6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

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

C is static compiled.

- C programs are compiled → machine code before execution.
- compiled code is self-contained doesn't require run-time envir. to be present while running.
- Python, Javascript, are DYNAMICALLY interpreted languages. they require an interpretor / run-time envir to be executed.

C is static typed

- The data-types of vars/exps. are determined at compile time
- These types are fixed throughout the execution of program.
- In contrast python, Javascript type of variable is determined at runtime.

```
int main(void)
{
   int a = 5;
   char c = 'm';
   float p = 3.14;
   printf("value: %d", x);
   c = "bar";                    → ERROR at compile time.
   printf("value: %f", x);  →
}
```

```c
#include <stdio.h>

int main() {
    int n = 10;
    int i = 1;
    int sum = 0;
loop:
    if (i > n) goto done;
    sum += i;
    i++;
    goto loop;
done:
    printf("The sum of the first %d integers is %d\n", n, sum);
    return 0;
}
```

Un-structured

- It lacks modular building blocks.
- control flow is NOT as clear as on RHS.

```c
#include <stdio.h>

int main() {
    int n = 10;
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    printf("The sum of the first %d integers is %d\n", n, sum);
    return 0;
}
```

Structured

- C provides program structure which is
  - modular
  - abstraction is possible.
- Clear control flow can be reasoned.

```c
#include <stdio.h>

int sum(int n) {
    return (n * (n + 1)) / 2;
}

int main() {
    int n = 10;
    int result = sum(n);
    printf("The sum of the first %d positive integers is %d\n", n, result);
    return 0;
}
```

Declarative

- In this approach the focus is ON what result should be using a math formula. NOT how to calc. it.

```c
#include <stdio.h>

int sum(int n) {
    int i, result = 0;
    for (i = 1; i <= n; i++) {
        result += i;
    }
    return result;
}

int main() {
    int n = 10;
    int result = sum(n);
    printf("The sum of the first %d positive integers is %d\n", n, result);
    return 0;
}
```

Imperative

- In this approach a step-by-step procedure can be seen.
- Focus is on how to calculate the sum.

Review: Basics

- Variable declarations: `int i; float f;`
- Intialization: `char c='A'; int x=y=10;`
- Operators: `+,-,*,/,%`
- Expressions: `int x,y,z; x=y*2+z*3;`
- Function: `int factorial (int n); /*function takes int, returns int*/`

6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

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 (?:)

```java
public class Main {
    public static void main(String[] args) {
        int i = 10;
        float f = 3.14f;
        System.out.printf("i + f = %f\n", (float)i + f);
    }
}
```

Strongly typed (java).

- In java we have to explicitly convert types.

```c
#include <stdio.h>

int main() {
    int i = 10;
    float f = 3.14;
    printf("i + f = %f\n", i + f);
    return 0;
}
```

Weakly typed (C)

- here the program implicitly converts the integer (i) → float to perform addition.

^ ^

implicit conversion.

```c
#include <stdio.h>

int main() {
    int i = 10;
    float f = 3.14;

    // force cast i to float and add it to f
    float result = (float)i + f;

    printf("result = %f\n", result);
    return 0;
}
```

Forced casting is allowed in C (dangerous)

- here we force cast i → float.
- ans we get here is correct.

```c
#include <stdio.h>

int main() {
    int i = 1000000000;
    char c = (char)i;

    printf("i = %d\n", i);
    printf("c = %d\n", c);

    return 0;
}
```

- char c clearly cant HOLD the large value
- Result is C lang implementation dependant.
- it is good practice to avoid forced casting.

```c
#include <stdio.h>

int main() {
    // Unary operators
    int x = 10, y = -5;
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("++x = %d\n", ++x); // Pre-increment, x is now 11
    printf("y++ = %d\n", y++); // Post-increment, y is now -4, but this statement returns -5
    printf("--y = %d\n", --y);
    printf("~x = %d\n", ~x);
    printf("!y = %d\n", !y);
    printf("-x = %d\n", -x);

    // Binary operators
    int a = 7, b = 3;
    printf("\na = %d, b = %d\n", a, b);
    printf("a + b = %d\n", a + b);
    printf("a - b = %d\n", a - b);
    printf("a * b = %d\n", a * b);
    printf("a / b = %d\n", a / b);
    printf("a %% b = %d\n", a % b);
    printf("a & b = %d\n", a & b);
    printf("a | b = %d\n", a | b);
    printf("a ^ b = %d\n", a ^ b);
    printf("a << 1 = %d\n", a << 1);
    printf("a >> 1 = %d\n", a >> 1);

    // Ternary operator
    int age = 17;
    char *status = (age >= 18) ? "Adult" : "Minor";
    printf("\nAge: %d, Status: %s\n", age, status);

    return 0;
}
```

```
x = 10
y = -5
++x = 11
y++ = -5
--y = -6
~x = -12
!y = 0
-x = -11

a = 7, b = 3
a + b = 10
a - b = 4
a * b = 21
a / b = 2
a % b = 1
a & b = 3
a | b = 7
a ^ b = 4
a << 1 = 14
a >> 1 = 3

Age: 17, Status: Minor
```

Definitions (contd.)

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 system's 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.

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)

Data types and sizes

C has a small family of datatypes.

- Numeric (int,float,double)
- Character (char)
- User defined (struct,union)

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** 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.
- Titbit: Flickr changed from unsigned long $(2^{32} - 1)$ to string two years ago.

Big endian vs. little endian

Also different on 32-bit, 64-bit, 16-bit machines.

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.

Big endian vs. little endian (cont.)

- 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.

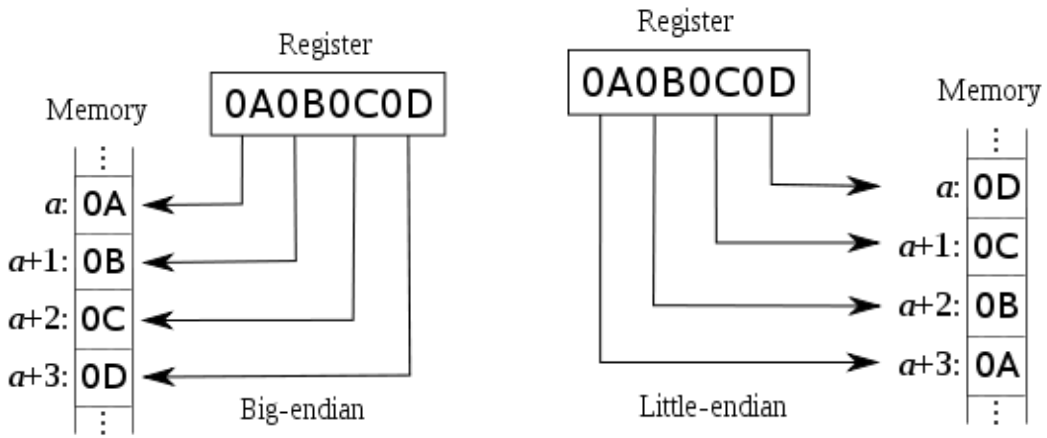


Figure: (from `http://en.wikipedia.org/wiki/Little_endian`)

Big Endian

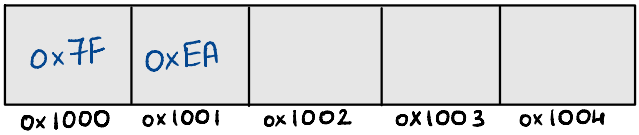
MSB (most significant byte) is stored on the LOWER memory address.

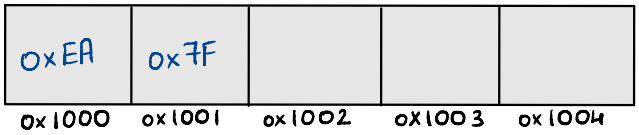
Little Endian

LSB (least significant byte) is stored on the LOWER mem. address.

0x7FEA

MSB — LSB






```c
#include <stdio.h>

int main()
{
    unsigned int num = 0x12345678;
    char *ptr = (char *)&num;

    printf("\n\n  Value of num: 0x%X\n", num);
    printf("Address of num: %p\n", &num);
    printf("\n\nByte Order in memory is: \n\n");
    printf("\t--------------------------------\n");
    for (int i = 0; i < sizeof(num); i++)
    {
        printf("\t|  %p   |   0x%02X   |\n", (void *)(ptr + i), *(ptr + i));
        printf("\t--------------------------------\n");
    }

    if (*ptr == 0x78)
    {
        printf("\n\nThis machine is little-endian.\n");
    }
    else if (*ptr == 0x12)
    {
        printf("\n\nThis machine is big-endian.\n");
    }
    else
    {
        printf("\n\nUnable to determine endianness.\n");
    }
    return 0;
}
```

casting here is IMP because char* → 1 Byte
we want pointer arithmatic to go 1 byte by byte.

%02X

X → uppercase hex.
2 → min. width 2
0 → padding with 0.

```
  Value of num: 0x12345678
Address of num: 0x7ffe9a3ab798


Byte Order in memory is:

        --------------------------------
        |  0x7ffe9a3ab798   |   0x78   |
        --------------------------------
        |  0x7ffe9a3ab799   |   0x56   |
        --------------------------------
        |  0x7ffe9a3ab79a   |   0x34   |
        --------------------------------
        |  0x7ffe9a3ab79b   |   0x12   |
        --------------------------------


This machine is little-endian.
```

Constants

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

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

```
float c = 3.14F;                  // Float
long double d = 3.14L;            // Long double
int x = 123U;                     // Unsigned integer
long y = 456L;                    // Long integer
unsigned long z = 789UL;          // Unsigned long integer
long long a = 123LL;              // Long long integer
unsigned long long b = 456ULL;    // Unsigned long long integer


F or f                    : Float
L or l                    : Long double
U or u                    : Unsigned integer
L or l                    : Long integer
UL or ul                  : Unsigned long integer
LL or ll                  : Long long integer
ULL or Ull or uLL or ull: Unsigned long long integer
```

float pi = 3.14159;

- This is perfectly valid.
- But in some cases it could happen that the value is implicity converted → Double
- To avoid such conversions.

Its best to explicity specify a suffix.

float pi = 3.14159 F or f;

Constants (contd.)

uppercase prints in upper case letters.

printf("Octal %o or %O");
printf("Hex %x or %X");

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

2 3

Declarations

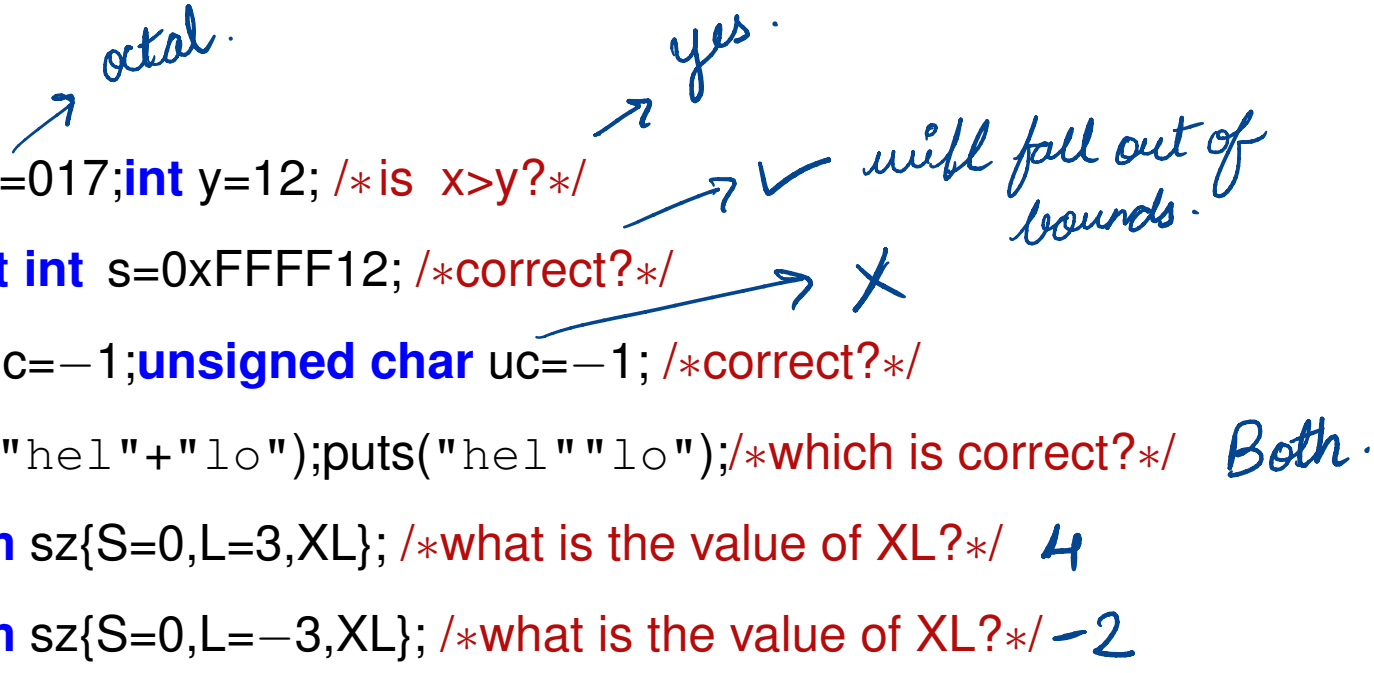
The general format for a declaration is
type variable-name [=*value*]
Examples:

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

int x=y=z=a=b=0;

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?*/



6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

Arithmetic operators

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

Arithmetic operators (contd.)

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

Relational Operators

TRUE : NON-0 value : 1, 2, 3 ....

FALSE : 0.

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 */ 2.99>3 /*evaluates to 0 */
>=	greater than or equal to	3>=3; /*evaluates to 1 */ 2.99>=3 /*evaluates to 0 */
<	lesser than	3<3; /*evaluates to 0 */ 'A'<'B' /*evaluates to 1*/
<=	lesser than or equal to	3<=3; /*evaluates to 1 */ 3.99<3 /*evaluates to 0 */

Relational Operators

Testing equality is one of the most commonly used relational operator.

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

Gotchas:

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

Logical operators

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

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=x−1
- 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=x−1. x is evaluated **before** it is decremented.

assign first then ++ or --

Increment and decrement operators

Prefix:

- ++x is a short cut for x=x+1
- −−x is a short cut for x=x−1
- 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=x−1;y=x;. x is evaluate **after** it is decremented.

Bitwise Operators

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

Assignment Operators

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

- x=x+1
- x=x∗10
- 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=x−1∗/
- x∗=10 /∗is the same as x=x∗10 ∗/
- x/=2 /∗ is the same as x=x/2
- x%=2 /∗is the same as x=x%2

Conditional Expression

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

```
if (cond)
   x=<expra>;
else
   x=<exprb>;
```

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

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

```
if (x>0)
   sign=1
else
   sign=-1
```

```
isodd=x%2==1?1:0;
```

```
if (x%2==1)
   isodd=1
else
   isodd=0
```

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

6.087 Lecture 2 – January 12, 2010

- Review
- Variables and data types
- Operators
- Epilogue

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 manupilations of character variables.

```
isupper=(c>='A' && c<='Z')?1:0; /*c and literal constants
                                  are converted to int*/
if(!isupper)
  c=c-'a'+'A'; /*subtraction is possible
                 because of integer conversion*/
```

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

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=x*3+2 /*same as y=(x*3)+2*/
- x!=0 && y==0 /*same as (x!=0) && (y==0)*/
- d= c>='0' && c<='9' /*same as d=(c>='0') && (c<='9')*/

6.087 Practical Programming in C

IAP 2010