Basis of Software

Leon Tabak

13 May 2019

Contents

License

The title of the MIT course was *6.087 Practical Programming in C*. Daniel Weller and Sharat Chikkerur taught the course during MIT's Independent Activities Period in 2010.

Lesson 1

0.1 What is C?

- Dennis Ritchie—AT&T Bell Laboratories—1972
  - 16-bit DEC PDP-11 computer (right)
- Widely used today
  - extends to newer system architectures
  - efficiency/performance
  - low-level access

0.2 People and places*

- Brian Kernighan
- Dennis Ritchie
- Ken Thompson
- The Idea Factory: Bell Labs and the Great Age of American Innovation

0.3 Features of C

- C features:
  - Few keywords
  - Structures, unions—compound data types
  - Pointers—memory, arrays
  - External standard library—I/O, other facilities
  - Compiles to native code
  - Macro preprocessor

0.4 Versions of C

- Evolved over the years:
  - 1972—C invented
  - 1978—The C Programming Language published; first specification of language
  - 1989—C89 standard (known as ANSI C or Standard C)

  - 1990—ANSI C adopted by ISO, known as C90
  - 1999—C99 standard
    * mostly backward-compatible
    * not completely implemented in many compilers
  - 2007—work on new C standard C1X announced
- In this course: ANSI/ISO C (C89/C90)

0.5 What is C used for?

- Systems programming:
  - OSes, like Linux
  - microcontrollers: automobiles and airplanes
  - embedded processors: phones, portable electronics, etc.
  - DSP processors: digital audio and TV systems

0.6 C vs. related languages

- More recent derivatives: C++, Objective C, C#
- Influenced: Java, Perl, Python (quite different)
- C lacks:
  - exceptions
  - range-checking
  - garbage collection
  - object-oriented programming
  - polymorphism
  - ...
- Low-level language ⇒ faster code (usually)

0.7 Warning: low-level language

- Inherently unsafe:
  - No range checking
  - Limited type safety at compile time
  - No type checking at runtime

- Handle with care.
  - Always run in a debugger like gdb (more later...)
  - Never run as root
  - Never test code on the Athena servers

0.8 Editing C code

- .c extension
- Editable directly
- More later...

0.9 Compiling a program

- gcc (included with most Linux distributions): compiler
  - .o extension
    - omitted for common programs like gcc

0.10 More about gcc

- Run gcc:

  athena% gcc -Wall infilename.c -o outfilename.o

- -Wall enables most compiler warnings
- More complicated forms exist
  - multiple source files
  - auxiliary directories
  - optimization, linking
- Embed debugging info and disable optimization:

  athena% gcc -g -O0 -Wall infilename.c -o outfilename.o

0.11 Debugging

0.12 Using gdb

- Some useful commands:
  - break linenumber — create breakpoint at specified line
  - break file:linenumber — create breakpoint at line in file
  - run — run program
  - c — continue execution
  - next — execute next line
  - step — execute next line or step into function
  - quit — quit gdb
  - print expression — print current value of the specified expression
  - help command in-program help

0.13 Memory debugging

- Figure: valgrind: diagnose memory-related problems

0.14 The IDE—all-in-one solution

- Popular IDEs: Eclipse (CDT), Microsoft Visual C++ (Express Edition), KDevelop, Xcode, . . .
- Integrated editor with compiler, debugger
- Very convenient for larger programs

0.15 Using Eclipse

- Need Eclipse CDT for C programs (see http://www.eclipse.org/cdt/)
- Use New > C Project
  - choose "Hello World ANSI C Project" for simple project
  - "Linux GCC toolchain" sets up gcc and gdb (must be installed separately)
- Recommended for final project

0.16 Hello, 6.087 students

- In style of "Hello, world!"
- .c file structure
- Syntax: comments, macros, basic declarations
- The main() function and function structure
- Expressions, order-of-operations
- Basic console I/O (puts(), etc.)

0.17 Structure of a .c file

```
/* Begin with comments about file contents */

Insert #include statements and preprocessor
definitions

Function prototypes and variable declarations

Define main() function
{
Function body
}

Define other function
{
Function body
}
```

0.18 Comments

- Comments: / this is a simple comment /
- Can span multiple lines

```
/
   This comment
   spans
   multiple lines
*/
```

- Completely ignored by compiler
- Can appear almost anywhere

```
/
  h e l l o . c         our first C program

  Created by Daniel Weller , 01/11/2010
   /
```

0.19 The #include macro

- Header files: constants, functions, other declarations
- **#include** <stdio.h> — read the contents of the header file stdio.h
- stdio.h: standard I/O functions for console, files

```
/
  hello.c         our first C program

  Created by Daniel Weller , 01/11/2010
   /

#include <stdio.h> /     basic I/O facilities     /
```

0.20 More about header files

- stdio.h — part of the C Standard Library
- other important header files: ctype.h, math.h, stdlib.h, string.h, time.h
- For the ugly details: visit http://www.unix.org/single_unix_specification/ (registration required)
- Included files must be on include path
- -Idirectory with gcc: specify additional include directories
- standard include directories assumed by default
- **#include** "stdio.h" — searches ./ for stdio.h first

0.21 Declaring variables

- Must declare variables before use
- Variable declaration:

```
int n;

float phi;
```

- **int** — integer data type
- **float** — floating-point data type
- Many other types (more next lecture...)

0.22 Initializing variables

- Uninitialized, variable assumes a default value
- Variables initialized via assignment operator:

```
n = 3;
```

- Can also initialize at declaration:

```
float phi = 1.6180339887;
```

- Can declare/initialize multiple variables at once:

```
int a, b, c = 0, d = 4;
```

0.23 Arithmetic expressions

Suppose x and y are variables...

- binary arithmetic

```
x+y, x-y, x*y, x/y, x%y
```

- A simple statement:

```
y = x+3    x /( y   4 );
```

- Numeric literals like 3 or 4 valid in expressions
- Semicolon ends statement (not newline)
- arithmetic and assignment

```
x += y, x -= y, x *= y, x /= y, x %= y
```

0.24 Order of operations

- Order of operations:

Operator	Evaluation direction
`+,-` (sign)	right-to-left
`*,/,%`	left-to-right
`+,-`	left-to-right
`=,+=,-=,*=,/=,%=`	right-to-left

- Use parentheses to override order of evaluation

0.25 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;

0.26 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;
2. Evaluate multiplies and divides, from left-to-right
   float z = x+3 x /2.0; → **float** z = x+6.0/2.0; **float** z = x+3.0;

0.27 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;
2. Evaluate multiplies and divides, from left-to-right
   float z = x+3 x /2.0; → **float** z = x+6.0/2.0; **float** z = x+3.0;
3. Evaluate addition
   float z = x+3.0; → **float** z = 5.0;

0.28 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;
2. Evaluate multiplies and divides, from left-to-right
   float z = x+3 x /2.0; → **float** z = x+6.0/2.0; → **float** z = x+3.0;
3. Evaluate addition
   float z = x+3.0; → **float** z = 5.0;
4. Perform initialization with assignment
   Now, z = 5.0.

0.29 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;
2. Evaluate multiplies and divides, from left-to-right
   float z = x+3 x /2.0; → **float** z = x+6.0/2.0; → **float** z = x+3.0;
3. Evaluate addition
   float z = x+3.0; → **float** z = 5.0;
4. Perform initialization with assignment
   Now, z = 5.0.

How do I insert parentheses to get z = 4.0?

0.30 Order of operations

Assume x = 2.0 and y = 6.0. Evaluate the statement **float** z = x+3 x /(y4);

1. Evaluate expression in parentheses
   float z = x+3 x /(y4); → **float** z = x+3 x /2.0;
2. Evaluate multiplies and divides, from left-to-right
   float z = x+3 x /2.0; → **float** z = x+6.0/2.0; → **float** z = x+3.0;

3. Evaluate addition
   float z = x+3.0; → **float** z = 5.0;
4. Perform initialization with assignment
   Now, z = 5.0.

How do I insert parentheses to get z = 4.0?

float z = (x+3 x)/(y4);

0.31 Function prototypes

- Functions also must be declared before use
- Declaration called function prototype
- Function prototypes:
  int factorial (**int**); or **int** factorial (**int** n);
- Prototypes for many common functions in header files for C Standard Library

0.32 Function prototypes

- General form:
  return_type function_name(arg1,arg2 ,...);
- Arguments: local variables, values passed from caller
- Return value: single value returned to caller when function exits
- **void** — signifies no return value/arguments
  int rand(**void**);

0.33 The main() function

- main(): entry point for C program
- Simplest version: no inputs, outputs 0 when successful, and nonzero to signal some error
  int main(**void**);
- Two-argument form of main(): access command-line arguments
  int main(**int** argc, **char** argv);
- More on the **char** **argv notation later this week...

0.34 Function definitions

```
Function declaration
{
  declare variables;
  program statements;
}
```

- Must match prototype (if there is one)
  - variable names dont have to match
  - no semicolon at end
- Curly braces define a block — region of code
  - Variables declared in a block exist only in that block
- Variable declarations before any other statements

0.35 Our main() function

```
/    The main() function    /
int main( void ) /    entry point    /
{
    /    write message to console    /
    puts ( "hello, 6.087 students" )
    return 0 ; /    exit ( 0 => success )    /
}
```

- puts(): output text to console window (stdout) and end the line
- String literal: written surrounded by double quotes
- **return** 0; exits the function, returning value 0 to caller

0.36 Alternative main() function

- Alternatively, store the string in a variable first:

```
int main( void ) /    entry point    /
{
    const char msg [ ] = "hello, 6.087 students" ;

    /    write message to console    /

    puts( msg ) ;
```

- **const** keyword: qualifies variable as constant
- **char**: data type representing a single character; written in quotes: a , 3 , n
- **const char** msg[]: a constant array of characters

0.37 More about strings

- Strings stored as character array
- Null-terminated (last character in array is `\0` null)
- Not written explicitly in string literals
- Special characters specified using `\` (escape character):
- `\\` — backslash, `\` — apostrophe, `\` — quotation mark
- `\b`, `\t`, `\r`, `\n` — backspace, tab, carriage return, linefeed
- `\ooo`, `\xhh` — octal and hexadecimal ASCII character codes, e.g. \x41 — A, \060 — 0

0.38 Console I/O

- stdout, stdin: console output and input streams
- puts(string): print string to stdout
- putchar(**char**): print character to stdout
- **char** = getchar(): return character from stdin
- string = gets(string): read line from stdin into string
- Many others — later this week

0.39 Preprocessor macros

- Preprocessor macros begin with # character
  #include <stdio.h>
- #define msg "hello, 6.087 students"
  defines msg as hello, 6.087 students throughout source file
- many constants specified this way

0.40 Defining expression macros

- **#define** can take arguments and be treated like a function
- **#define** add3(x,y,z) ((x)+(y)+(z))
- parentheses ensure order of operations
- compiler performs inline replacement; not suitable for recursion

0.41 Conditional preprocessor macros

- **#if** , #**ifdef** , #**ifndef** , #**else** , # **elif** , #**endif** conditional preprocessor macros, can control which lines are compiled
- evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
- the gcc option −Dname=value sets a preprocessor define that can be used
- Used in header files to ensure declarations happen only once

0.42 Conditional preprocessor macros

- **#pragma**

  preprocessor directive
- **#error**, #**warning**

  trigger a custom compiler error/warning
- **#undef** msg

  remove the definition of msg at compile time

0.43 Compiling our code

After we save our code, we run gcc:

```
gcc -g -O0 -Wall hello.c -o
hello.o
```

Assuming that we have made no errors, our compiling is complete.

0.44 Running our code

Or, in gdb,

```
athena% gdb hello.o
.
.
.
Reading symbols from hello.o...done.
(gdb) run
Starting program: hello.o
hello, 6.087 students
Program exited normally.
(gdb) quit
athena%
```

0.45 Summary

Topics covered:

- How to edit, compile, and debug C programs
- C programming fundamentals:
  - comments
  - preprocessor macros, including **#include**
  - the main() function
  - declaring and initializing variables, scope
  - using puts() — calling a function and passing an argument
  - returning from a function

Lesson 2

0.46 Review: C programming language

- C is a fast, small,general-purpose,platform independent programming language.
- C is used for systems programming (e.g., compilers and interpreters, operating systems, database systems, microcontrollers etc.)
- C is static (compiled), typed, structured and imperative.
- "C is quirky, flawed, and an enormous success."—Ritchie

0.47 Review: Basics

- Variable declarations: **int** i ; **float** f ;
- Intialization: **char** c= A ; **int** x=y=10;
- Operators: +, , ,/,%
- Expressions: **int** x,y,z; x= y2 + z3 ;
- Function: **int** factorial (**int** n); / function takes **int** , returns **int** /

0.48 Definitions

- Datatypes:
  - The datatype of an object in memory determines the set of values it can have and what operations that can be performed on it.
  - C is a weakly typed language. It allows implicit conversions as well as forced (potentially dangerous) casting.
- Operators:
  - Operators specify how an object can be manipulated (e.g.,, numeric vs. string operations).
  - operators can be unary(e.g., −,++), binary (e.g., +,−,∗,/), ternary (?:)

0.49 Definitions (continued)

- Expressions:
  - An expression in a programming language is a combination of values, variables, operators, and functions
- Variables:
  - A variable is as named link/reference to a value stored in the systems memory or an expression that can be evaluated.
- Consider: **int** x=0,y=0; y=x+2; .
  - x, y are variables
  - y = x + 2 is an expression
  - + is an operator.

0.50 Variable names

- Naming rules:
  - Variable names can contain letters,digits and _
  - Variable names should start with letters.
  - Keywords (e.g., **for**,**while** etc.) cannot be used as variable names
  - Variable names are case sensitive. **int** x; **int** X declares two different variables.
- Pop quiz (correct/incorrect):
  - **int** money$owed; (incorrect: cannot contain $)
  - **int** total_count (correct)
  - **int** score2 (correct)
  - **int** 2ndscore (incorrect: must start with a letter)
  - **int long** (incorrect: cannot use keyword)

0.51 Data types and sizes

- C has a small family of datatypes.
  - Numeric (**int**,**float**,**double**)
  - Character (**char**)
  - User defined (**struct**,**union**)

0.52 Numeric data types

Depending on the precision and range required, you can use one of the following datatypes.

	signed	**unsigned**
short	**short int** x; **short** y	**unsigned short int** x; **unsigned short int** y;
default	**int** x;	**unsigned int** x;
long	**long** x;	**unsigned long** x;
float	**float** x;	N/A
double	**double** x;	N/A
char	**char** x; **signed char** x;	**unsigned char** x;

- The unsigned version has roughly double the range of its signed counterparts.
- Signed and unsigned characters differ only when used in arithmetic expressions.
- Tidbit: Flickr changed from unsigned long ($2^{32}1$) to string two years ago.

0.53 Big endian vs. little endian

The individual sizes are machine/compiler dependent. However, the following is guaranteed:

sizeof(**char**) < **sizeof**(**short**) ≤ **sizeof**(**int**) ≤ **sizeof**(**long**)

and

sizeof(**char**) < **sizeof**(**short**) ≤ **sizeof**(**float**) ≤ **sizeof**(**double**)

"NUXI" problem: For numeric data types that span multiple bytes, the order of arrangement of the individual bytes is important. Depending on the device architecture, we have "big endian" and "little endian" formats.

0.54 Big endian vs. little endian (continued)

- Big endian: the **most** significant bits (MSBs) occupy the lower address. This representation is used in the powerpc processor. Networks generally use big-endian order, and thus it is called **network order**.
- Little endian : the **least** signficant bits (LSBs) occupy the lower address. This representation is used on all x86 compatible processors.

Example: Suppose that a 32 bit register holds the hexadecimal value $0A0B0C0D$ and that its contents are written to a block of memory that begins at address a;

address	**contents of cell (little endian)**	**contents of cell (big endian)**
a	0D	0A
a + 1	0C	0B
a + 2	0B	0C
a + 3	0A	0D

0.55 Constants

Constants are literal/fixed values assigned to variables or used directly in expressions.

Datatype	**example**	**meaning**
integer	**int** i =3;	integer
integer	**long** l=3;	long integer
integer	**unsigned long** ul= 3UL;	unsigned long
integer	**int** i =0xA;	hexadecimal
integer	**int** i =012;	octal number
floating point	**float** pi=3.14159	float
floating point	**float** pi=3.141F float	
floating point	**double** pi=3.14159265358979323 84L	double

0.56 Constants (continued)

Datatype	example	meaning
character	'A'	character
character	'\x41'	character specified in hexadecimal
character	'\0101'	character specified in octal
string	"hello␣world"	string literal
string	"hello""world"	same as "hello world"
enumeration	**enum** BOOL (NO, YES)	NO = 0, YES = 1
enumeration	**enum** COLOR (R=1,G,B,Y=10)	G = 2, B = 3

0.57 Declarations

The general format for a declaration is *type variable-name [=value]*

Examples:

- **char** x; / uninitialized /
- **char** x='A'; / initialized to 'A' /
- **char** x='A',y='B'; / multiple variables initialized /
- **char** x=y= Z ; / multiple initializations /

0.58 Pop quiz II

- **int** x=017;**int** y=12; / is x>y? /
- **short int** s=0xFFFF12; / correct ? /
- **char** c= 1 ;**unsigned char** uc=1; / correct ? /
- puts("hel"+"lo");puts("hel""lo");/ which is correct? /
- **enum** sz{S=0,L=3,XL}; /what is the value of XL? /
- **enum** sz{S=0,L=3,XL}; / what is the value of XL? /

0.59 Arithmetic operators

operator	meaning	examples
+	addition	x = 3 + 2; /* *constants* */
+	addition	y + z; /* *variables* */
+	addition	x + y + 2; /* *variables and constants* */
−	subtraction	3 − 2; /* *constants* */
−	subtraction	**int** x = y − z; /* *variables* */
−	subtraction	y − 2 − z; /* *variables and constants* */
*	multiplication	**int** x = 3 * 2; /* *constants* */
*	multiplication	**int** x = y * z; /* *variables* */
*	multiplication	x * y * 2; /* *variables and constants* */

0.60 Arithmetic operators (continued)

operator	meaning	examples
/	division	**float** = 3/2; /* *produces x = 1 (int)* */
/	division	**float** x = 3.0/2; /* *produces x = 1.5 (float)* */
/	division	**int** x = 3.0/2; /* *produces x = 1 (int conversion)* */
%	modulus (remainder)	**int** x = 3 % 2; /* *produces x = 1* */
%	modulus (remainder)	**int** y = 7; **int** x = y % 4; /* *produces x = 3* */
%	modulus (remainder)	**int** y = 7; **int** x = y % 10; /* *produces x = 7* */

0.61 Relational operators

Relational operators compare two operands to produce a 'boolean' result. In C any non-zero value (1 by convention) is considered to be 'true' and 0 is considered to be false.

operator	meaning	examples
>	greater than	3 > 2 /* *evaluates to 1* */
>	greater than	2.99 > 3 /* *evaluates to 0* */
>=	greater than or equal to	>= /* evaluates to 1 */@@
>=	greater than or equal to	2.99 >= 3 /* *evaluates to 0* */
<	less than	3 < 3 /* *evaluates to 0* */
<	less than	'A' < 'B' /* *evaluates to 1* */
<=	less than or equal to @ 3 <= 3 /* *evaluates to 1* */	
<=	less than or equal to @ 3.99 < 3 /* *evaluates to 0* */	

0.62 Relational operators

operator	meaning	examples
==	equal to	3 == 3 /*evaluates to 1 */
==	equal to	'A' == 'a' /* evaluates to 0 */
!=	not equal to	3 != 3 /* evaluates to 0 */
!=	not equal to	2.99 != 3 /* evaluates to 1 */

- Note that the == equality operator is different from the =, assignment operator.
- Note that the == operator on float variables is tricky because of finite precision.

0.63 Logical operators

operator	meaning	examples
&&	AND	((9/3) == 3) && (2 * 3 == 6) /*evaluates to 1 */
&&	AND	('A' == 'a') && (3 == 3) /*evaluates to 0 */
||	OR	2 == 3 || 'A' == 'A'/* evaluates to 1 */
||	OR	2.99 >= 3 || 0 /* evaluates to 0 */
!	NOT	!(3 == 3) /*evaluates to 0 */
!	NOT	!(2.99 >= 3) /*evaluates to 1 */

- Short circuit: The evaluation of an expression is discontinued if the value of a conditional expression can be determined early.
- Be careful of any side effects in the code.
- Examples:
  - (3==3) || ((c=getchar())== y)
    The second expression is not evaluated.
  - (0) && ((x=x+1)>0)
    The second expression is not evaluated.

0.64 Increment and decrement operators

Increment and decrement are common arithmetic operation.

C provides two short cuts for the same.

- Postfix
  - x++ is a short cut for x=x+1

- x is a short cut for x= x1
- y=x++ is a short cut for y=x;x=x+1.
  x is evaluated before it is incremented.
- y= x is a short cut for y=x;x= x1 .
  x is evaluated before it is decremented.

0.65 Increment and decrement operators

- Prefix:
  - ++x is a short cut for x=x+1
  - x is a short cut for x= x1
  - y=++x is a short cut for x=x+1;y=x;.
    x is evaluate after it is incremented.
  - y= x is a short cut for x= x1 ;y=x;.
    x is evaluate after it is decremented.

0.66 Bitwise operators

operator	**meaning**	**examples**
&	AND	0x77 & 0x3; / evaluates to 0x3 /
&	AND	0x77 & 0x0; / evaluates to 0 /
|	OR	0x700 | 0x33; / evaluates to 0x733 /
|	OR	0x070 | 0 / evaluates to 0x070 /
^	XOR	0x770 ^ 0x773; / evaluates to 0x3 /
^	XOR	0x33 ^ 0x33; / evaluates to 0 /
<<	left shift	0x01<<4; / evaluates to 0x10 /
<<	left shift	1<<2; / evaluates to 4 /
>>	right shift	0x010>>4; / evaluates to 0x01 /
>>	right shift	4>>1 / evaluates to 2 /

- Notes:
  - AND is true only if both operands are true.
  - OR is true if any operand is true.
  - XOR is true if only one of the operand is true.

0.67 Assignment operators

- Another common expression type found while programming in C is of the type var = var (op) expr

  - x=x+1
  - x= x10
  - x=x/2

- C provides compact assignment operators that can be used instead.
  - x+=1 / is the same as x=x+1 /
  - x =1 / is the same as x= x1 /
  - x =10 / is the same as x= x10 /
  - x/=2 / is the same as x=x/2 */
  - x%=2 / is the same as x=x%2 */

0.68 Conditional expression

A common pattern in C (and in most programming) languages is the following:

```
if ( condition ) {
    x = <expresion a> ;
} // if
else {
  x = <expression b> ;
} // else
```

C provides syntactic sugar to express the same using the ternary operator '?:'

```
sign = x > 0?1:    1 ;

/* is equivalent to ... */

if( x > 0 ) {
  sign = 1;
} // if
else {
  sign =   1 ;
} // else

isOdd = x %2==1?1:0;

/* is equivalent to ... */

if( x%2 == 1 ) {
    isOdd = 1;
} // if
else {
    isOdd = 0;
} // else
```

Notice how the ternary operator makes the code shorter and easier to understand (syntactic sugar).

0.69 Type conversions

When variables are promoted to higher precision, data is preserved. This is automatically done by the compiler for mixed data type expressions.

```
int i ;
float f ;
f = i +3.14159; /     i is promoted to float , f = (float) i +3.14159      /
```

Another conversion done automatically by the compiler is **char** → **int**. This allows comparisons as well as manipulations of character variables.

```
/     c and literal constants are converted to int     /
isUpper = ( c >= 'A' && c <= 'Z' ) ? 1 : 0 ;

if( !isUpper ) {
  /
    subtraction is possible
    because of integer conversion
     /
  c = c      'a' + 'A' ;
} // if
```

As a rule (with exceptions), the compiler promotes each term in an binary expression to the highest precision operand.

0.70 Precedence and order of evaluation

- ++, ,(cast),**sizeof** have the highest priority
- ∗,/,% have higher priority than +,
- ==,!= have higher priority than &&,||
- assignment operators have very low priority

Use () generously to avoid ambiguities or side effects associated with precendence of operators.

- y= x3 +2 /   same as y=(x3)+2   /
- x!=0 && y==0 / same as (x!=0) && (y==0) /
- d = c>= 0 && c<= 9 /   same as d=(c>= 0) && (c<= 9)   /

Lesson 3

Lesson 4

Lesson 5

Lesson 6

Lesson 7

Lesson 8

Lesson 9

Lesson 10