

BLM2031 YAPISAL PROGRAMLAMA – EYLÜL 2019

Sunan: Dr.Öğr.Üyesi Yunus Emre SELÇUK

GENEL BİLGİLER

DERS GRUPLARI

- Gr.1 Dr. Öğretim Üyesi Z. Cihan Tayşi (kapasite nedeniyle kapandı)
- Gr.2 Dr. Öğretim Üyesi H. İrem TÜRKMEN
- Gr.3 Dr. Öğretim Üyesi Zeyneb KURT
- Gr.4 Dr. Öğretim Üyesi Yunus Emre SELÇUK (Biz)

İLETİŞİM

- İletişim bilgileri
 - Oda : D-129
 - e-mail: yselcuk@yildiz.edu.tr, yunus.emre.selcuk.ytu@gmail.com
- İletişim için:
 - Öncelikle e-mail gönderiniz,
 - Yüz yüze görüşmemiz gerekiyor ise randevu isteyiniz

DERS NOTLARI

- <https://avesis.yildiz.edu.tr/yselecuk/dokumanlar>
 - Hazırlayan: Z. Cihan Tayşi (Bölüm1-5), Yunus Emre Selçuk (Bölüm 6)

1

BLM2031 YAPISAL PROGRAMLAMA– GENEL BİLGİLER

BAŞARIM DEĞERLENDİRME

- 1. ara sınav: 5/11/2019 (8.hafta) (bölümün sayfasında duyuracağı)
- 2. ara sınav: 3/12/2019 (12.hafta) vize programına göre)
- Final sınavı: Final haftasında (bölümün sayfasında duyurulacak)
- Puanlama (değişebilir):
 - Ara sınav %25*2, Lab %10, Final %40

DERS İÇERİĞİ

- Hatırlatma: C Dilinde Veri Tipleri / Kontrol Deyimleri / Döngüler / Diziler
- İşaretçiler: İşaretçiler Aritmetiği, diziler ve işaretçiler, İşaretçi Dizileri, Karakter Dizileri, İşaretçilerin İşaretçisi
- Dinamik Bellek Yönetimi ve Fonksiyonlar, Fonksiyon İşaretçileri, Özyineleme
- Yerel ve Global Değişkenler / Depolayıcı Sınıflar / Yapılar / Birlikler / Bitsel İşlemler
- Dosya işlemleri
- C Önışlemcileri ve Makrolar
- Statik ve Dinamik Kütüphaneler

2

BLM2031 YAPISAL PROGRAMLAMA – GENEL BİLGİLER

2018-2019 Güz Döneminden İtibaren Geçerli Olan Önemli Yenilikler

- Senato kararı uyarınca:
 - Öğrencinin ara sınav notunun %60'ı + Finalin %40'ı eğer "sayısal olarak" **40'ın altında** kalıyorsa öğrenci doğrudan "**FF notu**" ile **dersten kalmış sayılacaktır**.
 - Bütün öğrencilere derslere devam zorunluluğu gelmiştir (dersi tekrar alanların önceki notu ne olursa olsun).

2019-2020 Güz Döneminden İtibaren Geçerli Olan Önemli Yenilikler

- Senato kararı uyarınca:
 - Derslere ait devam durumu ilgili öğretim üyesi tarafından yarıyıl sonu sınavları başlamadan önce öğrenci bilgi sisteminde ilan edilir.
 - Devamsızlıktan kalan öğrenciler yarıyıl sonu sınavına giremezler ve bu öğrencilerin ilgili derse ait başarı notu (F0) olarak bilgi sistemine işlenir.

3

Bu yansı ders notlarının düzeni için boş bırakılmıştır.

4

A Fast Review of C Essentials Part I

Structural Programming
by
Z. Cihan TAYSI



Outline

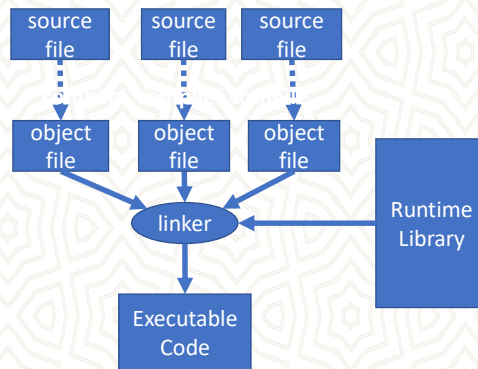
- Program development
- C Essentials
 - Functions
 - Variables & constants
 - Names
 - Formatting
 - Comments
 - Preprocessor
- Data types
- Mixing types

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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.



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



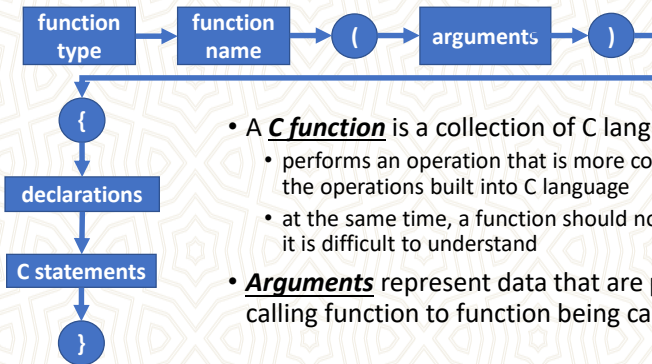
Program Development CONT'D

- 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** extension
- 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>`

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



C Essentials



- 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.



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.

Variable	Address	Contents
		4 bytes
	2482	
j	2486	15
	2490	



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
- __system_name
- sesquipedalial_name
- UpPeR_aNd_LoWeR_cAsE_nAmE

• ILLEGAL NAMES

- 5j
- \$name
- int
- bad%#*@name



Names cont'd

- reserved keywords = illegal names contd'.:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
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;

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Formatting Source Code

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



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



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



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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
 */
```



The main() Function

```
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 a `main()` function 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**



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>
```

- include standard input output library

```
int main ( void ) {
```

- start point of your program

```
printf("Hello World...\n");
```

```
return 0;
```

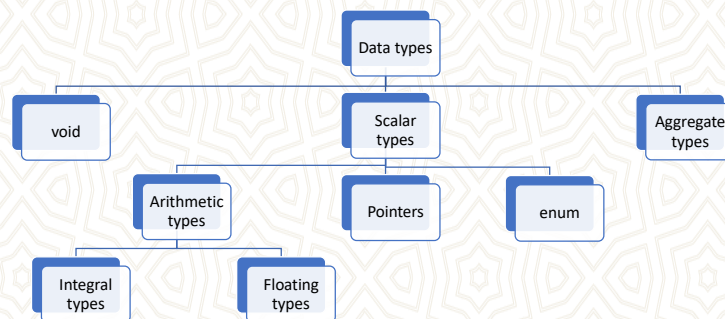
- return a value to calling program
 - in this case 0 to show success?

```
}
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Data Types



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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;
- 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**

char	double	short	signed
int	enum	long	unsigned
float			

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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 means the number of bits that the CPU usually handles in a single instruction

Type	Size (in bytes)	Value Range
int	4	-2^{31} to $2^{31} - 1$
short int	2	-2^{15} to $2^{15} - 1$
long int	4	-2^{31} to $2^{31} - 1$
unsigned short int	2	0 to $2^{16} - 1$
unsigned long int	4	0 to $2^{32} - 1$
signed char	1	-2^7 to $2^7 - 1$
unsigned char	1	0 to $2^8 - 1$

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Different Types of Integers cont'd

- Integer constants
 - Decimal, Octal, Hexadecimal
- 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
 - l or L

Decimal	Octal	Hexadecimal
3	003	0x3
8	010	0x8
15	017	0xF
16	020	0x10
21	025	0x15
-87	-0127	-0x57
255	0377	0xFF

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Floating Point Types

- to declare a variable capable of holding floating-point values
 - float
 - double
- 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
 - 3e2
 - 5E-5

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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';`

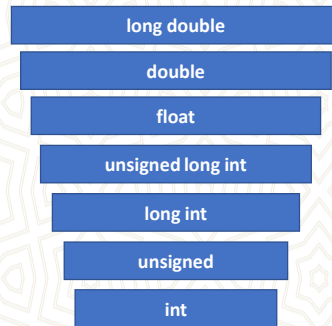


Mixing Types

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



Mixing Types cont'd



Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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 :** $-3 / 4 + 2.5$ # 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)**
- **Ex :** $1 + 2.5$ # involves two types, an int and a double

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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 = 2.5;           // j's value is 2
j = 999999999999.888888 // overflow
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Explicit Conversions - Cast

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

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

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

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

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



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
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
 - $j / k * 3$
 - $k - 'a'$
 - $3 + (\text{int}) 5.0$
- **Float expressions (double x,y)**
 - $x / y * 5$
 - $3 + (\text{float}) 4$
- **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.

$$2 + 3 * 4$$


$$3 * 4 + 2$$

$$a + b - c;$$

$$a = b = c;$$




Precedence & Associativity

Class of operator	Operators in that class	Associativity	Precedence
primary	() [] -> .	Left-to-Right	 <p>HIGHEST</p>
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	 <p>LOWEST</p>
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	



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
subtract-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 - 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



Increment & Decrement Operators

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

```
m++ --j
```

```
m += ++j * 2
```

```
m++ * m++
```

```
(m++) - (--j)
```

```
m = ( m + ((++j) * 2 )
```

```
(m++) * (m++)
```

(implementation-dependent)

(2)

(3)



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 (0)

m/n < x (1)

j <= m >= n (1)

++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 FALSE, including negative values.

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



Bit Manipulation Operators

Operator	Symbol	Form	Result
right shift	<code>>></code>	<code>x >> y</code>	x shifted right by y bits
left shift	<code><<</code>	<code>x << y</code>	x shifted left by y bits
bitwise AND	<code>&</code>	<code>x & y</code>	x bitwise ANDed with y
bitwise inclusive OR	<code> </code>	<code>x y</code>	x bitwise ORed with y
bitwise exclusive OR (XOR)	<code>^</code>	<code>x ^ y</code>	x bitwise XORed with y
bitwise complement	<code>~</code>	<code>~x</code>	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	11111111 11111011	00111111 11111110	$2^{13} - 1$
-5 >> 2	11111111 11111111	11111111 11111110	-2



Bit Manipulation Operators cont'd

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 & 5722	0x0452	00000100 01010010
Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 5722	0x36DE	00110110 11011110



Bit Manipulation Operators cont'd

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

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
~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.
 - $3 / 2$
 - $(\text{float}) 3 / 2$
- 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
 - $\text{sizeof}(3+5)$ returns the size of int
 - $\text{sizeof}(\text{short})$



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

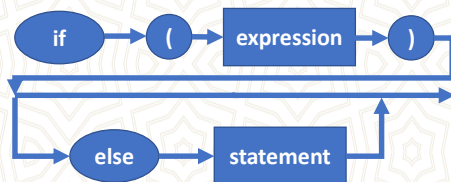


Control Flow

- **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
```



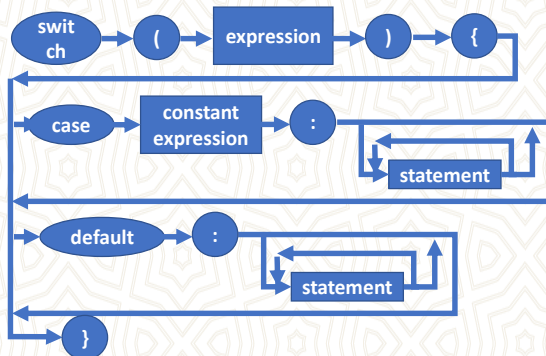
Nested if statements

- 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;
```



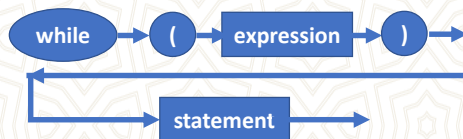
The switch Statement



- 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



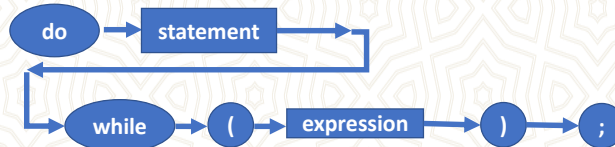
The while Statement



- 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.



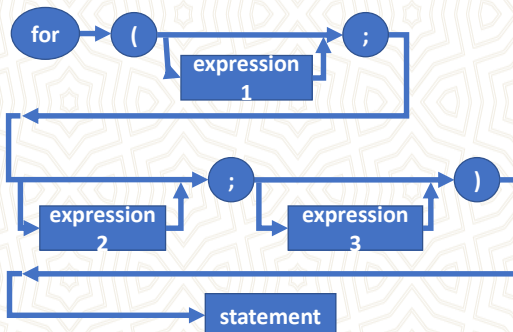
The do...while Statement



- 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.



Preprocessor (Part I)

Structural Programming

by Z. Cihan TAYSI

Corrections & additions by Yunus Emre SELÇUK



Outline

- Macro processing
 - Macro substitution
 - Removing a macro definition
 - Macros vs. functions
 - Built-in macros
- Conditional compilation
 - Testing macro existence
- Include facility
- Line control

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Macros

- All preprocessor directives begin with a pound sign (#), which must be the first nonspace character on the line
- **Unlike C statements, a macro command ends with a newline, not a semicolon.**
 - to span a macro over more than one line, enter a backslash immediately before the newline

```
#define LONG_MACRO "This is a very long  
macro that \  
spans two lines"
```



Macro Substitution

- The simplest and most common use of macros is to represent numeric constant values.
 - It is also possible to create function like macros

```
#define BUFF_LEN (512)  
  
char buf[BUFF_LEN];  
  
char buf[(512)];
```



Function Like Macros

- **Be careful not to use**
 - ';' at the end of macro
 - or '=' in macro definition
- No type checking for macro arguments
- Try to expand min macro example for three numbers

Example 1 :

```
#define MUL_BY_TWO(a) ((a) + (a))
```

```
j = MUL_BY_TWO(5);
```

```
f = MUL_BY_TWO(2.5);
```

Example 2 :

```
#define min(a, b) ( (a) < (b) ? (a) : (b) )
```



Side Effect

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

```
a = min(b++, c);
```

```
a = ((b++) < (c) ? (b++) : (c));
```

- Remember min macro

- Suppose, for instance, that we invoked the **min macro** like this!

- **The preprocessor translates this into !**



Macros vs. Functions

Advantages

- Macros are usually faster than functions, since they avoid the **function call overhead**.
- No type restriction is placed on arguments so that one macro **may serve for several data types**.

Disadvantages

- Macro arguments are reevaluated at each mention in the macro body, which can lead to unexpected behavior if an argument contains side effects!
- Function bodies are compiled once so that multiple calls to the same function can share the same code. Macros, on the other hand, are expanded each time they appear in a program.
- Though macros check the number of arguments, they don't check the argument types.
- It is more difficult to debug programs that contain macros, because the source code goes through an additional layer of translation.



Removing a Macro Definition

- Once defined a macro name retains its meaning until the end of the source file.
 - or until it is explicitly removed with an **#undef** directive.
- The most typical use of **#undef** is to remove a definition so you can **redefine** it.

```
#define FALSE 1
/* code requiring FALSE = 1 */
#undef FALSE
#define FALSE 0
/* code requiring FALSE = 0 */
```



Built-in Macros – I

- `__LINE__`
 - expands to the source file line number on which it is invoked.
- `__FILE__`
 - expands to the name of the file in which it is invoked.
- `__TIME__`
 - expands to the time of program compilation.
- `__DATE__`
 - expands to the date of program compilation.
- `__STDC__`
 - Expands to the constant 1, if the compiler conforms to the ANSI Standard.



Built-in Macros – II

```
void print_version( ) {  
  
    printf("This utility  
    compiled on %s at %s\n",  
    __TIME__,    __DATE__,  
    __FILE__);  
}
```

```
void print_version( ) {  
  
    printf("This meesage is at  
    %d line in %s\n",  
    __LINE__,  
    __FILE__);  
}
```



Conditional Compilation – I

- The preprocessor enables you to screen out portions of source code that you do not want compiled.
 - This is done through a set of preprocessor directives that are similar to *if* and *else* statements.
- The preprocessor versions are
 - #if, #else, #elif, #endif
- Conditional compilation particularly useful during the debugging stage of program development, since you can turn sections of your code on or off by changing the value of a macro
 - Most compilers have a command line option that lets you define macros before compilation begins.
 - gcc -DDEBUG=1 test.c

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Conditional Compilation – II

- The conditional expression in an #if or #elif statement **need not be** enclosed in parenthesis.
- Blocks of statements under the control of a conditional preprocessor directive **are not enclosed** in braces.
- Every #if block may contain **any number** of #elif blocks, but **no more than one** #else block, which should be **the last one!**
- **Every #if block must end with an #endif directive!**

```
#if x==1
    #undef x
    #define x 0
#elif x == 2
    #undef x
    #define x 3
#else
    #define y 4
#endif
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Conditional Compilation – III

```
#if defined TEST
```

```
#if defined macro_name
```

```
#if !defined macro_name
```

```
#if defined (TEST)
```

```
#ifdef macro_name
```

```
#ifndef macro_name
```

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Include Facility

- The `#include` command has two forms
 - `#include <filename>` : the preprocessor looks in a list of implementation-defined places for the file. In UNIX systems, standard include files are often located in the directory **`/usr/include`**
 - `#include "filename"` : the preprocessor looks for the file according to the file specification rules of operating system. If it can not find the file there, it searches for the file as if it had been enclosed in angle brackets.
- The `#include` command enables you to create common definition files, called header files, to be shared by several source files.
 - Traditionally have a `.h` extension
 - contain data structure definitions, macro definitions, function prototypes and global data

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü



Line Control

- Allows you to change compiler's knowledge of the current line number of the source file and the name of the source file.
- The #line feature is particularly useful for programs that produce C source text.
- For example yacc (Yet Another Compiler Compiler) is a UNIX utility that facilitates building compilers.
- We will not delve into further detail.

```
main() {  
#line 100  
printf("Current line :%d\nFilename :  
%s\n\n", __LINE__, __FILE__);  
#line 200 "new name"  
printf("Current line :%d\nFilename :  
%s\n\n", __LINE__, __FILE__);  
}
```



Bu yansı ders notlarının düzeni için boş bırakılmıştır.



Storage Classes

Structural Programming
by
Z. Cihan TAYSI



Outline

- Fixed vs. Automatic duration
- Scope
- Global variables
- The **register** specifier
- Storage classes
- Dynamic memory allocation



Fixed vs. Automatic Duration – I

- **Scope** is the technical term that denotes the region of the C source text in which a name's declaration is active.
- **Duration** describes the **lifetime** of a variable's memory storage.
 - Variables with **fixed duration** are guaranteed to retain their value even after their scope is exited.
 - There is **no such guarantee** for variables with **automatic duration**.
- **A fixed variable** is one that is stationary, whereas **an automatic variable** is one whose memory storage is automatically allocated during program execution.
- Local variables (whose scope limited to a block) are automatic by default. However, you can make them fixed by using keyword static in the declaration.
- The auto keyword explicitly makes a variable automatic, but it is rarely used since it is redundant.



Fixed vs. Automatic Duration – II

```
void increment ( void ) {  
    int j = 1;  
    static int k = 1;  
    j++;  
    k++;  
    printf("j : %d\t k:%d\n", j, k);  
}  
  
main ( void ) {  
    increment(); // j:2 k:2  
    increment(); // j:2 k:3  
    increment(); // j:2 k:4  
}
```

- Fixed variables initialized **only once**, whereas automatic variables are initialized **each time their block is reentered**.
- The **increment()** function increments two variables, **j** and **k**, both initialized to 1.
 - j has automatic duration by default
 - k has fixed duration because of the **static** keyword



Fixed vs. Automatic Duration – III

```
void increment ( void ) {  
    int j = 1;  
    static int k = 1;  
    j++;  
    k++;  
    printf("j : %d\t k:%d\n", j, k);  
}  
  
main ( void ) {  
    increment(); //j : 2 k : 2  
    increment(); //j : 2 k : 3  
    increment(); //j : 2 k : 4  
}
```

- When increment() is called the second time,
 - memory for *j* is reallocated and *j* is reinitialized to 1.
 - *k* has still maintained its memory address and is **NOT** reinitialized.
- Fixed variables get a default initial value of **zero**.



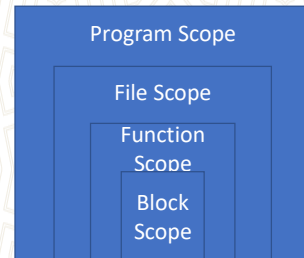
Scope – I

- The scope of a variable determines the region over which you can access the variable by name.
- There are four types of scope;
 - **Program scope** signifies that the variable is active among different source files that make up the entire executable program. Variables with program scope are often referred as **global variables**.
 - **File scope** signifies that the variable is active from its declaration point to the end of the source file.
 - **Function scope** signifies that the name is active from the beginning to the end of the function.
 - **Block scope** that the variable is active from its declaration point to the end of the block which it is declared.
 - A block is any series of statements enclosed in braces.
 - This includes compound statements as well as function bodies.



Scope – II

```
int i ;           // Program scope
static int j;     // File scope
func ( int k ) {  // function scope
    int m;        // function scope
    {
        int n; // Block scope
    }
}
```



Scope – III

```
foo ( void ) {
    int j, ar[20];
    ...
    {           // Begin debug code
        int j; // This j does not conflict with other j's.
        for(j=0; j <= 10; ++j)
            printf( "%d\t", ar[j] );
    }          // End debug code...
    ...
}
```

- A variable with a block scope can NOT be accessed outside its block.
- It is also possible to declare a variable within a nested block.
 - can be used for debugging purposes. **see the code on the left side of the slide!**
- Although variable hiding is useful in situations such as these, it can also lead to errors that are difficult to detect!



Scope – IV

- Function scope
 - The only names that have function scope are **goto** labels.
 - Labels are active from the beginning to the end of a function.
 - This means that labels must be unique within a function
 - Different functions may use the same label names without creating conflicts



Scope – V

- File & Program scope
 - Giving a variable file scope makes the variable active through out the rest of the file.
 - if a file contains more than one function, all of the functions following the declaration are able to use the variable.
 - To give a variable file scope, declare it outside a function with the **static** keyword.
 - Variable with program scope, called global variables, are visible to routines in other files as well as their own file.
 - To create a global variable, declare it outside a function without **static** keyword



Global Variables

- In general, you should avoid using global variables as much as possible!
 - they make a program harder to maintain, because they increase complexity
 - create potential for conflicts between modules
 - the only advantage of global variables is that they produce faster code
- There are two types of declarations, namely, ***definition and allusion***.
- An **allusion** looks just like a definition, but instead of allocating memory for a variable, it informs the compiler that a variable of the specified type exists but is defined elsewhere.
 - `extern int j;`
 - The `extern` keyword tells the compiler that the variables are defined elsewhere.



The *register* Specifier

- The ***register*** keyword enables you to help the compiler by giving it suggestions about which variables should be kept in registers.
 - it is only a hint, not a directive, so ***compiler is free to ignore it!***
 - The behavior is implementation dependent.
- Since a variable declared with `register` might never be assigned a memory address, ***it is illegal to take address of a register variable.***
- A typical case to use `register` is when you use a counter in a loop.

```
int strlen ( register char *p)
{
    register int len=0;
    while(*p++) {
        len++;
    }
    return len;
}
```



Storage classes summary

- **auto**
 - superfluous and rarely used.
- **static**
 - In declarations within a function, static causes variables to have fixed duration. For variables declared outside a function, the static keyword gives the variable file scope.
- **extern**
 - For variables declared within a function, it signifies a global allusion. For declarations outside of a function, extern denotes a global definition.
- **register**
 - It makes the variable automatic but also passes a hint to the compiler to store the variable in a register whenever possible.
- **const**
 - The const specifier guarantees that you can NOT change the value of the variable.
- **volatile**
 - The volatile specifier causes the compiler to turn off certain optimizations. Useful for device registers and other data segments that can change without the compiler's knowledge.



Bu yansı ders notlarının düzeni için boş bırakılmıştır.



Pointers and Arrays

Structural Programming

by Z. Cihan TAYSI

Additions and corrections by Yunus Emre SELÇUK



Outline

- Basics
- Declaration
- How arrays stored in memory
- Initializing arrays
- Accessing array elements through pointers
- Examples
- Strings
- Multi-dimensional arrays



Basics

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    short i,j; //short integers
    short *p; //pointer to short
    i = 123;    //statement #1
    j = 321;    //statement #2
    p = &i;     //statement #3: p now shows the memory address of i
    j = *p;     //statement #4: * means: use the indirect (pointer) value of p
    printf("i:%d j:%d", i, j);
    i += 2; j += 3; printf("i:%d j:%d", i, j); //statement #5
    return 0;
}
What will happen?
```



Basics

Initial state:

Variable name / symbolic name	memory address	memory contents
i	1200	
j	1202	
p	1204	

After statements 1-3:

Variable name / symbolic name	memory address	memory contents
i	1200	123
j	1202	321
p	1204	1200

PS: 1200 is just my assumption. The exact address where these variables will be held will be defined at runtime.



Basics

After statement 4:

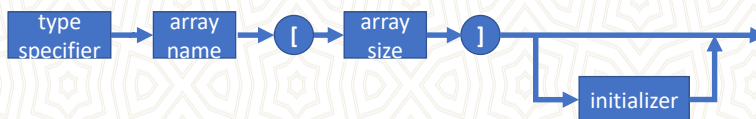
Variable name / symbolic name	memory address	memory contents
i	1200	123
j	1202	123
p	1204	1200

After statement 5:

Variable name / symbolic name	memory address	memory contents
i	1200	125
j	1202	126
p	1204	1200



Declaration



```
int dailyTemp[365];
```

```
dailyTemp[0] = 38;
```

```
dailyTemp[0] = 23;
```

- **subscripts begin at 0, not 1 !**



How Arrays Stored in Memory

```
int ar[5]; /* declaration */  
ar[0] = 15;  
ar[1] = 17;  
ar[3] = ar[0] + ar[1];
```

- **Note that ar[2] and ar[4] have undefined values!**

- the contents of these memory locations are whatever left over from the previous program execution

Element Address

	0x0FFC	
ar[0]	0x1000	15
ar[1]	0x1004	17
ar[2]	0x1008	undefined
ar[3]	0x100C	32
ar[4]	0x1010	undefined
	0x1014	



Initializing Arrays

- It is incorrect to enter more initialization values than the number of elements in the array
- If you enter fewer initialization values than elements, the remaining elements initialized to zero.

- **Note that 3.5 is converted to the integer value 3!**

- When you enter initial values, you may omit the array size
 - the compiler automatically figures out how many elements are in the array...

```
int a_ar[5];  
int b_ar[5] = {1, 2, 3.5, 4, 5};  
int c_ar[5] = {1, 2, 3};
```

```
char d_ar[] = {'a', 'b', 'c', 'd'};
```



Accessing Array Elements Through Pointers

```
short ar[4];  
short *p;
```

```
p = &ar[0]; // assigns the address  
of array element 0 to p.
```

- `p = ar;` **is same as above assignment!**
- `*(p+3)` refers to the same memory content as `ar[3]`

```
float ar[5], *p;
```

```
...
```

```
p = ar ;
```

```
// legal
```

```
ar = p;
```

```
// illegal
```

```
&p = ar;
```

```
// illegal
```

```
ar++;
```

```
// illegal
```

```
ar[1] = *(p+3);
```

```
// legal
```

```
p++;
```

```
// legal
```



examples

- Bubble sort

6 5 3 1 8 7 2 4

- Selection sort

8
5
2
6
9
3
1
4
0
7



Strings

- A string is an array of characters terminated by a null character.
 - null character is a character with a numeric value of 0
 - it is represented in C by the escape sequence '\0'
 - A string constant is any series of characters enclosed in double quotes
 - it has datatype of array of char and each character in the string takes up one byte!
- `char str[] = "some text";`
 - `char str[10] = "yes";`
 - `char str[3] = "four"`
 - `char str[4] = "four"`
 - `char *ptr = "more text";`



String Assignments

```
main () {  
    char array[10];  
    char *ptr1="10 spaces";  
    char *ptr2;  
    array = "not OK";  
    array[5] = 'A';  
    array[0] = 'C';  
    array[1] = '\0';  
    ptr1[8] = 'r';  
    *ptr2 = "not OK";  
    ptr2="OK";  
}
```

// can NOT assign to an address! Does not compile (☹)
// Buggy¹ because: Array is not populated yet. So, ...
// ... Always begin from 0 and
// use null-terminated strings where necessary
// Buggy¹ because: The entire string is not populated yet.
// Type mismatch warning. Does not compile (☹)

¹ in DevCPP4, linker gives warning at first but if you make a second attempt, it compiles!



Strings vs. Chars

Chars

```
char ch = 'a';    // one byte is allocated for 'a'
*p = 'a';         // OK
p = 'a';          // Illegal
```

Strings

```
char *p = "a";    // two bytes allocated for "a"
*p = "a";         // INCORRECT
p = "a";          // OK
```



Reading & Writing Strings

```
#include <stdio.h>
#define MAX_CHAR 80
int main(int argc, char *argv[]) {
    char str[MAX_CHAR];
    printf("Enter a string: ");
    scanf("%s", str);
    printf("\nYou wrote:");
    printf("%s", str);
    return 0;
}
```

- You can read strings with scanf() function.
 - the data argument should be a pointer to an array of characters ***that is long enough to store*** the input string.
 - after reading input characters scanf() automatically appends a null character to make it a proper string
- You can write strings with printf() function.
 - the data argument should be a pointer to a null terminated array of characters



String Length Function

- We test each element of array, one by one, until we reach the null character.
 - it has a value of zero, making the while condition **false**
 - any other value of str[i] makes the while condition **true**
 - once the null character is reached, we exit the while loop and return *i*, which is the last subscript value
- The strlen function is already defined in string.h, therefore the function on the left is named strlen

```
int strlen( char *str ) {  
    int i=0;  
    while( str[i] != '\0' ) {  
        i++;  
    }  
    return i;  
}
```



Other String Functions Defined in string.h

- char* strcpy(char* szCopyTo, const char* szSource)
- char* strncpy(char* szCopyTo, const char* szSource, size_t sizeMaxCopy)
- char* strcat(char* szAddTo, const char* szAdd)
- char* strncat(char* szAddTo, const char* szAdd, size_t sizeMaxAdd)
- int strcmp(const char* sz1, const char* sz2)
- int strncmp(const char* sz1, const char* sz2, size_t sizeMaxCompare)
- etc
- You can look them up in the string.h file and in any C book/site

