
unary	cast, sizeof, &, *, -, +, ~, ++, --, !	R-to-L	
multiplicative	*, /, %	L-to-R	
additive	+, -	L-to-R	
shift	<<, >>	L-to-R	
relational	<, <=, >, >=	L-to-R	
equality	==, !=	L-to-R	
bitwise AND	&	L-to-R	
bitwise Xor	^	L-to-R	
bitwise OR	|	L-to-R	
logical AND	&&	L-to-R	
logical OR	||	L-to-R	
conditional	? :	R-to-L	
assignment	=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=	R-to-L	
comma	,	L-to-R	Lowest

Operators and Expressions

- Precedence and Associativity
  - Precedence and associativity both affect the way in which operands are attached to operators. Operators with higher precedence have their operands bound to them before operators of lower precedence regardless of the order in which they appear.
    - `2 + 3 * 4;    3 * 4 + 2`
  - In cases where operators have the same precedence, associativity is used to determine the order in which operands are grouped with operators.
    - `a + b – c;   /* L-to-R */    a=b=c; /*R-to-L */`
  - Parenthesis can be used to specify a particular grouping order. They constitute a valuable stylistic function even though they may be redundant from a semantic point of view.

Operators and Expressions

- An important point to understand is that the precedence and associativity have little to do with *order of evaluation* . This refers to the order in which the compiler specifies that operators should be evaluated. For most operators the compiler is free to specify the evaluation order of subexpressions. It may even reorganise the expression so long as the result is not affected.
  - `(2 + 3) * 4;  ----> (2  * 4) + ( 3 * 4);`
- The order of evaluation can have a critical impart on expressions than contain side-effects. Moreover, reorganisation of expressions can sometimes cause overflow conditions.

Operators and Expressions

- Unary Minus Operator
  - The operand can be any integer or a floating-point value. The type of the result is the type of the operand after integral promotions.
  - -e is a shorthand for 0 - e
  - `j = 3 - -x is interpreted as j = (3 - (-x));`
  - Note that the space between the two dashes prevents them from being interpreted as a decrement operator.

Operators and Expressions

- The Remainder Operator `%`
  - Unlike the other arithmetic operators, which accept both integer and floating-point operands, the remainder operator only accepts integer operands.
  - The ANSI standard requires that a equals `a%b + (a/b) *b`

```
void break_line(int interval){
  int c, j = 1;
  while ((c = getchar()) != '\n'){
    putchar(c);
    if (j%interval == 0)
      printf("\n");
    j++;
  }
}
```

Try altering this function so that a newline is not inserted in the middle of a word.

Operators and Expressions

- Arithmetic Assignment Operators
  - The assignment operator has right-to-left associativity, so the expression `a = b = c = d = 1;`

    is interpreted as

    `(a = (b = (c = (d = 1))));`
  - Note that each assignment may cause quiet conversions, so,

```
int j;
double f;
f = j = 3.5;
```

  - assigns the truncated value 3 to both f and j. On the other hand,

```
j= f = 3.5;
```

  - assigns `3.5` to f and 3 to j.

Operators and Expressions

- In addition to the simple assign operator, the C language supports five additional assignment operators that combine assignment with each of the arithmetic operations. For example:

`j = j * 5;` can be written `j *= 5;`

Operators and Expressions

Operator	Symbol	Form	Operation
Assign	=	a = b	put the value of b into a
add-assign	+=	a += b	put the value of a+b into a
subtract-assign	-=	a -= b	put the value of a-b into a
multiply-assign	*=	a *= b	put the value of a*b into a
divide-assign	/=	a /= b	put the value of a/b into a
remainder-assign	%=	a %= b	put the value of a%b into a

•Sometimes produce more efficient code since some machines have special machine instructions to perform arithmetic-assign combinations.

•If the left hand operand contains a side-effect, the side-effect occurs only once. This feature has special significance for arrays.

Operators and Expressions

- The assign operators have relatively low precedence. The following two expressions are not the same.

$$\begin{tikzcd}
\texttt{j = j * 3 + 4;} \arrow[d] & \texttt{j *= 3 + 4;} \arrow[d] \\
\texttt{j = ((j * 3) + 4);} & \texttt{j *= (3 + 4);} \arrow[d] \\
 & \texttt{j = (j * (3 + 4));}
\end{tikzcd}$$

Operators and Expressions

Bug Alert - Integer Divisions and Remainder

When both operands of the division operator (/) are integers, the result is an integer If both operands are positive, the division is inexact , the fractional part is truncated.

If either operands is negative, however, the compiler is free to round the result either up or down:

-5/2 evaluates to -2 or -3; -1/-3 evaluates to 0 or -1;

By the same token, the sign of the remainder operations is undefined by the C standard:

-5 % 2 evaluates to 1 or -1;

Obvious you should avoid division and remainder operations with negative numbers since the results can vary from one compiler to another.

To avoid the problem during division, cast the operands to float or double. Even if the result is assigned to an integer, you are guaranteed that the compiler will convert to an integer by truncating the fractional part:

```
int j = (float) 5/-2; /* j get the value -2 */
```

Although this is a portable solution, it is expensive since it requires the CPU to perform floating -point arithmetic.

The sign of the remainder is more difficult to circumvent because its operands must be integer.

Use `j = abs(k%m);` /* if you always want the sign to be positive. Otherwise use the `div()` function to calculate the quotient and remainder in a portable manner.

Operators and Expressions

- Increment and decrement Operators
  - The are two versions of each increment and decrement operator --- called the prefix versions and the postfix version. There are subtle differences between both types which can prove to be very important.

`a++`	get value of a, increment a.
`++a`	increment a, get value of a
`a--`	get value of a, decrement a.
`--a`	decrement a, get value of a

  - In many cases you are interested in the side effect, not in the result of the expression. In these instances, it does not matter which operator you use. For example, as a stand-alone assignment, or as the third expression in a for loop, the side effect is the same whether you use the prefix or postfix version.

Operators and Expressions

- You need to be careful when you use the increment and decrement operators within an expression

```
void break_line(int interval){
  int c, j = 0;
  while ((c = getchar()) != '\n'){
    if (j++ % interval) == 0)
      printf("\n");
    putchar( c );
  }
}
```

- If we were to use the prefix operator, the function would break the first line one character early.

Operators and Expressions

- Precedence of Inc. and Dec. Operators
  - The increment and decrement operators have the same precedence, but bind from right to left. So the expression,
    - `--j++` is evaluated as `--(j++)`. This expression is illegal because j++ is not a memory location as required by the -- operator.
  - In general, you should avoid using multiple increment or decrement operators together.

```
int  j = 0, m = 1, n = -1;
```

Expression	Equivalent Expression	Result
m++ - --j	(m++) - (--j)	2
m += ++j * 2	m = (m + ((++j) * 2))	3
m++ * m++	(m++) * (m++)	implementation dependent

Operators and Expressions

Bug Alert - Side Effects

The increment and decrement operators, and the assignment operators, cause side effects. That is, they not only result in a value, but they change the value of a variable as well.

It is not always possible to predict the order in which the side effects occur:

```
x = j * j++;
```

The C language does not specify which multiplication operand is to be evaluated first. Thus, `if j = 5` (say) there are two possible answers: 25 and 30.

Statements such as this one are not portable and should be avoided. The side effect problem also crops up in function calls because the C language does not guarantee the order in which arguments are evaluated:

```
f(a, a++);
```

is not portable.

If you use a side effect operator in an expression, do not use the affected variable anywhere else in that expression.

`x = j * j; ++j;` disambiguates the first example.

Operators and Expressions

- Comma Operator
  - a,b; /* evaluate a, evaluate b, result is b*/
  - The comma operator allows you to evaluate two or more distinct expressions wherever a single expression is allowed. The result is the value of the rightmost operand. The comma operator is one of the few operators for which the order of evaluation is specified. The compiler must evaluate the left-hand operand first.
  - Although the comma operator is legal in a number of situations, it leads to confusing code in many of them. By convention, it is primarily used in the first and last expressions of a *for statement*.

```
for (j=0,k=100; k-j>0;  j++,k--);
```

Operators and Expressions

- There is a temptation to fit as much as possible into the *for* expressions. For example, the break_line() function could also be written :

```
void break_line(int interval){
  int c, j;
  for(c = getchar(), j=0; c != EOF; c = getchar(), putchar(c), j++)
    if (j%interval == 0)
      printf("\n");
}
```

This is more compact but not better since it is harder to read.

Operators and Expressions

- Relational Operators

  We look at some of the ramifications of the precedence and associativity rules when applied to these operators.

  The relational operators have a lower precedence than the arithmetic operators and have a left-to-right associativity.

`a + b * c < d / f` is evaluated as `(a + (b * c)) < ( d / f))`

int j = 0, m = 1, n= -1; float x = 2.5, y=0.0;

Expression	Equivalent Expression	Result
j>m	j >m	0
m/n<x	(m/n)<x	1
j<=m>=n	((j <=m) >=n)	1
j<=x==m	((j<=x) ==m)	1
-x+j==y>n>m	((-x)+j) ==((y>n)>=m)	0
x+=(y>=n)	x = (x+(y>=n))	3.5
++j==m!=y*2	((++j) ==m) != (y*2)	1

Operators and Expressions

Bug Alert - Comparing Floating-Point Values

It is very dangerous to compare floating-point values for equality because floating-point representations are inexact for some numbers.

Although algebraically TRUE the following will evaluate to FALSE on most computers:

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0) == 1.0
```

You should refrain from using strict equality comparisons with floating-point types.

Operators and Expressions

- Logical Operators
  - In Algebra, the expression: x < y < z is true is true if y is greater than x and less than z. Unfortunately the expression has a very different meaning in C since it is evaluated as : (x < y) < z
  - The subexpression (x<y) is evaluated first and results in either 0 or 1. So in C, the expression is true if x is less than y and z is greater than 1, or if x is not greater than y and z is greater than 0.
  - To obtain the algebraic meaning you must rewrite the expression using relational operators.

Operators and Expressions

- Logical AND (&&), OR (||) and Negation (!)
  - The && operator returns TRUE only if both expressions are TRUE. The OR operator return TRUE if either expression id TRUE.
  - To test whether y is greater than x and less than z, you would write
    (x<y) && (y <z)
  - The logical negation operator (!) takes only one operand. If the operand id TRUE it returns FALSE, if the operand is FALSE, it returns TRUE.
- The operands to the logical operators may be integers or floating-point objects.
  - 1 && -5; 0.5 && -5.

Operators and Expressions

- The compiler must ensure that the operands are evaluated from left to right. Moreover, the compiler is guaranteed not to evaluate an operand if it's unnecessary:

```
if((a!=0) && (b/a ==6.0)
```

- If a equals 0, the expression `b/a == 6.00` will not be evaluated. This rule can have unexpected consequences when one of the expressions contains a side effect.

Operators and Expressions

Bug Alert - Side effects in relational expressions

Relational operators (and the conditional and comma operators) are the only operators for which the order of evaluation is defined.

Furthermore, a compiler will cause the evaluation of only as much of a relational expression as is necessary to determine the result.

This can cause a problem if some of the expressions contain side effects.

```
if ((a < b) && ( c == d++))
```

In this case, d is only incremented when a is less than b. This may or may not be what the programmer intended. In general, you should avoid using side effect operators in relational expressions.

Operators and Expressions

- Bit-Manipulation Operators
  - The bit-manipulation operators enable you to access specific bits within an object, and to compare the bit sequences of pairs of objects. The operands for all the bit-manipulation operators must be integer.

```
>>,     <<,     &,     |,     ^,     ~
```

Shifting to the left is equivalent to multiplying by powers of 2 x << y is equivalent to $x * 2^y$

Shifting non-negative integers to the right is equivalent to dividing by powers of 2

x >> y is equivalent to $x/2^y$

When a positive value is shifted to the left or to the right, the vacant bits are filled with zeros.

When a negative number is shifted to the right, the vacant bits can be filled with ones or zeros, depending on the implementation.