A Fast Review of C Essentials Part I

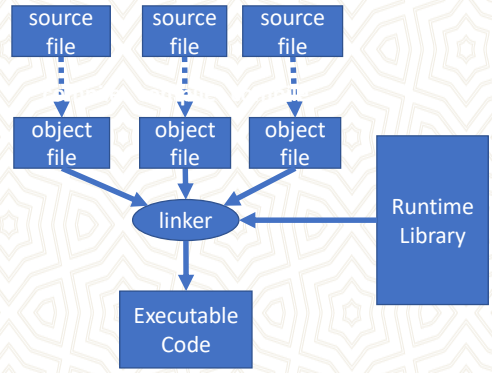
Structural Programming
by Z. Cihan TAYSI
additions by Yunus Emre SELÇUK

Outline

- Program development
- C Essentials
  - Variables & constants
  - Names
  - Functions
  - Formatting
  - Comments
  - Preprocessor
- Data types
- Mixing types

Program Development

- The task of compiler is to translate source code into machine code
- The compiler's input is ***source code*** and its output is ***object code***.
- The linker combines separate object files into a single file
- The linker also links in the functions from the runtime library, if necessary.
- Linking usually handled automatically.

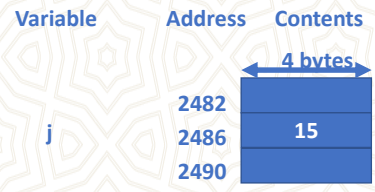


Program Development CONT'D

- One of the reasons C is such a small language is that it defers many operations to ***a large runtime library.***
- The runtime library is a collection of object files
  - Each file contains the machine instructions for a function that performs one of a wide variety of services
    - The functions are divided into groups, such as I/O, memory management, mathematical operations, and string manipulation.
  - For each group there is a source file, called a ***header file***, that contains information you need to use these functions
    - by convention , the names for header files end with *.h* extention
- For example, one of the I/O runtime routines, called ***printf()***, enables you to display data on your terminal. To use this function you must enter the following line in your source file
  - #include <stdio.h>

Variables & Constants

- The statement
  - j = 5 + 10;
- ***A constant*** is a value that never changes
- ***A variable*** achieves its variableness by representing a location, ***or address***, in computer memory.



Names

- In the C language, you can name just about anything
  - variables, constants, functions, and even location in a program.
- Names may contain
  - letters, numbers, and the underscore character (_)
  - **but must start with a letter or underscore**...
- The C language is ***case sensitive*** which means that it differentiates between lowercase and uppercase letters
  - VaR, var, VAR
- A name **can NOT be** the same as one of the ***reserved keywords***.

Names cont'd

- **LEGAL NAMES**
  - j
  - j5
  - __sesquipedalial_name_system_name
  - UpPeR_aNd_LoWeR_cAsE_nAmE
- **ILLEGAL NAMES**
  - 5j
  - $name
  - int
  - bad%#*@name

Names cont'd

- reserved keywords = illegal names cont.'d:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void[*1]
default	goto	sizeof	volatile
do	if	static	while

Expressions

- An ***expression*** is any combination of operators, numbers, and names that donates the computation of a value.
- **Examples**
  - 5 A constant
  - j A variable
  - 5 + j A constant plus a variable
  - f() A function call
  - f()/4 A function call, whose result is divided by a constant

Assignment Statements

- The left hand side of an assignment statement, called an ***lvalue***, must evaluate to a memory address that can hold a value.
- The expression on the right-hand side of the assignment operator is sometimes called an ***rvalue***.

answer = num * num;

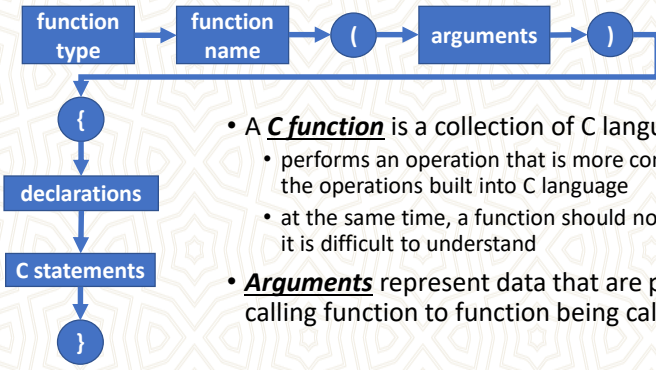
num * num = answer;

Comments

- A comment is text that you include in a source file to explain what the code is doing!
  - Comments are for human readers – compiler ignores them!
- The C language allows you to enter comments between the symbols /* and */
- Nested comments are NOT supported
- **What to comment ?**
  - Function header
  - changes in the code

```
/* square()
 * Author : P. Margolis
 * Initial coding : 3/87
 * Params : an integer
 * Returns : square of its parameter
 */
```


Functions



- A ***C function*** is a collection of C language operations.
  - performs an operation that is more complex than any of the operations built into C language
  - at the same time, a function should not be so complex that it is difficult to understand
- ***Arguments*** represent data that are passed from calling function to function being called.

Functions

- You can write your own functions and you should do so!
  - Grouping statements that execute a sub-task under a function leads to modular software
  - You can reuse functions in different programs
  - Functions avoid duplicate code that needs to be corrected in multiple places of the entire program if a bug removal or change request emerges.
    - Bugs and requirement changes are inevitable in software development!

Functions

- You should declare a function before it can be used …

```
int combination( int, int ); //This is also called allusion
void aTaskThatNeedsCombination( ) {
      //some code
      c = combination(a, b);
      //more code
}
int combination( int a, int b ) {
      //necessary code
}
```

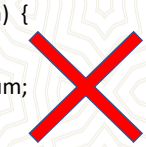

Functions

- … or the required function should be completely coded before it is called from another function.

```
int combination( int a, int b ) {
    //necessary code
}
void aTaskThatNeedsCombination( ) {
    //some code
    c = combination(a, b);
    //more code
}
```


Formatting Source Code

```
int square (int num) {
int answer;
answer = num * num;
return answer;
}
```



```
int square (int num) {
        int answer;
        answer = num * num;
        return answer;
}
```

```
int square (int num) {
 int
answer;
        answer  = num
* num;
return answer;
}
```

```
int square ( int num )
{
        int answer;
        answer = num * num;
        return answer;
}
```

The main() Function

- All C programs must contain a function called ***main()***, which is always the first function executed in a C program.
- It can take two arguments but we need to learn much more before going into details.
- When ***main()*** returns, the program is done.

```
int main ( ) {
        extern int square();
        int solution;
        solution = square(5);
        exit(0);
}
```

- The ***exit()*** function is a runtime library routine that causes a program to end, returning control to operating system.
  - If the argument to exit() is zero, it means that the program is ending normally without errors.
  - Non-zero arguments indicate abnormal termination of the program.
  - Calling exit() from main() is exactly the same as executing ***return*** statement.

printf() and scanf() Functions

```
int num;
scanf("%d", &num);
printf("num : %d\n", num);
```

- The printf() function can take any number of arguments.
  - The first argument called the ***format string***. It is enclosed in double quotes and may contain text and ***format specifiers***
- The scanf() function is the mirror image of printf(). Instead of printing data on the terminal, it reads data entered from keyboard.
  - The first argument is a format string.
  - **The major difference between scanf() and printf() is that the data item arguments must be lvalues**
  - **Scanf requires a memory address as 2nd parameter, hence comes the &**

Preprocessor

- The preprocessor executes automatically, when you compile your program
- All preprocessor directives begin with pound sign (#), which must be the first non-space character on the line.
  - unlike C statements a preprocessor directive ends with a newline, **NOT a semicolon**
- It is also capable of
  - macro processing
  - conditional compilation
  - debugging with built-in macros

Preprocessor cont'd

- The ***define*** facility
  - it is possible to associate a name with a constant
    - #define NOTHING 0
  - It is a common practice to all uppercase letters for constants
  - naming constants has two important benefits
    - it enable you to give a descriptive name to a nondescript number
    - it makes a program easier to change
  - be careful NOT to use them as variables
    - **NOTHING = j + 5**

Preprocessor cont'd

- The ***include*** facility
  - #include directive causes the compiler to read source text from another file as well as the file it is currently compiling
  - the #include command has two forms
    - #include <filename>
      - **the preprocessor looks in a special place designated by the operating system. This is where all system include files are kept.**
    - #include "filename"
      - **the preprocessor looks in the directory containing the source file. If it can not find the file, it searches for the file as if it had been enclosed in angle brackets!!!**

hello world!!!

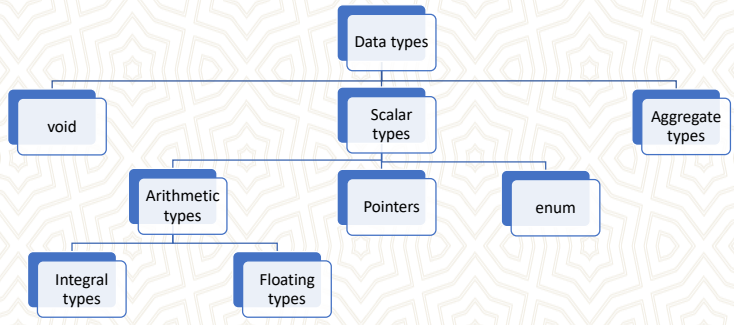
```
#include <stdio.h>

int main ( void ) {

        printf("Hello World...\n");
        return 0;
}
```

- include standard input output library
- start point of your program
- return a value to calling program
  - in this case 0 to show success?
- Hint: getch

Data Types



Data Types cont'd

- There are 9 reserved words for scalar data types
- Basic types
  - char, int, float, double, enum
- Qualifiers
  - long, short, signed, unsigned
- To declare j as an integer
  - int j;

char	**double**	**short**	**signed**
int	**enum**	**long**	**unsigned**
float			

- You can declare variables that have the same type in a single declaration
  - int j,k;
- ***All declarations in a block must appear before any executable statements***

Data Types cont'd (Void data type)

- The void data type is used primarily for indicating that a function does not return a value.
- A pointer can also be defined as having the void type, however this usage will not be covered in this course.
- The void data type can also be used when a function does not take any parameters. This notation is optional though, a simple () pair is enough.

Different Types of Integers

- The only requirement that the ANSI Standard makes is that a byte must be ***at least 8 bits long***, and that ints must be ***at least 16 bits long*** and must represent the "***natural***" size for computer.
  - natural: the number of bits that the CPU usually handles in a single instruction

Type	Size (in bytes)	Value Range	Format String
int	4	-2^{31} to $2^{31}-1$	%d
unsigned int	4	0 to $2^{32}-1$	%u
short int	2	-2^{15} to $2^{15}-1$	%hi
long int (just like int!)	4	-2^{31} to $2^{31}-1$	%li
long long int (now we are talking!)	8	-2^{64} to $2^{64}-1$	%lli
unsigned short int	2	0 to $2^{16}-1$	%hu
unsigned long int	4	0 to $2^{32}-1$	%lu
signed char	1	-2^{7} to $2^{7}-1$	%c
unsigned char (rather meaningless)	1	0 to $2^{8}-1$	%hhu

Format Strings for Integers

- A format string determines the representation of a value in output (printf) and the interpretation of a value in input (scanf).
- Try the following code with different values:

```
#include <stdio.h>

int main(int argc, char *argv[]){
        int sayi = 65;
        printf("     int \t%d\n",sayi);
        printf(" uns.int \t%u\n",sayi);
        printf(" srt.int \t%hi\n",sayi);
        printf(" lng.int \t%li\n",sayi);
        printf("usrt.int \t%hu\n",sayi);
        printf("ulng.int \t%llu\n",sayi);
        printf("     char\t%c\n",sayi);
        printf(" uns.char\t%hhu\n",sayi);

        system("PAUSE");
        return 0;
}
```


Different Types of Integers cont'd

- Integer constants
  - ***d***ecimal (%d), ***o***ctal (%o), He***x***adecimal (%x)

Decimal	Octal (leading 0 zero)	Hexadecimal (leading 0x zeroX, case insensitive)
3	003	0x3
8	010	0x8
15	017	0xF
16	020	0x10
21	025	0x15
-87	-0127	-0x57
255	0377	0xFF

- In general, an integer constant has type int, if its value can fit in an int. Otherwise it has type long int.
- Suffixes
  - u or U (for unsigned)
  - l or L (for long)

Floating Point Types

- to declare a variable capable of holding floating-point values
  - ***float (%f)***
  - ***Double (%lf)***
- The word ***double*** stands for double-precision
  - it is capable of representing about twice as much precision as a ***float***
  - A float generally requires ***4 bytes***, and a double generally requires ***8 bytes***
  - ***read more about limits in <limits.h>***
- Decimal point
  - 0.356
  - 5.0
  - 0.000001
  - .7
  - 7.
- ***Scientific notation (%e)***
  - 3e2
  - 5E-5
- Long double can be defined but they can become plain double in some computer platforms
- Refer to the source book and the Internet for different representation format modifiers (such as %5.7f)

Format Strings for Real Numbers

```
#include <stdio.h>

int main(int argc, char *argv[]){
        float ondalikli = 700.555;
        printf("     dbl \t%f\n",ondalikli);
        printf("     dbl \t%.3f\n",ondalikli);
        printf(" lng.dbl \t%lf\n",ondalikli);
        printf("     exp \t%e\n",ondalikli);

        system("PAUSE");
        return 0;
}
```

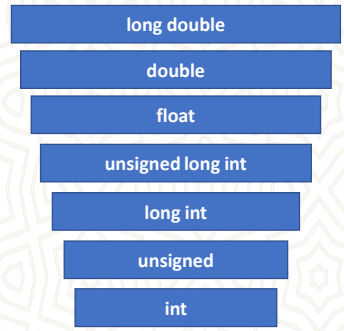

Initialization

- A declaration allocates memory for a variable, but it does not necessarily store an initial value at the location
  - ***If you read the value of such a variable before making an explicit assignment, the results are unpredictable***
- To initialize a variable, just include an assignment expression after the variable name
  - char ch = 'A' ;
- It is same as
  - char ch;
  - ch = 'A';

Mixing Types

- Implicit convertion
- Mixing signed and unsigned types
- Mixing integers with floating point types
- Explicit conversion

Mixing Types cont'd



Implicit Conversions

- When the compiler encounters an ***expression***, it divides it into ***subexpressions***, where each expression consists of one operator and one or more objects, called ***operands***, that are **bound to the operator**.
- **Ex :** 1 + 2.5 # involves two types, an int and a double
- **Ex :** – 3 / 4 + 2.5 # The expression contains three operators –, /, +
- 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.
  - For this conversion, compiler resorts to the hierarchy of data types. **(Please remember previous slide)**

Mixing Signed and Unsigned Variables

- The only difference between signed and unsigned integer types is the way they are interpreted.
  - They occupy same amount of storage
- 11101010
  - has a decimal value of -22 (in two's complement notation)
  - An unsigned char with the same binary representation has a decimal value of 234
- 10u – 15 = ?
  - – 5
  - 4,294,967,291

Mixing Integers with Floating Types

- Invisible conversions

```
int j;
float f;
j + f ;          // j is converted to float
j + f + 2.5;     // j and f both converted to double
```

- **Loss of precision**

```
j = 2.9;                        // j's value is 2
j = 999999999999.888888         // overflow
```


Explicit Conversions - Cast

```
int j=2, k=3;
float f;
f = k / j ;
```

- **Explicit conversion is called casting and is performed with a construct called *a cast***

```
f = (float) k / j;
```

- **To cast an expression, enter the target data type enclosed in parenthesis directly before expression**

Enumeration Data Type

```
enum { red, blue, green, yellow } color;
enum { bright, medium, dark } intensity;

color = yellow;        // OK
color = bright;        // Type conflict
intensity = bright;    // OK
intensity = blue;      // Type conflict
color = 1;             // Type conflict
color = green + blue; // Misleading usage
```

- ***Enumeration types*** enable you to declare variables and the set of named constants that can be legally stored in the variable.
- The default values start at zero and go up by one with each new name.
- You can override default values by specifying other values

void Data Type

- The void data type has two important purposes.
- The first is to indicate that a function does not return a value
  - void func (int a, int b);
- The second is to declare a generic pointer
  - **We will discuss it later !**

typedef

- ***typedef*** keyword lets you create your own names for data types.
- Semantically, the variable name becomes a synonym for the data type.
- By convention, typedef names are capitalized.

```
typedef long int INT32;

long int j;
INT32 j;
```


A Fast Review of C Essentials Part II

Structural Programming & Control Flow

by Z. Cihan TAYSI

Outline

- Operators
  - expressions, precedence, associativity
- Control flow
  - if, nested if, switch
  - Looping

Expressions

- **Constant expressions**
  - 5
  - 5 + 6 * 13 / 3.0
- **Integral expressions (int j,k)**
  - j
  - j / k * 3
  - k –'a'
  - 3 + (int) 5.0
- **Float expressions (double x,y)**
  - x / y *5
  - 3 + (float) 4
- **Pointer expressions (int * p)**
  - p
  - p+1
  - "abc"

Precedence & Associativity

- All operators have two important properties called ***precedence*** and ***associativity***.
  - Both properties affect how operands are attached to operators
- Operators with higher precedence have their operands bound, or grouped, to them before operators of lower precedence, regardless of the order in which they appear.
- In cases where operators have the same precedence, associativity (sometimes called binding) is used to determine the order in which operands grouped with operators.

- 2 + 3 * 4
- 3 * 4 + 2

- a + b – c;
- a = b = c;
- a < b < c

Precedence & Associativity

Class of operator	Operators in that class	Associativity	Precedence
primary	() [] -> .	Left-to-Right	HIGHEST ↑
unary	cast operator sizeof & (address of) * (dereference) - + ~ ++ -- !	Right-to-Left	
multiplicative	* / %	Left-to-Right	
additive	+ -	Left-to-Right	
shift	<< >>	Left-to-Right	
relational	< <= > >=	Left-to-Right	
equality	== !=	Left-to-Right	

Precedence & Associativity

Class of operator	Operators in that class	Associativity	Precedence
bitwise AND	&	Left-to-Right	↓
bitwise XOR (exclusive OR)	^	Left-to-Right	
bitwise OR (inclusive OR)	|	Left-to-Right	
logical AND	&&	Left-to-Right	
logical OR	||	Left-to-Right	
conditional	? :	Right-to-Left	
assignment	= += -= *= /= %= >>= <<= &= ^=	Right-to-Left	
comma	,	Left-to-Right	LOWEST

Parenthesis

- The compiler groups operands and operators that appear within the parentheses first, so you can use parentheses to specify a particular grouping order.
  - (2 – 3) * 4
  - 2 – (3 * 4)
- The inner most parentheses are evaluated first. The expression (3+1) and (8-4) are at the same depth, so they can be evaluated in either order.

  1 + ((3+1) / (8 – 4) – 5)
  1 + (4 / (8 – 4) – 5)
  1 + (4 / 4 – 5)
  1 + (1 – 5)
  1 + -4
  -3

Binary Arithmetic Operators

Operator	Symbol	Form	Operation
multiplication	*	**x * y**	x times y
division	/	**x / y**	x divided by y
remainder	%	**x % y**	remainder of x divided by y
addition	+	**x + y**	x plus y
subtraction	-	**x - y**	x minus y

The Remainder Operator

- Unlike other arithmetic operators, which accept both integer and floating point operands, the remainder operator accepts only integer operands!
- If either operand is negative, the remainder can be negative or positive, depending on the implementation
- The ANSI standard requires the following relationship to exist between the remainder and division operators
  - ***a*** equals ***a***%***b*** + (***a***/***b***) * ***b*** for any integral values of ***a*** and ***b***

Arithmetic Assignment Operators

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***
substract-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***

Arithmetic Assignment Operators

int m = 3, n = 4;
float x = 2.5, y = 1.0;

m += n + x – y
m /= x * n + y
n %= y + m
x += y -= m

m = (m + ((n+x) –y))
m = (m / ((x*n) + y))
n = (n % (y + m))
x = (x + (y = (y – m)))

Increment & Decrement Operators

Operator	Symbol	Form	Operation
postfix increment	++	a++	get value of a, then increment a
postfix decrement	--	a--	get value of a, then decrement a
prefix increment	++	++a	increment a, then get value of a
prefix decrement	--	--b	decrement a, then get value of a

Increment & Decrement Operators

```
main () {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", j++, k--);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

Postfix

```
main () {
        int j=5, k=5;
        printf("j: %d\t k : %d\n", ++j, --k);
        printf("j: %d\t k : %d\n", j, k);
        return 0;
}
```

Prefix

They work as they are intended, even in functions like printf !

Increment & Decrement Operators

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

s = m++ - --j	(m++) – (--j)	(s=2)
s = m += ++j * 2	m = (m + ((++j) * 2)	(s=3)
s = m++ * m++	(m++) * (m++)	(implementation-dependent)

Comma Operator

- Allows you to evaluate two or more distinct expressions wherever a single expression allowed!
- Ex : for (j = 0, k = 100; k – j > 0; j++, k--)
- Result is the value of the rightmost operand

Relational Operators

Operator	Symbol	Form	Result
greater than	>	**a > b**	1 if a is greater than b; else 0
less than	<	**a < b**	1 if a is less than b; else 0
greater than or equal to	>=	**a >= b**	1 if a is greater than or equal to b; else 0
less than or equal to	<=	**a < = b**	1 if a is less than or equal to b; else 0
equal to	==	**a == b**	1 if a is equal to b; else 0
not equal to	!=	**a != b**	1 if a is NOT equal to b; else 0

Relational Operators

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

j > m	j > m	(0)
m/n < x	(m / n) < x	(1)
j <= m >= n	((j <= m) >= n)	(1)
++j == m != y * 2	((++j) == m) != (y * 2)	(1)

Logical Operators

Operator	Symbol	Form	Result
logical AND	**&&**	**a && b**	1 if a and b are non zero; else 0
logical OR	**||**	**a || b**	1 if a or b is non zero; else 0
logical negation	**!**	**!a**	1 if a is zero; else 0

Logical Operators

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

float x=2.5, y=0.0;

Hint: All non-zero values are interpreted as TRUE, including negative values.

j && m	(j) && (m)	(0)
j < m && n < m	(j < m) && (n < m)	(1)
x * 5 && 5 || m / n	((x * 5) && 5) || (m / n)	(1)
!x || !n || m + n	((!x) || !n) || (m + n)	(0)

Bit Manipulation Operators

Operator	Symbol	Form	Result
right shift	>>	**x >> y**	x shifted right by y bits
left shift	<<	**x << y**	x shifted left by y bits
bitwise AND	&	**x & y**	x bitwise ANDed with y
bitwise inclusive OR	|	**x | y**	x bitwise ORed with y
bitwise exclusive OR (XOR)	^	**x ^ y**	x bitwise XORed with y
bitwise complement	~	**~x**	bitwise complement of x

Bit Manipulation Operators cont'd

Expression	Binary model of Left Operand	Binary model of the result	Result value
5 << 1	00000000 00000101	00000000 00001010	10
255 >> 3	00000000 11111111	00000000 00011111	31
8 << 10	00000000 00001000	00100000 00000000	2^{13}
1 << 15	00000000 00000001	10000000 00000000	-2^{15}

Expression	Binary model of Left Operand	Binary model of the result	Result value
5 >> 2	00000000 00000101	00000000 00000001	1
- 5 >> 2	11111111 11111011	11111111 11111110	-2

Bit Manipulation Operators cont'd

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 & 5722 (=1106)	0x0452	00000100 01010010

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 | 5722 (=14046)	0x36DE	00110110 11011110

Bit Manipulation Operators cont'd

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 ^ 5722 (=12940)	0x328C	00110010 10001100

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
~9430 (-9430)	0xDB29	11011011 00101001

Bitwise Assignment Operators

Operator	Symbol	Form	Result
right-shift-assign	>>=	**a >>= b**	Assign a>>b to a.
left-shift-assign	<<=	**a <<= b**	Assign a<<b to a.
AND-assign	&=	**a &= b**	Assign a&b to a.
OR-assign	|=	**a |= b**	Assign a|b to a.
XOR-assign	^=	**a ^= b**	Assign a^b to a.

cast & sizeof Operators

- Cast operator enables you to convert a value to a different type
- One of the use cases of cast is to promote an integer to a floating point number of ensure that the result of a division operation is not truncated.
  - 3 / 2
  - (float) 3 / 2
- The ***sizeof*** operator accepts two types of operands: an expression or a data type
  - **the expression may not have type function or void or be a bit field !**
- ***sizeof*** returns the number of bytes that operand occupies in memory
  - sizeof (3+4) returns the size of int
  - sizeof(short)

Conditional Operator (? :)

Operator	Symbol	Form	Operation
conditional	? :	a ? b : c	if a is nonzero result is b; otherwise result is c

- The conditional operator is the only ternary operator.
- It is really just a shorthand for a common type of ***if...else*** branch

```
z = ( (x<y) ? x : y );
```

```
if (x<y)
        z = x;
else
        z = y;
```

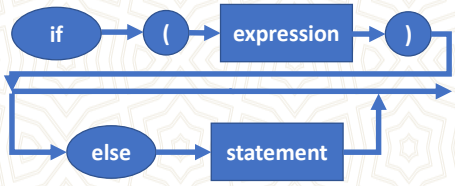
Memory Operators

Operator	Symbol	Form	Operation
address of	**&**	**&x**	Get the address of x.
dereference	*	***a**	Get the value of the object stored at address a.
array elements	[]	**x[5]**	Get the value of array element 5.
dot	.	**x.y**	Get the value of member y in structure x.
right-arrow	->	**p -> y**	Get the value of member y in the structure pointed to by p

Control Flow

- **Conditional branching**
  - if, nested IF
  - switch
- **Looping**
  - for
  - while
  - do...while

The if...else statement



Ex1 :

```
if (x)
   statement1;     // executed only if x is nonzero
statement2;        //always executed
```

Ex2:

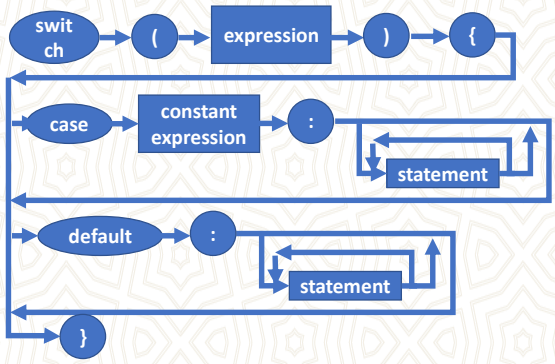
```
if (x)
   statement1;     // executed only if x is nonzero
else
   statement2;     // executed only if x is zero
statement3;        //always executed
```

Nested if statements

- Note that when an ***else*** is immediately followed by an ***if***,
  - they are usually placed on the same line.
  - this is commonly called an ***else if*** statement.
- Nested if statements create the problem of matching each else phrase to the right if statement.
  - This is often called the ***dangling else*** problem !
  - An else is always associated with the nearest previous if.

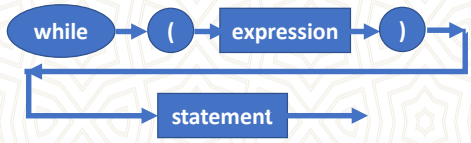
```
if(a<b)
   if(a<c)
         return a;
   else
         return c;
else if (b<c)
         return b;
else
   return c;
```

The switch Statement



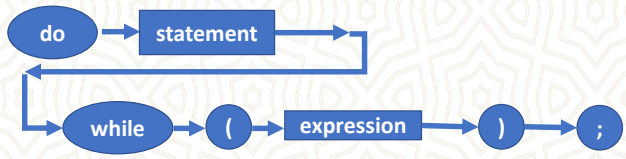
- The ***switch*** expression is evaluated,
  - if it matches one of the case labels, program flow continues with the statement that follows the matching case label.
  - If none of the case labels match the switch expression, program flow continues at the default label, **if exists!**
- No two case labels may have the same value!
- The default label need not be the last label, though it is good style to put it last

The while Statement



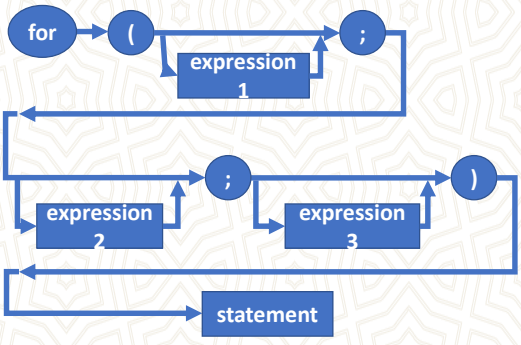
- First the expression is evaluated. If it is a ***nonzero*** value, statement is executed.
- After statement is executed, program control returns to the top of the while statement, and the process is repeated.
- This continues indefinitely until the expression evaluated to zero.

The do...while Statement



- The only difference between a do..while and a regular while loop is that the test condition is at the bottom of the loop.
  - This means that the program always executes statement ***at least one.***

The for Statement



- First, ***expression1*** is evaluated.
- Then ***expression2*** is evaluated.
  - This is the conditional part of the statement.
  - If ***expression2*** is **false**, program control exists the for statement.
  - If ***expression2*** is **true**, the ***statement*** is executed.
- After ***statement*** is executed, ***expression3*** is evaluated.
- Then the statement loops back to test ***expression2*** again.

NULL Statements

- It is possible to omit one of the expressions in a for loop, it is also possible to omit the body of the for loop.

```
for(c = getchar(); isspace(c); c = getchar());
```

- **ATTENTION**
- Placing a semicolon after the test condition causes compiler to execute a null statement whenever the if expression is **true**

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

Nested Loops

- It is possible to nest looping statements to any depth
- However, keep that in mind inner loops must finish before the outer loops can resume iterating
- It is also possible to nest control and loop statements together.

```
for( j = 1; j <= 10; j++)  {
        // outer loop
        printf("%5d|", j);
        for( k=1; k <=10; k++)  {
                printf("%5d", j*k);
        // inner loop
        }
        printf("\n");
}
```

break & continue & goto

- ***break***
  - We have already talked about it in ***switch statement***
  - When used in a loop, it causes program control jump to the statement following the loop
- ***continue***
  - continue statement provides a means for returning to the top of a loop earlier than normal.
  - it is useful, when you want to bypass the reminder of the loop for some reason.
  - Please do NOT use it in any of your C programs.
- ***goto***
  - goto statement is necessary in more rudimentary languages!
  - Please do NOT use it in any of your C programs.

Bu yansı ders notlarının düzeni için boş bırakılmıştır.