



COMP327

Mobile Computing

Session: 2011-2012

Lecture Set 2b - Operators and Control Flow

In these Tutorial Slides...

- To cover in more depth most of the aspects of C, including:
 - Types, Operands, and Expressions
 - Functions and Program Structure
 - Control Flow



Recap

- In the previous Tutorial Slides we introduced C
 - Considered its origins, strengths and weaknesses.
 - Briefly considered the compilation process
 - Looked at several code examples
 - Introduced and discussed many elements of the language
- In this and the following Tutorial Slides
 - We will look at language issues in much more depth
 - This includes code fragments, tips, and insights
- As Objective-C is a superset of C, it is important to master C first...

Variable Names

- Variables normally consist of numbers and letters
 - Underscore ('_') is also permitted.
 - However, don't use underscores to start variable names
 - By convention, they are used in this way by library routines
 - Variables are case-specific
- Several naming conventions are used
 - variables are lower case, or at least start lower case
 - functions start with an upper case character
 - symbolic constants are all upper case
- Other constraints exist due to legacy systems,
 - at least the first 31 characters of a name are significant
 - externs should be unique for the first six characters in a single case

Data Types

- Only a few basic data types:

Type	Description
char	a single byte, capable of holding one character in the local character set
int	an integer, typically reflecting the natural size of integers on the host machine
float	single-precision floating point
double	double-precision floating point

- Quantifiers can also be used
 - short int - often a smaller int (e.g. 16-bit vs 32-bit int)
 - long int - at least 32-bits
 - Often the token “int” is omitted as it is implied
- long double is an extended-precision floating point
- chars and ints can be signed or unsigned
 - signed char has the value range -128..127
 - unsigned char has the value range 0..255
- Data type sizes are machine dependent, but specified in the headers `<limits.h>` and `<float.h>`

Data Types

```
/*      @(#)limits.h      8.3 (Berkeley) 1/4/94 */
...
#define SCHAR_MAX      127          /* min value for a signed char */
#define SCHAR_MIN      (-128)     /* max value for a signed char */

#define UCHAR_MAX      255         /* max value for an unsigned char */
#define CHAR_MAX       127         /* max value for a char */
#define CHAR_MIN       (-128)     /* min value for a char */

#define USHRT_MAX      65535       /* max value for an unsigned short */
#define SHRT_MAX       32767       /* max value for a short */
#define SHRT_MIN       (-32768)    /* min value for a short */

#define UINT_MAX       0xffffffff   /* max value for an unsigned int */
#define INT_MAX        2147483647   /* max value for an int */
#define INT_MIN        (-2147483647-1) /* min value for an int */

#ifdef __LP64__
#define ULONG_MAX      0xffffffffffffffffUL /* max unsigned long */
#define LONG_MAX       0x7fffffffffffffffL /* max signed long */
#define LONG_MIN       (-0x7fffffffffffffffL-1) /* min signed long */
#else /* !__LP64__ */
...

```

Value Constants

- We've already seen that a number is an int
 - unless it is followed by a decimal point (double)
- Other constant types can be specified
 - 123456789L - the suffix l or L signifies a long
 - the suffix ul or UL signifies an unsigned long
 - doubles can also be given with exponents, eg. 1e-2
 - a suffix of f or F indicates a float
 - a suffix of l or L with a decimal indicates a long double
- Number Bases
 - a '0' (zero) prefix indicates an octal value (31 == 037)
 - an '0x' prefix indicates a hexadecimal value (31 == 0x1F)
 - Both can be unsigned (suffix of u) or long (suffix of l)

Why do we care?

The type of a const affects how expressions are evaluated.

We've seen that dividing an int with an int generates an int, which may not give the desired value ($5/9 = 0$).

This is because the choice of operator used depends on type - integer division is *much* faster than floating point division.

Also, a const in a function should ideally match the function type definition.

Finally, constants might be used to mask logical bits, in which case size (or matching up types) is important!

Character and string constants

- A character constant is an integer, but represented as a character in single quotes
 - char a = '2' has the value 50 (in the ASCII character set)
 - Certain special characters are given as escape sequences
 - Characters can also be referred to using octal or hex values from their character sets
- String constants are arrays of chars, represented as characters in double quotes
 - the end of string character is the null '\0' sequence
 - thus the array size is the number of characters+1
- Note the use of quotes
 - 'x' (single quote) is **not** the same as "x" (double quote)
 - 'x' is an int; "x" is a two element char array

Character	Sequence
Alert / Bell	\a
Backspace	\b
Form feed	\f
Newline	\n
Carriage Return	\r
Horizontal tab	\t
Vertical tab	\v
Backslash	\\
Question Mark	\?
Single Quote	\'
Double Quote	\"
Octal Number	\ooo e.g. \013
Hexadecimal Number	\xhh e.g. \x3a

ASCII Character Set

```
% man ascii
```

```
ASCII(7)
```

```
BSD Miscellaneous Information Manual
```

```
ASCII(7)
```

NAME

```
ascii -- octal, hexadecimal and decimal ASCII character sets
```

DESCRIPTION

```
The hexadecimal set:
```

00 nul	01 soh	02 stx	03 etx	04 eot	05 enq	06 ack	07 bel
08 bs	09 ht	0a nl	0b vt	0c np	0d cr	0e so	0f si
10 dle	11 dc1	12 dc2	13 dc3	14 dc4	15 nak	16 syn	17 etb
18 can	19 em	1a sub	1b esc	1c fs	1d gs	1e rs	1f us
20 sp	21 !	22 "	23 #	24 \$	25 %	26 &	27 '
28 (29)	2a *	2b +	2c ,	2d -	2e .	2f /
30 0	31 1	32 2	33 3	34 4	35 5	36 6	37 7
38 8	39 9	3a :	3b ;	3c <	3d =	3e >	3f ?
40 @	41 A	42 B	43 C	44 D	45 E	46 F	47 G
48 H	49 I	4a J	4b K	4c L	4d M	4e N	4f O
50 P	51 Q	52 R	53 S	54 T	55 U	56 V	57 W
58 X	59 Y	5a Z	5b [5c \	5d]	5e ^	5f _
60 `	61 a	62 b	63 c	64 d	65 e	66 f	67 g
68 h	69 i	6a j	6b k	6c l	6d m	6e n	6f o
70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7a z	7b {	7c	7d }	7e ~	7f del

```
...
```

Enumerated Types

- Enums are another way of defining constants that form some list or progression:

```
enum boolean { NO, YES };      /* NO == 0, and YES == 1 */

enum days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };
          /* SUNDAY == 0; SATURDAY == 6; THURSDAY - MONDAY == 3; */

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
            /* FEB = 2, MAR = 3, etc. */
```

- The first name has the value 0 by default
 - Unspecified values continue the progression of the last specified value.
- Names must be unique, but values can be shared
 - e.g. many enumerated types start with zero (0)
- However, in most cases, the code refers to the constant, not the value
 - Provide a better way to create symbols than using #defines

Declarations

- All variables should be declared before use
 - A declaration specifies the type, and a list of one or more variables
 - Initialisation can also be done as part of the declaration
 - External and static variables are initialised to zero by default, all others are undefined
 - Variables can be declared within any braced body (i.e. within '{' and '}'), not just within a function
 - Can be used to hide variables within a local scope
 - However, can lead to confusion
- If in doubt, **always** initialise your variable.
- The const qualifier
 - can be applied to the declaration of any variable to specify the value will not be changed.
 - If violated, then the result is compiler specific

```
int lower, upper, step;  
int i=0;  
char c, line[1000];  
char esc = '\\';  
int limit = MAXLINE+1;  
float eps = 1.0e-5;
```

```
if (n > 0) {  
    int i; /* declare new i */  
    for i=0; i<n; i++)  
        ....  
}
```

```
const double e = 2.71828182845905;  
const char msg[] = "warning: ";
```

Functions

Functions

```
return-type function-name(argument declarations)
{
    declarations and statements
}
```

- Functions are similar to methods in Java, except that they are not class specific
 - They perform some job, taking arguments, and returning values
 - Variables/Data structures can also be side effected, depending on scope
- Not all elements need be defined
 - This dummy function does nothing and returns nothing
 - such functions are often useful as place-holders during development
 - If no return value is stated, an int return value is assumed
- Functions can be declared in any order in the source file
 - Function prototypes should be declared before a function is used
 - However, if the function is defined earlier in the same source, no prototype is needed

```
/* a function that
 * does nothing
 */
dummy() {}
```

Function Prototypes

Function Prototypes

```
return-type function-name(argument declarations);
```

- Function declarations are necessary for determining how much memory is needed when making a function call
 - Used by the compiler to calculate the number of memory bits needed for arguments
 - Also, the number of bits needed for the return value
- Functions can be implicitly or explicitly declared
 - Explicit declarations help to avoid mismatches between function definition and use
 - and thus avoid bugs
 - If a function has not been declared, then an implicit declaration will occur when it is first used
 - Based on the type of its arguments and whether or not a return value (and its type) is used.
- A function should be declared before use
 - This includes its arguments and return type
 - Not necessary if the function appears in the source code before use

```
/* Prototypes */  
int getop(char []);  
void push (double);  
double pop (void);
```

External Variables

- Variables may be global in scope (i.e. *external*)
 - Not declared in any function, hence can be used by all functions
 - Long lived, as they are never deleted
 - Need to be defined only once, but declared before use
 - Not a problem when one source file is used, as declaration is implied by definition
 - However, what happens when multiple source files are used?
 - **Declaration:**
 - Announces the properties of the variable, e.g. type
 - Extern used to indicate that the variable has been defined
 - **Definition:**
 - Declares the variable and sets aside storage.
- Externs are used with multiple source files
 - Often appear in Header Files, which are *#included*

```
/* Definition in file1.c */  
int sp;  
double val[MAXVAL];
```

```
/* Declaration in file2.c */  
extern int sp;  
extern double val[];
```

Static Variables

- Static declarations
 - Similar to private variables in methods
 - The static declaration is used to limit the scope of a variable to the source file being compiled
 - Good way of creating global variables that are shared by some functions (defined in the same source file), but hidden from others
 - Initial value can be defined
 - Can also be used to limit the scope of functions to a single source file
- Internal Static Variables in functions
 - Variables are normally deallocated (lost) once a function finishes executing
 - Static Variables are variables whose lifetime extends across the entire run of the program
 - Useful when a value should be shared between each function call

```
/* Buffer for ungetch */
static char buf[BUFSIZE];

/* Next free position in buf */
static int bufp = 0;
```

```
/* Function that returns
 * the number of times it
 * is called, using an
 * internal static variable
 */

int genval(double d) {
    static int count=0;
    ...
    /* do stuff */
    ...
    return (++count);
}
```

Arithmetic, Relational and Logical Operators

- Arithmetic Operators

- Modulo $x \% y$ produces the remainder when x is divided by y
 - Modulo cannot be applied to a float or double
- Note that \wedge is not a power operator, but a bitwise exclusive OR operator (see later).

- Relational and Logical Operators

- The numeric value of a relational or logical expression is 1 if true, and 0 otherwise
- Logical AND and OR stop evaluation as soon as the truth or falsehood of the expression is determined
- The unary negation operator converts a non-zero operand into 0, and zero operand into 1

Operator	Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo	%

Operator	Symbol
Greater Than	>
Greater Than or Equals	>=
Less Than	<
Less Than or Equals	<=
Equals (equivalent)	==
Not Equals (equivalent)	!=
Logical OR	&&
Logical AND	
Logical NOT	!

```
if (!valid)
```

is often used instead of

```
if (valid == 0)
```


chars and ints

- chars are just ints, so char arithmetic is valid
- This allows a number of “tricks” when handling chars (see below)
 - These can be character set specific
 - However, most architectures are now ASCII based...
- Several functions are also available within the `<ctype.h>` library
 - return non-zero for true or 0 for false

```
/* naive atoi: convert s to integer */
int atoi(char s[]) {
    int i, n;
    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

```
/* lower: convert single character
 * c to lower case; ASCII only */
int lower(int c) {
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Type conversion

- When an operator has different types, they are converted to a common type according to simple rules
 - “lower” operands are promoted into “higher” types, e.g.
 - If either operand is long double, convert the other to long double.
 - Otherwise, if either operand is double, convert the other to double.
 - Otherwise, if either operand is float, convert the other to float.
 - Otherwise, convert char and short to int.
 - Then, if either operand is long, convert the other to long.
 - Note, that if floats are converted into ints, the fractional part is truncated.
 - Values on the right hand side of an assignment are converted into the left type
 - e.g `int i=5.3; /* i has the value 5 */`

Forcing a type conversion through a cast

- Explicit type conversions can be forced in any expression using a cast
 - *(type name) expression*
- This changes the type of the result of the expression to the specified type (in parentheses).
 - Useful if you want to call an expression on something of a different type
- Casts become more important when using pointers
 - A cast tells the compiler how to interpret a variable
 - The type `void *` can be used to “anonymously” pass a pointer, but the compiler cannot do anything until it is cast to another object type

Example of type conversion

```
#include <stdio.h>
#include <limits.h>

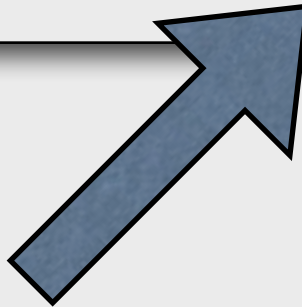
int main() {
    short sm = SHRT_MAX;
    unsigned short um = USHRT_MAX;
    char c;

    short sr = sm + 1;
    short ur = um + 1;
    c = sm;

    printf("unsigned short um = %d; um + 1 = %d\n", um, ur);
    printf("short sm = %d; sm + 1 = %d\n", sm, sr);
    printf("char c = sm; c = %d\n", c);
    printf("(double) sm = %f\n", (double) sm);

    return 0;
}
```

```
% ./a.out
unsigned short um = 65535; um + 1 = 0
short sm = 32767; sm + 1 = -32768
char c = sm; c = -1
(double) sm = 32767.000000
%
```



Increment / Decrement Operators

- **Increment** operator: adds 1 to the operand
- **Decrement** operator: subtracts 1 from the operand
- Both can be used as either prefix or postfix operators:
 - ++n increments n before its value is used in the expression
 - n++ increments n after its value is used in the expression
- In the example below, x = 5, whereas y = 7

```
...  
n = 5;  
x = n++;      /* n now has the value 6 */  
y = ++n;     /* n has the value 7 */  
...
```

Example code using the increment operator

The function takes a string (char array) *s*, and modifies that array to remove all values of *c* found in *s*.

NOTE: that the array *s* is side-affected; no value is returned from the function (i.e. its return value is void)

Traverse through *s* until the end of string character is found. The index *i* is used to check a character, whereas *j* refers to the character that is modified.

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c) {
    int i, j;
    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];      /* copy the value of s[i] into s[j]
                               * and then increment j
                               */
    s[j] = '\0';              /* terminate the string */
}
```

Finally we add the end of string character at position *j*.

NOTE that if characters have been deleted, then $j < i$.

We only copy characters we want to copy (i.e. we ignore characters of type *c*). *j* is only incremented when we do this copying.

Bitwise Operators

- Operators for bit manipulation
 - Only applicable to certain primitive types
 - char, short, int, long
- Often used for bit masking
 - AND operator used to mask off defined bits
 - OR operator used to switch on bits
- Shift operators perform left/right shifts
 - left shift fills vacated spaces with zeros
 - right shift on unsigned will fill with zero

Operator	Symbol
Bitwise AND	&
Bitwise inclusive OR	
Bitwise exclusive OR	^
Left shift	<<
Right shift	>>
One's complement (unary)	~

Applications

Bitwise operators are commonly used in low-level programming, such as device drivers etc. System calls that may take a lot of options often use a bitwise parameter, constructed by ORing a number of constants; e.g. setting file permissions

Assignment Operators

- Assignment expressions that modify variables by a factor of themselves can be shortened using *assignment operators*
 - e.g. `i = i + 2` can be rewritten as `i += 2`;
- If `expr1` and `expr2` are expressions
 - `expr1 op= expr2`
- Is equivalent to:
 - `expr1 = (expr1) op (expr2)`

Operator	Description	Example (x=6)
<code>+=</code>	Addition	<code>x += 4; /* x == 10 */</code>
<code>-=</code>	Subtraction	<code>x -= 2; /* x == 4 */</code>
<code>*=</code>	Multiplication	<code>x *= 3; /* x == 18 */</code>
<code>/=</code>	Division	<code>x /= 3; /* x == 2 */</code>
<code>%=</code>	Modulo	<code>x %= 5; /* x == 1 */</code>
<code><<=</code>	Bitwise Shift Left	<code>x <<= 2; /* x == 24 */</code>
<code>>>=</code>	Bitwise Shift Right	<code>x >>= 1; /* x == 3 */</code>
<code>&=</code>	Bitwise AND	<code>x &= 3; /* x == 2 */</code>
<code>^=</code>	Bitwise Exclusive OR	<code>x ^= 11; /* x == 15 */</code>
<code> =</code>	Bitwise Inclusive OR	<code>x = 11; /* x == 13 */</code>

Control Flow

- Almost all of the control flow constructs used in Java are based on the ones in C
 - These include:
 - Conditional Constructs
 - If-Else, Else-If, Switch
 - Looping Constructs
 - while, for, do-while
 - Flow constructs
 - break, continue, return, goto and labels
 - These constructs operate over:
 - A single statement terminated by a semicolon - ;
 - A block of code surround by left and right braces - { }

If-Else and Else-If

- These are used to express decisions, based on the truth value of the expression
 - The first statement is executed if the expression is true
 - The else part is optional, but is executed if the expression is false
 - Further If statements can follow the else part (see Else-IF)

If-Else

```
if (expression)  
    statement1  
else  
    statement2
```

Else-IF

```
if (expression)  
    statement1  
else if (expression)  
    statement2  
else if (expression)  
    statement3  
else  
    statement4
```

If-Else Ambiguity

- Because the else part is optional, ambiguities can occur when nesting if statements

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = n;
```

- The else statement is associated with the closest (inner) if statement.
 - This is indicated by the indentation in the above example
 - Blocks can be used to change this behaviour

```
if (n > 0) {
    if (a > b)
        z = a;
} else
    z = n;
```

switch

switch

```
switch (expression) {  
    case const-expr: statements  
    case const-expr: statements  
    default: statements  
}
```

- Used for multiway decision tests
 - Each case is labelled by a constant expression (*const-expr*)
 - The switch expression is then compared with each *const-expr*
 - When one is found, the following code is then executed
 - An optional default case can be used when no other expressions are matched.
 - Cases and defaults can occur in any order

while & do-while

while

```
while (expression)  
    statement
```

do-while

```
do  
    statement  
while (expression)
```

- Loop, until expression is true
 - They differ only in when the expression is tested
 - While
 - Evaluate the expression. If true, then execute the statement/body
 - Once complete, return to the expression and repeat...
 - Do-While
 - Execute the statement/body. Then evaluate the expression
 - If true, then repeat the execution of the statement/body...
- While is used more than do-while
 - However, do-while is invaluable when the body must be executed at least once.

for

for

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

- For loops are similar to while loops, but more compact.
- Three expressions:
 - for (*expr*₁; *expr*₂; *expr*₃)
 - **expr**₁ is executed once before the loop
 - **expr**₂ is the condition that is evaluated prior to executing the statement
 - **expr**₃ is executed after the statement, prior to re-evaluating the condition *expr*₂
 - Any of the expressions can be omitted
 - An expression may consist of multiple expressions, separated by a comma

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

...is equivalent to...

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

Example of using commas in for loops

The function takes a string (char array) `s`, and reverses the order of characters in the string.

NOTE: that the array `s` is side-affected; no value is returned from the function (i.e. its return value is void)

```
#include <string.h>

/* reverse: reverse string s in place */
void reverse(char s[]) {
    int c, i, j;

    for (i=0, j=strlen(s)-1; i<j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

The integers `i` and `j` are used to count from each end of the array.

- `i` is initialised to the first character (at position 0)
- `j` is initialised to the last character (at `strlen(s)-1`)

NOTE: `strlen(s)` returns the length of a string in a char array, by counting the number of characters before the null (end of string) character.

The first and last character are swapped, followed by the first+1 and last -1, etc. This continues until they meet in the middle.

Break and Continue

- Sometimes it is convenient to exit from a loop without evaluating the condition
- **Continue** suspends the current iteration of a loop, forcing the condition evaluation and (if true) starting the next loop iteration

```
for (i=0; i<n; i++)
    if (a[i] < 0) /* skip negative elements */
        continue;
    ... /* do positive elements */
```

- **Break** exits the loop immediately

```
/* trim: remover trailing blanks, tabs, newlines */
int trim(char s[]) {
    int n;
    for (n=strlen(s)-1; n>=0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n; /* return the length of the trimmed string */
}
```


Goto and labels

- Goto allows the process to jump to a label in the code
 - Typically makes code hard to understand and maintain
 - Rarely used, and considered bad practice
- Whilst formally not necessary, they sometimes make code easier to manage, especially for error handling with nested structures, when break cannot be used

```
for (...)
    for (...) {
        if (disaster)
            goto error;
    }
    ...
error:
    /* clean up the mess */
```

```
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        if (a[i] == b[j])
            goto found;
...
/* code to execute if no common element is found in both a[] and b[] */
...
found:
    /* got one! a[i] == b[j] */
    ...
```

Return

Return

```
return expression;
```

- The `return` statement is used for returning a value from the called function
 - May appear anywhere in the function, but will terminate the function's execution.
 - `return` can also be used to terminate the function without specifying a return value
- However, the calling function can ignore the returned value!
- If no return statement is used, execution will terminate, and no value will be returned
- If a return-type is specified, but no value is returned, then garbage will be returned
 - Although this is not illegal, a compiler warning will probably be generated.

```
if (y > x)
    return y;
...
return x+1;
}
```

To Summarise

- We've now covered the basics of the language.
 - Variables, constants, externs and statics
 - Data types and enumerated types
 - Declarations and definitions
 - Functions and function prototypes
 - Operators
 - Type conversion
 - Conditional, flow, and looping constructs
- The next slide set will explore pointers, arrays, memory management and data structures

Exercises

- Exercise 2-3. Write a function `htoi(s)`, which converts a string of hexadecimal digits (including an optional `0x` or `0X`) into its equivalent integer value. The allowable digits are `0` through `9`, `a` through `f`, and `A` through `F`.
- Exercise 2-4. Write an alternative version of `squeeze(s1,s2)` that deletes each character in `s1` that matches any character in the string `s2`.
- Exercise 3-2. Write a function `escape(s,t)` that converts characters like newline and tab into visible escape sequences like `\n` and `\t` as it copies the string `t` to `s`. Use a switch. Write a function for the other direction as well, converting escape sequences into the real characters.
- Exercise 3-3. Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally.