## C OVERVIEW

Goals

- speed
- portability
- allow access to features of the architecture
- speed

C

- fast executables
- allows high-level structure without losing access to machine features
- many popular languages such as C++ , Java, Perl use C syntax/C as a basis
- generally a compiled language
- reasonably portable
- very available and popular

## BASIC C PROGRAM STRUCTURE

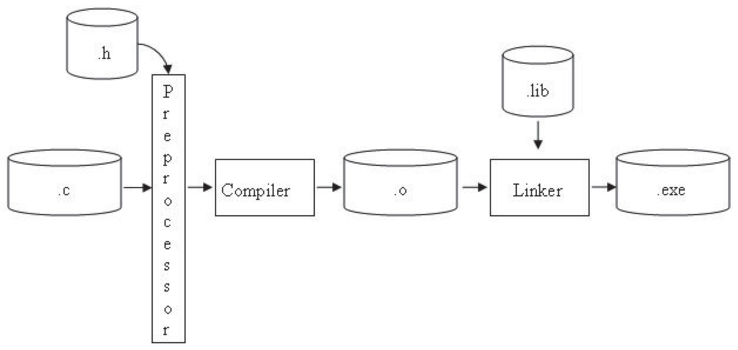
- The function main( ) is found in every C program and is where every C program begins execution.
- C uses braces { } to delimit the start/end of a function and to group sets of statements together into what C calls a block of code.
- Semicolons are used to terminate each C statement.
- Groups of instructions can be gathered together and named for ease of use and ease of programming. These “modules” are called functions in C.

Ex:

```
/*     FILE: first.c     */

#include <stdio.h>

int main( )
{
  printf("Hello world! \n");

  return 0;
}


/*    OUTPUT: first.c

          Hello world!

*/
```

## COMPILING AND LINKING

- Producing an executable file from C source code involves a two step process, compiling and linking.
- The compiler translates the C code into machine code, the linker combines the new machine code with code for existing routines from available libraries and adds some startup code.
- The end result is a file full of machine instructions that are executable on the target machine.



Ex:

gcc first.c
- compile and link to a.exe

gcc -c first.c
- compile only, stop at object module

gcc -lm first.c
- link in math libraries

## FUNDAMENTAL DATA TYPES

- there are three basic types of data, integer, floating-point, and character
- character data type is really a small integer
- signed and unsigned integer types available

| Type | Size | Min | Max |
|---|---|---|---|
| char | 1 byte | 0 / -128 | 255 / 127 |
| short | 2 bytes | -32768 | 32767 |
| int | 2,4 bytes | -2147483648 | 2147483647 |
| long | 4 bytes | -2147483648 | 2147483647 |
| long long | 8 bytes | $2^{63}$ ~ -9x$10^{18}$ | $2^{63}$-1 ~ 9x$10^{18}$ |
| float | 4 bytes ~ 7 digits | $\pm 1.0 \times 10^{-37}$ | $\pm 3.4 \times 10^{+38}$ |
| double | 8 bytes ~ 14 digits | $\pm 1.0 \times 10^{-307}$ | $\pm 1.8 \times 10^{+308}$ |
| long double | 12 bytes ~ 20 digits | $\pm 1.0 \times 10^{-4931}$ | $\pm 1.0 \times 10^{+4932}$ |

Ex:

```
/*     FILE: unsigned.c     */

/*
   Illustration of the unsigned keyword.  Allows
   recovering the use of the lead bit for magnitude.
*/
#include <stdio.h>

int main( )
{
  unsigned int x;

  x = 3333222111;

  printf("Unsigned x = %u\n", x);

  printf("Signed x = %d\n", x);

  return 0;
}


/*    OUTPUT: unsigned.c

          Unsigned x = 3333222111
          Signed x = -961745185

*/
```

## COMMENTS

- Block-style comments /* … */   Everything between the opening /* and the first */ is a comment.
- Comment-to-end-of-line:  //   Everything from the  //  to the end of the line is a comment.
- Nesting of block-style comments doesn't work.
- Note: Some older compilers may not recognize the  // comment indicator.

Ex:

```
/*     FILE: example.c     */

#include <stdio.h>

/* C-style comments can span several lines
   ...where they are terminated by:
*/

int main( )
{
  printf("Here's a program\n");

  return 0;
}


/*    OUTPUT: example.c

          Here's a program

*/
```

## IDENTIFIERS

- C identifiers must follow these rules:
    - C is case sensitive
    - may consist of letters, digits, and the underscore
    - first character must be a letter, (could be underscore but this is discouraged)
    - no length limit but only the first 31-63 may be significant

## KEYWORDS

| | | | |
|---|---|---|---|
| auto | extern | short | while |
| break | float | **signed** | *_Alignas* |
| case | for | sizeof | *_Alignof* |
| char | goto | static | *_Bool* |
| **const** | if | struct | *_Complex* |
| continue | *inline* | switch | *_Generic* |
| default | int | typedef | *_Imaginary* |
| do | long | union | *_Noreturn* |
| double | register | unsigned | *_Static_assert* |
| else | *restrict* | void | *#_Thread_local* |
| **enum** | return | **volatile** | |

## BASIC INPUT AND OUTPUT

- The basic I/O functions are printf( ) and scanf( ).
- printf( ) and scanf( ) are very generic. They always are processing text on one end. They get all their information about the data type they are to print or scan from the conversion specifiers.
- printf( ) always is producing text output from any number of internal data formats, i.e. int, float, char, double. The job of the conversion specifiers is to tell printf( ) how big a piece of data it's getting and how to interpret the internal representation.
- scanf( ) always receives a text representation of some data and must produce an internal representation. It is the conversion specifiers job to tell scanf( ) how to interpret the text and what internal representation to produce.
- printf( ) tips and warnings:
  * Make sure your conversion specifiers match your data values in number, type and order.
  * Use %f for both float and double.
  * Everything you put in the format string prints exactly as it appears, except conversion specifiers and escape sequences.
- scanf( ) tips and warnings:
  * Make sure your conversion specifiers match your data values in number, type and order.
  * As a general rule, scan only one value with each scanf( ) call, unless you really know what you are doing.
  * Use %f for float, %lf for double and %Lf for long double.
  * Don't forget the &, except with strings. {Someday you'll know why that is, and it will make sense.}
  * For %c every character you type is a candidate, even <return>. Placing a space in front of the %c in the format string will cause scanf( ) to skip whitespace characters.
  * scanf( ) is NOT without it's problems. However, it provides an easy way to get text input into a program and has some very handy conversion capabilities.

## CONVERSION SPECIFIERS

### printf( )

| | |
|---|---|
| %d | signed decimal int |
| %hd | signed short decimal integer |
| %ld | signed long decimal integer |
| %lld | signed long long decimal integer |
| %u | unsigned decimal int |
| %lu | unsigned long decimal int |
| %llu | unsigned long long decimal int |
| %o | unsigned octal int |
| %x | unsigned hexadecimal int with lowercase |
| %X | unsigned hexadecimal int with uppercase |
| %f | float or double [-]dddd.dddd. |
| %e | float or double of the form [-]d.dddd e[+/-]ddd |
| %g | either e or f form, whichever is shorter |
| %E | same as e; with E for exponent |
| %G | same as g; with E for exponent if e format used |
| %Lf, | |
| %Le, | |
| %Lg | long double |
| %c | single character |
| %s | string |
| %p | pointer |

### scanf( )

| | |
|---|---|
| %d | signed decimal int |
| %hd | signed short decimal integer |
| %ld | signed long decimal integer |
| %u | unsigned decimal int |
| %lu | unsigned long decimal int |
| %o | unsigned octal int |
| %x | unsigned hexadecimal int |
| %f | float |
| %lf | double **<u>NOTE:</u>** double & float are distinct for scanf !!!! |
| %LF | long double |
| %c | single character |
| %s | string |

## ESCAPE SEQUENCES

- Certain characters are difficult to place in C code so an escape code or escape sequence is used to encode these characters.
- These escape sequences all begin with a backslash ‘\’ and cause the encoded character to be placed into the program.

| Escape | value |
|---|---|
| \n | newline |
| \t | tab |
| \f | formfeed |
| \a | alarm |
| \b | backspace |
| \r | carriage return |
| \v | vertical tab |

Ex:

```
/*     FILE: print.c     */

/*
   Illustration of printf( ) and conversion specifiers.
*/
#include <stdio.h>

int main( )
{
  int x = 12;
  float y = 3.75;

  printf("%d", x);

  printf("\nx = %d\n", x);

  printf("y = %f\n", y);

  return 0;
}


/*    OUTPUT: print.c

          12
          x = 12
          y = 3.750000

*/
```

## OPERATORS

Arithmetic operators:

| | |
|---|---|
| * / % | multiplication/division/modulus |
| + − | addition/subtraction |
| + − | positive/negative sign (unary) |
| ++ − − | increment/decrement (unary) |

Logical operators:

| | |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT (unary) |

Relational operators:

| | |
|---|---|
| < <= > >= | less than, less than or equal to, greater than, greater than or equal to |
| = = != | equal to and not equal to |

Bit operators:

| | |
|---|---|
| << >> | left and right bit shift |
| & | bitwise AND |
| \| | bitwise OR |
| ^ | bitwise exclusive or XOR |
| ~ | bitwise NOT (unary) |

Assignment operators:

= += −= *= /= %= &= ^= |= <<= >>=

Address/Pointer operators:

| | |
|---|---|
| & | address of (unary) |
| * | dereference (unary) |

Structure operators:

| | |
|---|---|
| . | structure member acccess |
| −> | member access thru a structure pointer |

Other operators:

| | |
|---|---|
| ( ) | function call |
| [ ] | array access |
| (*type*) | type cast (unary) |
| sizeof | data object size in bytes (unary) |
| ?: | conditional operator |
| , | comma operator |

## OPERATOR PRECEDENCE

- The C compiler determines the order of operations in a C statement by operator precedence.
- Operator Precedence is the ranking of operators in C. The higher the rank the sooner the operator is evaluated.
- Parentheses can be used to override operator precedence.
- There are many kinds of operators but all operators are ranked via operator precedence.
- In the case of operators with the same rank, associativity is used and the operators are evaluated left-to-right or right-to-left.
- Operator precedence and associativity are detailed in the Operator Precedence Chart in the appendix, on the following page, and on pg. 53 in the K&R book

## OPERATOR PRECEDENCE CHART

| **Operators** | **Associativity** |
|---|---|
| ( )   [ ]   –>   . | left to right *{( ) function call}* |
| !   ~   ++   – –   +   –   *   &   (*type*)   sizeof | right to left *{All Unary}* |
| *   /   % | left to right |
| +   – | left to right |
| <<   >> | left to right |
| <   <=   >   >= | left to right |
| = =   != | left to right |
| & | left to right |
| ^ | left to right |
| \| | left to right |
| && | left to right |
| \|\| | left to right |
| ?: | right to left |
| =   +=   –=   *=   /=   %=   &=   ^=   \|=   <<=   >>= | right to left |
| , | left to right |

## ARITHMETIC OPERATORS

- Arithmetic operators are the symbols used by C to indicate when an arithmetic operation is desired.
- Arithmetic operators follow the same precedence rules we learned as kids. Multiplication & division before addition and subtraction. In case of a tie evaluate left-to-right. { Look at a precedence chart and see if this is true. }
- The modulus operator, %, is an additional arithmetic operator. It produces the remainder of integer division and ranks at the same level as division in the precedence chart.
- The increment, ++, and decrement, --, operators are basically a shorthand notation for increasing or decreasing the value of a variable by one.

Ex:
```
/*     FILE: arith_1.c     */

/* Arithmetic operators */

#include <stdio.h>

int main( )
{
  int first, second, sum;

  first = 11;
  second = 12;

  sum = first + second;
  printf("sum = %d\n", sum);

  sum = first - second;
  printf("sum = %d\n", sum);

  sum = first * second;
  printf("sum = %d\n", sum);

  sum = first / second;
  printf("sum = %d\n", sum);

  return 0;
}

/*    OUTPUT: arith_1.c

          sum = 23
          sum = -1
          sum = 132
          sum = 0

*/
```

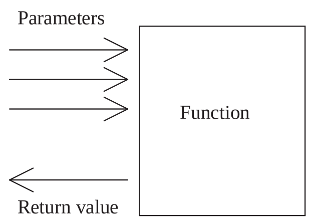
## INCREMENT ++/DECREMENT -- OPERATORS

- C has two specialized operators for incrementing and decrementing the value of a variable.
  - ++ - will increase a variables value by "one"
  - -- - will decrease a variables value by "one"
- Both operators can be written in both prefix and postfix notation. Each has implications as to when the actual increment or decrement takes place. Fortunately the implications are reasonable. Prefix notation causes the increment/decrement to occur "before" the value of the variable is supplied to an expression. Postfix notation causes the increment/decrement to occur "after" the value of the variable is supplied to an expression. In all cases the variables value is increased/decreased by "one"

Ex:
```
/*     FILE: incDec.c     */

/* Example of increment & decrement, postfix and prefix. */

#include <stdio.h>

int main( )
{
  int i =7;

  printf("i = %d\n", i++);
  printf("After postfix ++, i = %d\n", i);

  printf("i = %d\n", ++i);
  printf("After prefix ++, i = %d\n", i);

  printf("i = %d\n", i--);
  printf("After postfix --, i = %d\n", i);

  printf("i = %d\n", --i);
  printf("After prefix --, i = %d\n", i);

  return 0;
}

/*    OUTPUT: incDec.c

          i = 7
          After postfix ++, i = 8

          i = 9
          After prefix ++, i = 9

          i = 9
          After postfix --, i = 8

          i = 7
          After prefix --, i = 7

*/
```

## FUNCTIONS

- C allows a block of code to be separated from the rest of the program and named.
- These named blocks of code, or modules, are called functions.
- Functions can be passed information thru a parameter list and can pass back a result thru a return value.
- Any number of parameters can be passed to a function but at most one return value can be produced.
- All the C data types are candidates for parameter types and return types.
- Ideally a function can be treated as a black-box. If you know what to pass it and what it will return you don't need to know how it works.
- C has a special keyword, *void*, that is used to explicitly state that there are no parameters or no return value.



## LOGICAL, TRUE/FALSE VALUES

- The C definition of true and false is that 0 is false and any non-zero value is true.
- This definition allows some unusual expressions to be used as test conditions.

## RELATIONAL OPERATORS

- Relational operators are used quite often to produce the logical value for a conditional statement.

| operator | function |
|---|---|
| = = | equality |
| < | less than |
| > | greater than |
| <= | less than or equal |
| >= | greater than or equal |
| != | not equal |

## LOGICAL OPERATORS

- Logical operators work on logical values, i.e. true and false.

| operator | function |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

## LOOPING

- C has three looping constructs, for, while, and do while.
- The while loop is a fundamental pre-test condition loop that repeats as long as the test condition is true.
- The for loop is just a specialized while loop that allows initialization and post-iteration processing to be specified adjacent to the test condition. It is the most commonly used loop in C.
- The do while is just a while loop with the test condition moved to the bottom of the loop. It is a post-test condition loop so the test is executed after each iteration of the loop. (The positioning of the test makes the timing clear.) The main feature of the do while is that it will always execute the body of the loop at least once.

Ex:
```
/*     FILE: for_1.c     */

/* for loop example. */

#include <stdio.h>

int main( )
{
  int i;

  for(i = 0; i < 10; i++)
  {
     printf("i = %d\n", i);
  }

  return 0;
}


/*    OUTPUT: for_1.c

          i = 0
          i = 1
          i = 2
          i = 3
          i = 4
          i = 5
          i = 6
          i = 7
          i = 8
          i = 9

*/
```

## CONDITIONAL STATEMENTS

- C has two conditional statements and a conditional operator.
- The basic conditional statement in C is the if. An if is associated with a true/false condition. Code is conditionally executed depending on whether the associated test evaluates to true or false.
- The switch statement allows a labeled set of alternatives or cases to be selected from based on an integer value.
- The conditional operator ?: allows a conditional expression to be embedded in a larger statement.

Ex:
```
/*     FILE: if.c     */

/* if examples. */

#include <stdio.h>

int main( )
{
  int i;

  i = 5;
  if(i > 0)
    printf("%d > 0\n", i);

  i = -2;
  if(i > 0)
    printf("%d > 0\n", i);
  else
    printf("%d <= 0\n", i);

  i = -2;
  if(i > 0)
    printf("%d > 0\n", i);
  else
    if(i == 0)              /* Test for equality is == */
      printf("%d == 0\n", i);
    else
      printf("%d < 0\n", i);

  return 0;
}


/*    OUTPUT: if.c

          5 > 0
          -2 <= 0
          -2 < 0

*/
```