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

DERS İÇERİĞİ

- Hatırlatma: C'de veri tipleri, Bitsel işlemler, 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
- Dosya işlemleri
- C Önişlemcileri ve Makrolar
- Statik ve Dinamik Kütüphaneler

KAYNAKLAR

- Darnell P. A. and Margolis P. E., C: A Software Engineering Approach, 3rd ed., Springer-Verlag, 1996.

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 altı**nda 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

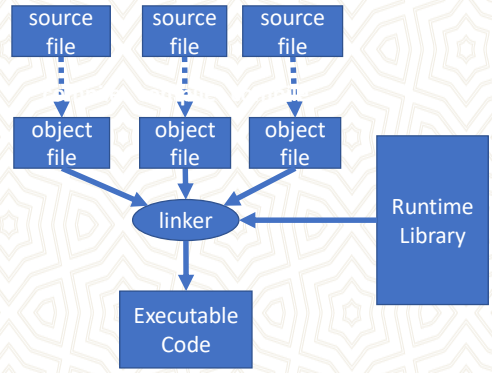
Z. Cihan TAYSI

Outline

- Program development
- C Essentials
  - 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.

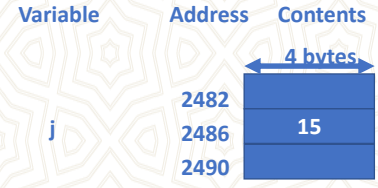


Program Development CONT'D

- 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
  - __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; 

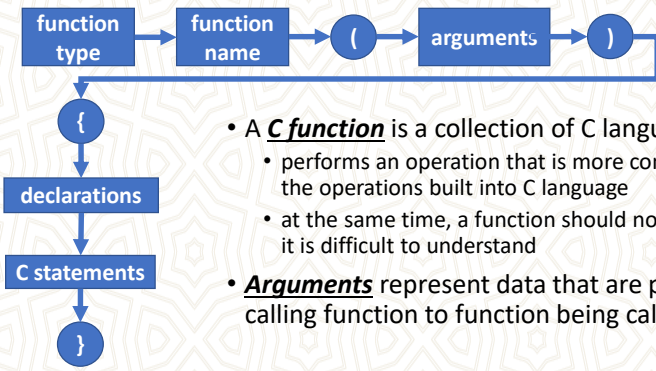
num * num = answer; 

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

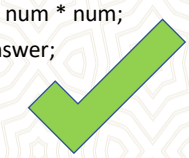
- … or the required function should be completely coded before it is called from another function.

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


Formatting Source Code

```
int square (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;
}
```

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!!!

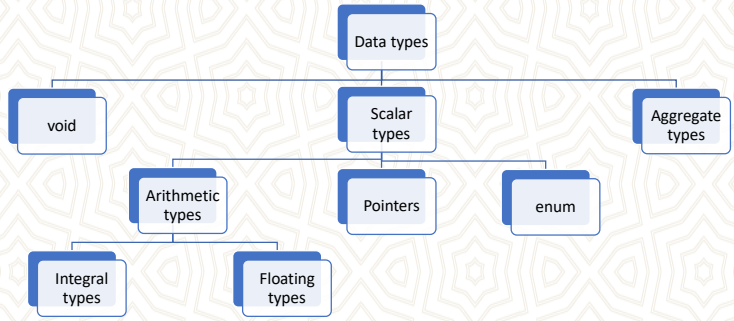
```
#include <stdio.h>

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?

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

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

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 $2^{8}-1$

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

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

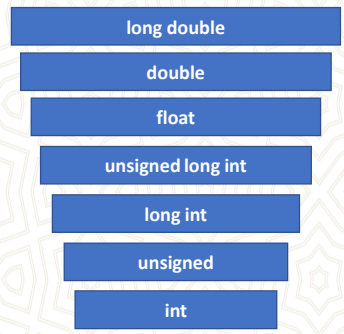
Initialization

- 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 :** – 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

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 = 2.5;                          // 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;
```


Bu yansı ders notlarının düzeni için boş bırakılmıştır.

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 + (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;

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

Increment & Decrement Operators

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

m++ - --j	(m++) – (--j)	(2)
m += ++j * 2	m = (m + ((++j) * 2)	(3)
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 FALSE, 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	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
  - (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+5) 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;
```

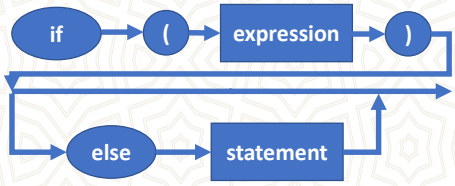
Memory Operators

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:

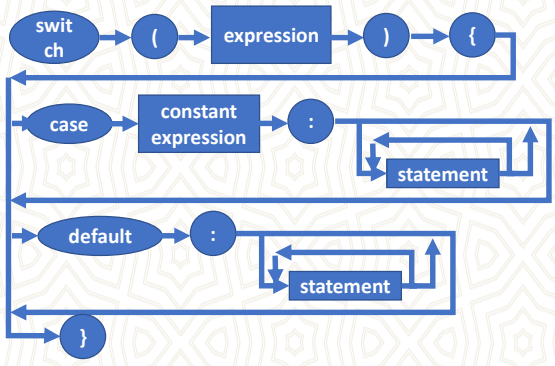
```
if (x)
   statement1;     // executed only if x is nonzero
else
   statement2;     // executed only if x is zero
statement3;        //always executed
```

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.

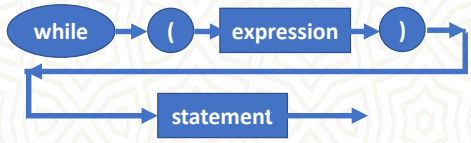
```
if(a<b)
   if(a<c)
         return a;
   else
         return c;
else if (b<c)
         return b;
else
   return c;
```

The switch Statement



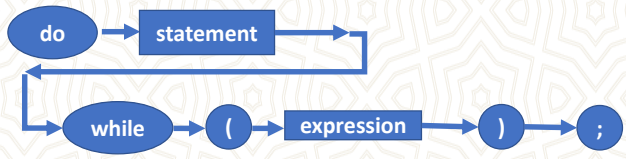
- The ***switch*** expression is evaluated,
  - if it matches one of the case labels, program flow continues with the statement that follows the matching case label.
  - If none of the case labels match the switch expression, program flow continues at the default label, **if exists!**
- No two case labels 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



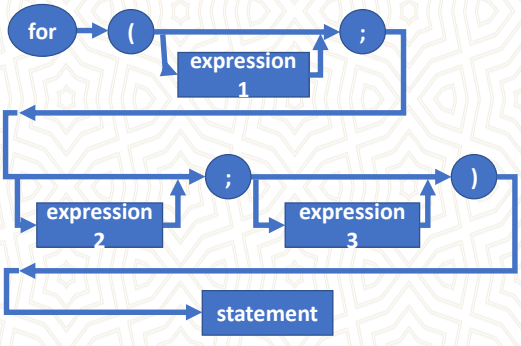
- First the expression is evaluated. If it is a ***nonzero*** value, statement is executed.
- After statement is executed, program control returns to the top of the while statement, and the process is repeated.
- This continues indefinitely until the expression evaluated to zero.

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.

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

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 expilicitly 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",
                        __DATE__,
__TIME__);

}
```

```
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

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
```


Conditional Compilation – III

#if defined TEST	#if defined (TEST)
#if defined *macro_name*	#ifdef *macro_name*
#if !defined macro_name	#ifndef *macro_name*

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 extention
  - contain data structure definitions, macro definitions, function prototypes and global data

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
}
```

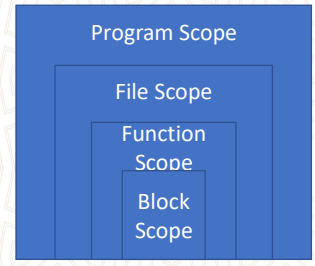
- When increment() is called the second time,
  - memory for ***j*** is reallocated and ***j*** is reinitialized to 1.
  - k has still maintained its memory address and is **NOT** reinitialized.
- Fixed variables get a default initial value of **zero**.

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

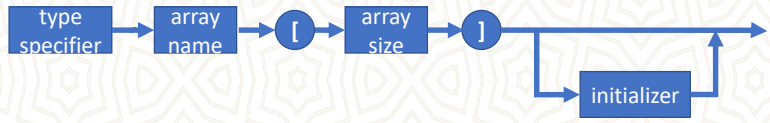
After statement 4:

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



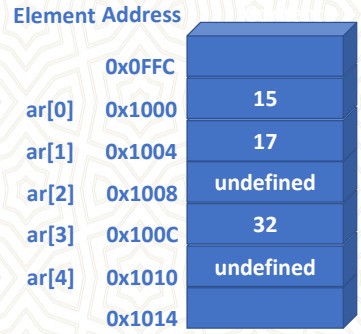
int dailyTemp[365];
dailyTemp[0] = 38;
dailyTemp[0] = 23;

- **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



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";          // can NOT assign to an address! Does not compile (☺)
    array[5] = 'A';            // Buggy[1] because: Array is not populated yet. So, …
    array[0] = 'C';            // … Always begin from 0 and
    array[1] = '\0';           // use null-terminated strings where necessary
    ptr1[8] = 'r';             // Buggy[1] because: The entire string is not populated yet.
    *ptr2 = "not OK";          // Type mismatch warning. Does not compile (☺)
    ptr2="OK";
}
```

[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

Pattern Matching Example

- Write a program that
  - gets two strings from the user
  - search the first string for an occurrence of the second string
  - if it is successful
    - return byte position of the occurence
  - otherwise
    - return -1
- Use pointer operations

Pattern Matching Example, Answer 1:

```
int indexOfV1( char *ptr1, char *ptr2) {
    int i, matchCount = 0;
    int len1 = strlen(ptr1), len2 = strlen(ptr2);
    for( i=0; i<=len1-len2; i++ ) {
        while( *ptr1 == *ptr2 && matchCount != len2 ) {
              matchCount++; ptr1++; ptr2++;
        }
        if( matchCount == len2 ) return i;
        else {
              ptr1 -= (matchCount-1);
              ptr2 -= matchCount; matchCount = 0;
        }
    }
    return -1;
}
```

Pattern Matching Example, Answer 2:

```
int indexOfV2( char *ptr1, char *ptr2) {
    char *ptr;
    ptr = strstr(ptr1, ptr2);
    if( ptr != NULL )   return ptr-ptr1;
    else                return -1;
}
```

- char* strstr (const char* szSearch, const char *szFor);
  - Notice that this function of string.h returns:
    - either a valid pointer to the beginning of the first occurrence of *szFor in *szSearch
    - or a null pointer

Pattern Matching Example, main function:

```
int main () {
    char str1[MAX_CHAR], str2[MAX_CHAR];
    int pos;
    printf("Enter the 1st string (Max. %d characters): ", MAX_CHAR);
    scanf("%s",str1);
    printf("Enter the 2nd string (Max. %d characters): ", MAX_CHAR);
    scanf("%s",str2);
    printf("Found at: %d", indexOfV1(str1,str2));
    return(0);
}
```

Multi-Dimensional Arrays

- In the following, ar is a 3-element array of 5-element arrays

  int ar[3][5];

  - the array reference — ar[1][2]
  - is interpreted as — *(ar[1]+2)
  - which is further expanded to — *(*(ar+1)+2)

- In the following, x is a 3-element array of 4-elemet arrays of 5-element arrays

  char x[3][4][5];

Initialization of Multi-Dimensional Arrays

```
int exap[5][3] = { {  1, 2, 3 },
                   { 4 },
                   { 5, 6, 7 } };
```

1	2	3
4	0	0
5	6	7
0	0	0
0	0	0

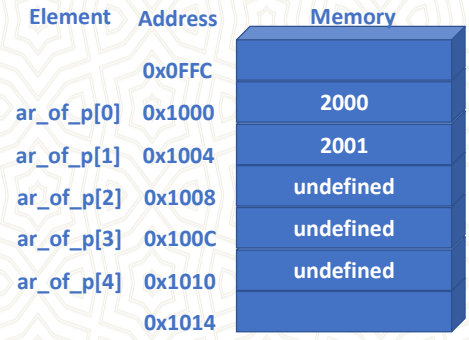
```
int exap[5][3] = {  1, 2, 3,
                    4,
                    5, 6, 7 };
```

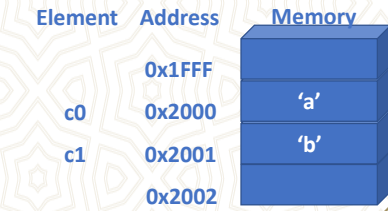
1	2	3
4	5	6
7	0	0
0	0	0
0	0	0

array of pointers

```
char *ar_of_p[5];
char c0 = 'a';
char c1 = 'b';

ar_of_p[0] = &c0;
ar_of_p[1] = &c1;
```





Pointers to Pointers

int r = 5;	declares ***r*** to be an int
int *q = &r;	declares ***q*** to be a pointer to an int
int **p = &q;	declares ***p*** to be a pointer to a pointer to an int
r = 10;	Direct assignment
*q = 10;	Assignment with one indirection
**p = 10;	Assignment with two indirections

Dynamic Memory Allocation

Structural Programming

By Z. Cihan TAYSI

Outline

- Memory allocation functions
- Array allocation
- Matrix allocation
- Examples

Memory Allocation Functions

- ***void *malloc(total_size_in_bytes)***
  - Allocates a specified number of bytes in memory. Returns a pointer to the beginning of the allocated block.
- ***void *calloc(number_of_elements, element_size)***
  - Similar to malloc(), but initializes the allocated bytes to zero.
  - calloc has 2 parameters while malloc has one but the resulting allocated free space will be the same (total size = n * element size).
- ***void *realloc()***
  - Changes the size of a previously allocated block.
  - When extending the size of a dynamically extended block, never assume that the additional are will be cleared.
  - However, contents of previously allocated memory will remain intact.
- ***void free(void *ptr)***
  - Frees up memory that was previously allocated with *malloc(), calloc(), or realloc().*

Array Allocation

```
int n;
int *list;
...
printf("How many numbers are you going to enter ?");
scanf("%d", &n);
list = (int *) malloc( n * sizeof(int) ); //OR: (int *) calloc( n, sizeof(int) );
if(list==NULL) {
        printf("%s:%d>Can not allocate memory for the array...\n",__FILE__, __LINE__);
        return -1;
}
//use the memory and then
free(list)
```

Matrice Allocation

```
int **mat;
int n,m;
printf("Please enter number of rows");scanf("%d", &n);
printf("Please enter number of columns");scanf("%d", &m);
mat = (int **) malloc( n * sizeof(int *) );
if(mat == NULL) {
        printf("%s:%d>Can not allocate memory for the array...\n",__FILE__, __LINE__);
        return -1;
}
for(i = 0; i < n; i++) {
        mat[i] = (int *)malloc(m * sizeof(int) );
}
//will be continued in the next slide
```

Matrice Allocation (cont'd)

```
//use the memory and then
for(i = 0; i < n; i++) {
        mat[i] = (int *)malloc(m * sizeof(int) );
}
free(mat);
```

- to avoid memory leaks, the general rule is this: for each malloc(), there must be exactly one corresponding free()

Example 1, 2

- Write a simple program
  - ask number of elements in the array
  - allocate necessary space
  - ask for elements
  - sort the array
- Write a program
  - ask dimensions of the matrices
  - check if it is possible to multiple them !
  - allocate necessary space
  - ask for elements
  - perform multiplication
  - write the result matrice
- To do: Left as an exercise to code at home

Example 3

- String matrix operations

Functions

Structural Programming

by Z. Cihan TAYSI

Corrections and additions by Yunus Emre SELÇUK

Outline

- Passing arguments
  - pass by reference, pass by value
- Declarations and calls
  - definition, allusion, function call
- The main function

Passing Arguments

- Because C passes arguments by value, a function can assign values to the formal arguments without affecting the actual arguments
- If you do want a function to change the value of an object, you must pass a pointer to the object and then make an assignment through the dereferenced pointer.
  - ***remember the scanf function***
  - ***also remember how we have coded the indexOf function***

Passing Arguments: Demonstration

```
#include <stdio.h>
void increaseRegular( int aa, int bb ) {
    aa += bb;
    printf("increaseRegular finishes with %d\n", aa);
}
void increasePointer( int *aa, int bb ) {
    *aa += bb;
    printf("increasePointer finishes with %d\n", *aa);
}
int main() {
    int a=3, b=5;
    increaseRegular( a, b );
    printf("main says the value is %d\n", a);
    increasePointer( &a, b );
    printf("main says the value is %d\n", a);
    return(0);
}
```

- Please run the code and check the output.

Declarations and Calls

- Definition
  - Actually defines what the function does, as well as number and type of arguments
- Function Call
  - Invokes a function, causing program execution to jump to the next invoked function. When the function returns, execution resumes at the point just after the call

Function Allusion Examples

- Function Allusion
  - Declares a function that is defined somewhere else
  - We will study how to create a project that contains multiple source files later. This topic will be demonstrated then.

```
void simpleFunction1( void );    // prototype of last example
simpleFunction1();

extern float simpleFunction2();
int factorial( int );
void sortArray(int *, int);
float *mergeSort(float *, int, float *, int, int *);
```


Function Definition

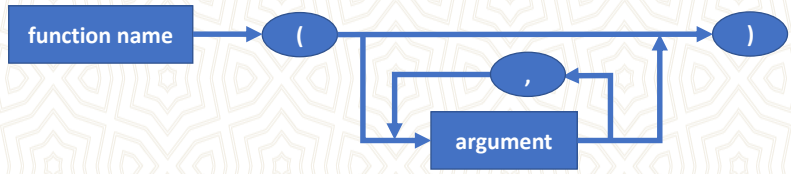
- A very simple example
  - no arguments
  - no return

```
void simpleFunction1 ( void )  {
    printf("\nThis is simpleFunction1\n");
}
```

- A relatively complex example
  - a function to calculate factorial n

```
int factorial( int n)  {
    int i,f=1;
    for(i=2;i<=n;i++)
        f = f * i;
    return f;
}
```


Function Call




```
printf("%d : %d  : %s : %d\n", i, j, line, rc);
matT = transpose(mat, rows, cols);
```

```
printf("Hello World\n");
printf("Result is %d\n", factorial(10));
scanf("%s", str);
x = factorial(n) / factorial(m);
```


Order of Functions

- In order to use a function you must define it beforehand.
  - In order to use your own function in the ***main() function***, you should define it ***before the main()*** in the same file
- It is also possible to use function allusion (function prototype)
  - You can write the prototype of your function before the ***main() function*** and use it anywhere (main() or any other function of yours)

Example 1

- Write a simple function that returns the factorial of a given number.
- Remember factorial
  - 1! = 1
  - 2! = 2 x 1! = 2
  - 3! = 3 x 2! = 6
  - 4! = 4 x 3! = 24
  - and so on…
- To do: Left as an exercise to code at home

Example 2

- Write a simple function that controls if the given char variable is alphabetic ?
- Must check
  - a – z
  - A– Z
- Returns
  - 1, if it is a alphabetic
  - 0, if not
- To do: Left as an exercise to code at home

Example 3

- Write a function to swap values of two integer parameters.
- function takes two integers (a, b)
- When function returns, we must have the value of a in b, and value of b in a.
- Remember
  - tmp = a;
  - a = b;
  - b = tmp;
- To do: Left as an exercise to code at home

Passing Arrays as Function Parameter

- Several ways to do it…
- Do NOT forget
  - No boundary checking !
  - remember your motivation to create a function
- Using actual array size
  - void printArray(int ar[5])
  - Not very convenient, what if you need to print arrays of multiple sizes?
- Using array and a size parameter
  - void printArray(int ar[], int size)
  - This is more convenient than the previous method
- Using a pointer and an integer
  - void myFunction(int *ar, int size)
  - This is also convenient.

Passing Arrays as Function Parameter

- A hint for obtaining the size of any type of array
  - Define a macro to obtain the size of any type of array such as the one below.
- However, this does not eliminates the necessity of passing array size as an extra parameter to a function
  - An array sent as a parameter to a function is treated as a pointer, so sizeof will return the pointer's size, instead of the array's.
  - Thus, inside functions this macro does not work.
  - You will probably ask the user how many elements that s/he will enter or you should keep a counter if you obtain array elements in a while loop.

```
#define SIZE_OF_ARRAY(x) (sizeof(x) / sizeof((x)[0]))
```


Example

- Create a sort function for one dimensional arrays
- Use any type of sorting algorithm

- To do: Left as an exercise to code at home

How to Return an Array from a Function

- We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned.
- We must, make sure that the array exists after the function ends!
  - you can **NOT** return local arrays!
- **SOLUTION** : dynamic memory allocation

Example (concatenation)

- Write a function that takes two arrays and returns the concatenation of them.

```
int* concatArraysV1( int arr1[], int size1, int arr2[], int size2 ) {
    int *merged = (int*) malloc( (size1+size2)*sizeof(int) );
    int i;
    for( i=0; i<size1; i++ )
        merged[i] = arr1[i];
    for( ; i<size1+size2; i++ )
        merged[i] = arr2[i-size1];
    return merged;
}
```


Example (concatenation) (cont'd.)

- Before the function:

```
#include <stdio.h>
#define SIZE_OF_ARRAY(x) (sizeof(x) / sizeof((x)[0]))

void printArray( int a1[], int size ) {
    int i;
    for( i=0; i<size; i++ )
        printf("%d\t", a1[i]);
    printf("\n");
}
```


Example (concatenation) (cont'd.)

- After the function:

```
int main() {
    int arr1[] = {1,5,7,19}, arr2[] = {2,6,8,11,28};
    int *ptrM = concatArraysV1(
        arr1, SIZE_OF_ARRAY(arr1), arr2, SIZE_OF_ARRAY(arr2));
    printf("Array 1 is:\t");
    printArray(arr1, SIZE_OF_ARRAY(arr1));
    printf("Array 2 is:\t");
    printArray(arr2, SIZE_OF_ARRAY(arr2));
    printf("Array 3 is:\t");
    printArray(ptrM, SIZE_OF_ARRAY(arr1)+SIZE_OF_ARRAY(arr2));
    return(0);
}
```


Example (concatenation) (cont'd.)

- Highlights:

```
int* concatArraysV1( int arr1[], int size1, int arr2[], int size2 ) {
    int *merged = (int*) malloc( (size1+size2)*sizeof(int) );
    int i;
    for( i=0; i<size1; i++ )          merged[i] = arr1[i];
    for( ; i<size1+size2; i++ )       merged[i] = arr2[i-size1];
    return merged;
}
int main() {
    int arr1[] = {1,5,7,19}, arr2[] = {2,6,8,11,28};
    int *ptrM = concatArraysV1(
        arr1, SIZE_OF_ARRAY(arr1), arr2, SIZE_OF_ARRAY(arr2));
    ...
```


Alternative to Returning an Array from a Function

- Instead of having the function to allocate memory and return a pointer to the result, you can have the caller of the function to define a blank array and pass this to the function for populating.

Example (concatenation)(alternative)

```
#include <stdio.h>
void printArray( int a1[], int size ) {
    int i;
    for( i=0; i<size; i++ )
        printf("%d\t", a1[i]);
    printf("\n");
}
void concatArraysV2( int arr1[], int size1, int arr2[],
        int size2, int arr3[] ) {
    int size3 = size1+size2;
    int i;
    for( i=0; i<size1; i++ )
        arr3[i] = arr1[i];
    for( ; i<size1+size2; i++ )
        arr3[i] = arr2[i-size1];
}
```


Example (concatenation)(alternative)(cont'd.)

```
int main() {
    int arr1[] = {1,5,7,19}, arr2[] = {2,6,8,11,28};
    int size1 = sizeof(arr1)/sizeof(arr1[0]);
    int size2 = sizeof(arr2)/sizeof(arr2[0]);
    int size3 = size1+size2;
    int arr3[size3];
    concatArraysV2( arr1, size1, arr2, size2, arr3);
    printf("Array 1 is:\t"); printArray(arr1,size1);
    printf("Array 2 is:\t"); printArray(arr2,size2);
    printf("Array 3 is:\t"); printArray(arr3,size3);
    return(0);
}
```

- By the way, I have removed the macro definiton `SIZE_OF_ARRAY`. You decide whether it is worthy or not.

Example

- Write a function that takes two ordered array and returns the ordered union of them.
- To do: Left as an exercise to code at home

Example

- Write a function that return compresses a sparse matrix
- The function should take a matrix as a parameter
- The function should return a new matrix 3 x n or n x 3

- To do: Left as an exercise to code at home

main() Function arguments

- All C programs must contain a function called ***main()***, which is always the first function executed in a C program.
- When ***main()*** returns, the program is done.
- The compiler treats the main() function like any other function, except that at runtime the host environment is responsible for providing two arguments
  - **argc** – number of arguments that are presented at the command line
  - **argv** – an array of pointers to the command line arguments

```
main(int argc, char *argv[])  {

   while(--argc > 0 )
         printf("%s\n",  *++argv);
   exit(0);
}
```


main() Function

- A better way to handle command line arguments
  - getopt
  - argp
  - suboptions

```
while ((c = getopt (argc, argv, "ac:")) != -1)
{
    switch (c)
    {
      case 'a':
        aflag = 1;
        break;
      ....
      default:
        abort ();
    }
}
```


Bu yansı ders notlarının düzeni için boş bırakılmıştır.

Structures, Unions, Enums

Structural Programming

by Z. Cihan TAYSI

Corrections and additions

by Yunus Emre SELÇUK, Zeyneb YAVUZ

Outline

- Structure definition
- Nested structures
- Structure arrays
- Passing structures as function parameters
- An example: Linked list implementation
- Union definition
- Passing unions as function parameters

Structure Definition – I

- Arrays are useful for dealing with identically typed variables but managing groups of differently typed data needs a better way.
- For example, to keep the record of an employee, we need to store his/her name as string, surname as string, ID as integer and salary as float.
- If we insist on using arrays, we need to use multiple 1-D arrays
- Moreover, assume that we need to track 1000 employees

```
char names[1000][20], surnames[1000][20];
int  IDs[1000]; float salaries[1000];
```


Structure Definition – II

- A more natural organization would be to create a single variable that contains all four pieces of data for one employee. C enables you to do this with a data type called a structure.
  - Defining a structure type that can keep the information of an employee:

```
struct Employee {
    char name[20], surname [20];
    int  ID;
    float salary;
};
```

  - Creating an array of employees:

```
struct Employee employees[1000];
```


Structure Definition – III

- A more convenient way to define and use a structure:

```
typedef struct {
    char name[20], surname [20];
    int  ID;
    float salary;
} EMPLOYEE ;
```

  - In that case, EMPLOYEE represents the entire structure definition, including the struct keyword.
    - Using capital case is a naming convention to keep such structs from regular variable names.
  - Then the array definition becomes:

```
EMPLOYEE employees[1000];
```


Accessing to the Fields of a Structure

- You can access the fields of structure variable by the dot sign.

```
EMPLOYEE yunus;
yunus.ID = 1234;
```

- You can access the fields of structure pointer by the arrow sign.

```
EMPLOYEE *e_ptr;
e_ptr->ID = 1234;
```

- The arrow notation is a tidier way of writing:

```
(*e_ptr).ID = 1234;
```


Nested Structures

- You can define a structure within another, creating data hierarchies.
  - They can also be used separately, therefore define separately and nest them as needed.
  - Adding the enlisting date of an employee:

```
typedef struct {
    short day, month;
    int year;
} DATE ;
typedef struct {
    char name[20], surname [20];
    int  ID;
    float salary;
    DATE enlisted;
} EMPLOYEE ;
```

- Later, you can write:

```
yunus.enlisted.year = 2008;
```


Passing structures as function parameters

- There are two ways to pass structures as arguments:
  - pass the structure itself (called pass by value)

```
EMPLOYEE emp;
printReport(emp);
```

  - pass a pointer to the structure (called pass by reference)

```
EMPLOYEE emp;
increaseSalary(&emp);
```

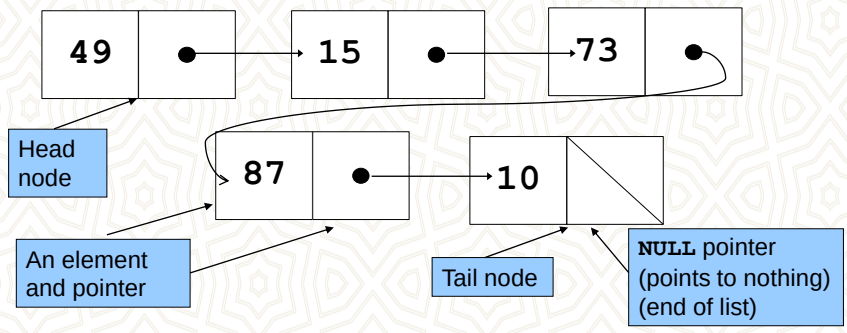
- Passing by reference is faster and more efficient
- Depending on your choice, declare the argument of the function as either a structure or a pointer to a structure
  - Then use `.` or `->` in the body of the function.
- The pointer points to an entire structure, not to its first field.

Structure example: Linked List Implementation

- Array structure is not efficient enough because:
  - They cannot be resized automatically
    - You need to allocate memory for worst-case, which is a waste of memory
  - Insertions are hard
    - You need to shift elements
  - A more efficient data structure is a Linked list:
    - A linked list is a chain of structures that are linked one to another, like sausages.
    - In the simplest linked-list scheme, each structure contains an extra member which is a pointer to the next structure in the list.
  - You will learn about lists and other data structures in the next term in the namesake course

Structure example: Linked List Implementation

- An example linked list holding integers:



Structure example: Linked List Implementation

- We will use the Employee struct as the data element AND the node.
- Advantage: This will keep things a little bit simpler.
- Definition:

```
#include <stdio.h>
typedef struct Employee {
    char name[20], surname [20];
    int  ID;
    float salary;
    struct Employee *next;
} EMPLOYEE;
EMPLOYEE *head;
```


Structure example: Linked List Implementation

- We can also define our node structure as follows
- Advantage: This will keep the struct related with the problem domain separate from the struct related with data representation.
- Left to students as an exercise

```
#include <stdio.h>
typedef struct {
    char name[20], surname [20];
    int  ID;
    float salary;
} EMPLOYEE;
typedef struct emp_node {
    EMPLOYEE data;
    struct emp_node *next;
} EMP_NODE;
EMP_NODE *head;
```


Structure example: Linked List Implementation

- Printing the information of an employee and the entire list:

```
void printElementP( EMPLOYEE *emp ) {
    printf("Employee %d %s has a salary of %f\n",
        emp->ID, emp->name, emp->salary );
}
void printList( ) {
    int j; EMPLOYEE *p;
       for(j=0, p=head; p != NULL; p=p->next, j++)
           printf("%d-th person: %d\t%s\t%f\n",
            j+1, p->ID, p->name, p->salary);
}
```


Structure example: Linked List Implementation

- We will need functions to allocate memory for an employee, creating an employee, …

```
EMPLOYEE* create_list_element(  ) {
    EMPLOYEE *emp; int i; float s;
    emp = (EMPLOYEE*) malloc( sizeof( EMPLOYEE ) );
    if( emp == NULL ) {
        printf("create_employee: out of memory."); exit(1);
    }
    printf("Enter name of the person: "); scanf("%s", emp->name);
    //cannot get non-pointer struct fields directly!
    printf("Enter ID of the person: ");
    scanf("%d", &i); emp->ID = i;
    printf("Enter salary of the person: ");
    scanf("%f", &s); emp->salary = s;
    emp->next=NULL; return emp;
}
```


Structure example: Linked List Implementation

- … and adding her/him to the list.

```
/* The create_list_element() function allocates memory,
   but it doesn't link the element to the list.
   For this, we need an additional function, add_element(): */
```

- … code will continue in the next slide

Structure example: Linked List Implementation

```
void add_element(EMPLOYEE *e){
    EMPLOYEE *p;
    // if the 1st element (head) has not been created, create it now:
    if(head == NULL){ head=e; return; }
    // otherwise, find the last element in the list:

    //Span through each element testing to see whether p.next is NULL.
    //If not NULL, p.next must point to another element.
    //If NULL, we have found the end of the list and we end the loop.
    for (p=head; p->next != NULL; p=p->next); // null statement

    // append a new structure to the end of the list
    p->next=e;
}
```


Structure example: Linked List Implementation

- We may need to fire an employee (deleting a node):

```
/* To delete an element in a linked list,
   you need to find the element before the one you are deleting
   so that you can bond the list back together after removing one of the links.
   You also need to use the free() func,
   to free up the memory used by the deleted element.  */
void delete_element(EMPLOYEE *goner){
    EMPLOYEE *p;
    if(goner == head)
        head=goner->next;
    else // find element preceding the one to be deleted:
        for(p=head; (p!=NULL) && (p->next != goner); p=p->next);
        if(p == NULL){
            printf("delete_element(): could not find the element \n"); return;
    }
    p->next=p->next->next; free(goner);
}
```


Structure example: Linked List Implementation

- We may need to search an employee:

```
/* Finding an Element in the Linked List:
   There is no easy way to create a general-purpose find() function
   because you usually search for an element based on one of its data fields
   (e.g. person's name), which depends on the structure being used.
   To write a general-purpose find() function, you can use function pointers
   (will be studied later).
   The following function, based on the personalstat structure,
   searches for an element, whose name field matches with the given argument.*/
EMPLOYEE *find(char *name) {
    EMPLOYEE *p;
    for(p=head; p!= NULL; p=p->next)
        if(strcmp(p->name, name) == 0) // returns 0, if 2 strings are same
            return p;
    return NULL;
}
```


Structure example: Linked List Implementation

- We may need to put a new employee between two existing people (inserting a node in between:)

```
/* To insert an element in a linked list, you must specify
   where you want the new element inserted.
   The following function accepts 2 pointer arguments, p and q,
   and inserts the structure pointed by p,
   just after the structure pointed by q. */
void insert_after(EMPLOYEE *p, EMPLOYEE *q){
    // if p and q are same or NULL, or if p already follows q, report that:
    if(p==NULL || q==NULL || p==q || q->next == p){
            printf("insert_after(): Bad arguments \n");
            return;
    }
    p->next = q->next;
    q->next = p;
}
```


Structure example: Linked List Implementation

- Let's put them all together and make a demonstration:

```
int main(){
        EMPLOYEE *p,*q;
        int val, j;
        for(j=0; j<2; j++)
                add_element( create_list_element());

        for(j=0, p=head; p != NULL; p=p->next, j++)
        //for(p=head; p != NULL; p=p->next)
        {
                printf("%d-th person: ",(j+1)); printElementP(p);
        }
```


Structure example: Linked List Implementation

- Demonstration cont'd:

```
    // CREATE A NEW ELEMENT AND INSERT IT IN
    // BETWEEN THE 1st AND 2nd ELEMENTS IN THE LIST:
    p=create_list_element();

    q=head; //to keep the first element, head
    insert_after(p, q); //and we insert p, after q:

    printList( );

    return 0;
}
```


Structure Alignment

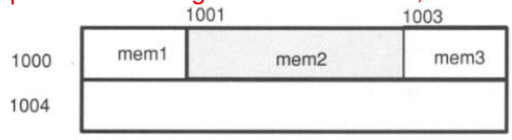
- Some computers require that any data object larger than a char must be assigned an address that is a multiple of a power of 2 (all objects > than a char be stored at even addresses).
- Normally, these alignment restrictions are invisible to the programmer. However, they can create holes, or gaps, in structures.
- Consider how a compiler would allocate memory for the following structure:

```
structure ALIGN_EXAMP{
  char mem1;
  short mem2;
  char mem3;
} s1;
```

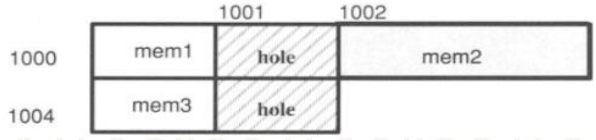

Structure Alignment

- structure ALIGN_EXAMP{ char mem1; short mem2; char mem3; } s1;

If the computer has no alignment restrictions, s1 would be stored as:



If the computer requires objects > a char to be stored at even addresses, s1 would be stored as:



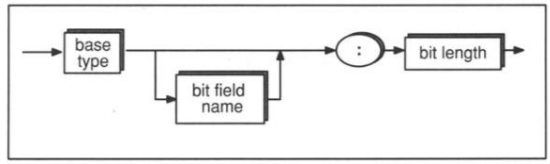
*This storage arrangement results in a I-byte hole between mem1 and mem2 and following mem3.

Bit fields

- The smallest data type that C supports is char(8 bits)
- But in structures, it is possible to declare a smaller object called a *bitfield*.
- Bit fields behave like other int variables, except that:
  - You cannot take the address of a bit field and
  - You cannot declare an array of bit fields.

Bit fields

- Syntax:



- The base type may be **int**, **unsigned int**, or **signed int**.
- If the bit field is declared as int, the implementation is free to decide whether it is an unsigned int or a signed int (**For portable code, use the signed or unsigned qualifier**).
- The *bit length* is an integer constant expression that may not exceed the length of an int.
- On machines where ints are 16 bits long, e.g. the following is illegal:
  int too_long: 17;

Bit fields

- ZK paylaşımını ekle: Hafta5 yansı 12-13

Example: A Struct having a bit field

- Let's keep employment dates of employees:

```
struct personalstat {
    char ps_name[20], ps_tcno[11];
    unsigned int ps_birth_day : 5;
    unsigned int ps_birth_month : 4;
    unsigned int ps_birth_year : 11;
    // pointer to the next element in the linked list:
    struct personalstat *next;
};
// ELEMENT becomes synonymous with struct personalstat:
typedef struct personalstat ELEMENT;

// Always keep a pointer to the beginning of the linked list
static ELEMENT *head;
```


Example: A Struct having a bit field

- The rest will be very similar to our previous example:

```
void printElementP( ELEMENT *emp ) {
    printf("Employee %s %s born in %d.%d.%d\n", emp->ps_tcno, emp->ps_name,
        emp->ps_birth_day, emp->ps_birth_month, emp->ps_birth_year  );
}
void printList( ) {
    int j; ELEMENT *p;
            for(j=0, p=head; p != NULL; p=p->next, j++)
                printf("%d-th person: %s\t%s\t%u.%u.%u\n", j+1, p->ps_name, p->ps_tcno,
                        p->ps_birth_day, p->ps_birth_month, p->ps_birth_year);
}
ELEMENT *create_list_element(){
    ELEMENT *p;
    int val; //ilkel ve bitfield olduğu için geçici değişken şart
    p=(ELEMENT*) malloc (sizeof (ELEMENT));
    if(p == NULL) { printf("create_list_element (): malloc failed. \n"); exit(1); }
    printf("Enter name of the person:"); scanf("%s", p->ps_name);
    printf("Enter tcno of the person:"); scanf("%s", p->ps_tcno);
    printf("Enter the birth-date (day) of the person:"); scanf("%u", &val); p->ps_birth_day=val;
    printf("Enter the birth-date (month) of the person:"); scanf("%u", &val); p->ps_birth_month=val;
    printf("Enter the birth-date (year) of the person:"); scanf("%u", &val); p->ps_birth_year=val;
    p->next=NULL; return p;
}
void add_element(ELEMENT *e){
    ELEMENT *p;
    if(head==NULL){ head=e; return; }
    for (p=head; p->next != NULL; p=p->next); // null statement
    p->next=e;
}
```


Example: A Struct having a bit field

- The rest will be very similar to our previous example:

```
void insert_after(ELEMENT *p, ELEMENT *q){
    // if p and q are same or NULL, or if p already follows q, report that:
    if(p==NULL || q==NULL || p==q || q->next == p){
                printf("insert_after(): Bad arguments \n");
                return;
    }
    p->next = q->next;
    q->next = p;
}
void delete_element(ELEMENT *goner){
    ELEMENT *p;
    if(goner == head)
                head=goner->next;
    else // find element preceding the one to be deleted:
                for(p=head; (p!=NULL) && (p->next != goner); p=p->next); // null statement

    if(p == NULL){
                printf("delete_element(): could not find the element \n"); return;
    }
    p->next=p->next->next;
    free(goner);
}
ELEMENT *find( char * name){
                ELEMENT *p;
    for(p=head; p!= NULL; p=p->next)
                if(strcmp(p->ps_name, name) == 0) // returns 0, if 2 strings are same
                   return p;
    return NULL;
}
```


Example: A Struct having a bit field

- The rest will be very similar to our previous example:

```
int main(){
    ELEMENT *p,*q;
    int val, j;
    for(j=0; j<2; j++)
    add_element( create_list_element());

    for(j=0, p=head; p != NULL; p=p->next, j++) //for(p=head; p != NULL; p=p->next)
        //printf("%d-th person: %s\t%s\t%u.%u.%u\n", j+1, p->ps_name, p->ps_tcno, p->ps_birth_day, p->ps_birth_month, p->ps_birth_year);
        printf("%d-th person: ",(j+1)); printElementP(p);
    }

    // CREATE A NEW ELEMENT AND INSERT IT IN BETWEEN THE 1st AND 2nd ELEMENTS IN THE LIST:
    p=create_list_element();

    q=head; // to keep the first element, head and we'll insert p, after q:
    insert_after(p, q);

    printList( );

    return 0;
}
```


Unions

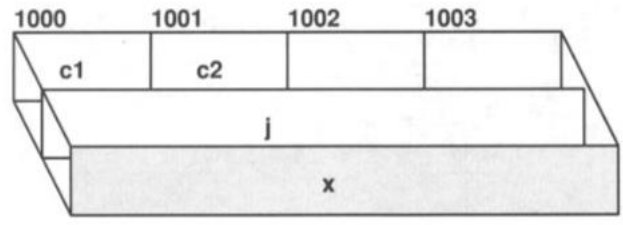
- Unions are similar to structures except that the members are overlaid one on top of another, so members share the same memory.
- There are two basic applications for unions:
  - Interpreting the same memory in different ways.
  - Creating flexible structures that can hold different types of data.

Unions

- Example:

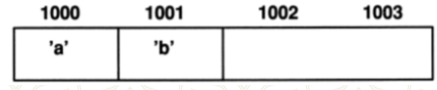
```
typedef union
{
  struct
  {
    char c1, c2;
  } s;
  long j;
  float x;
} U;

U example;
```



- Usage:

```
example.s.c1 = 'a';
example.s.c2 = 'b';
```



- If you make the assignment:

example.j = 5; //overwrites the 2 chars, using all 4 bytes to store value 5.

Yıldız Teknik Üniversitesi - Bilgisayar Mühendisliği Bölümü

Real life example for Unions in Structures

- Consider our PERSONALSTAT example (name, tcno, birth_date), we want to add additional information as follows:
  - **Are you T.C. citizen?**
  - If you are a T.C. citizen, in which city were you born?
  - If not a T.C. citizen, what is your nationality?

```
typedef struct {
  unsigned int day : 5;
  unsigned int month : 3;
  unsigned int year : 11;
} DATE;
```

- This definition wastes memory in each record for either nationality or city_of_birth.

```
typedef struct {
  char ps_name[20], ps_tcno[11];
  DATE ps_birth_date;
  // Bit field for TC citizenship:
  unsigned int TCcitizen : 1;
  char nationality[20];
  char city_of_birth[20];
} PERSONALSTAT;
```


Real life example for Unions in Structures

- Let's construct a better struct with a union so that we eliminate unnecessary waste of memory:

```
typedef struct {
  unsigned int day : 5;
  unsigned int month : 3;
  unsigned int year : 11;
} DATE;
```

```
typedef struct {
  char ps_name[20], ps_tcno[11];
  DATE ps_birth_date;
  unsigned int TCcitizen : 1;
  union{
    char nationality[20];
    char city_of_birth[20];
  } location;
} PERSONALSTAT;
```


Recursion

Structural Programming

by Zeyneb YAVUZ

Corrections and additions

by Yunus Emre SELÇUK

Recursion

- A recursive function is one that calls itself.
  - An example is given on the right side
- It is important to notice that this function will call itself forever.
  - Actually not forever, but till the computer runs out of stack memory
  - It means a runtime error
- Thus, remember to include a stop point in your recursive functions.

```
void recurse () {
        static count = 1;
        printf("%d\n", count);
        count++;
        recurse();
}
main() {
        recurse();
}
```


Recursion

- When a program begins executing in the function main(), space is allocated on the stack for all variables declared within main(), **Figure 14.13(a)**

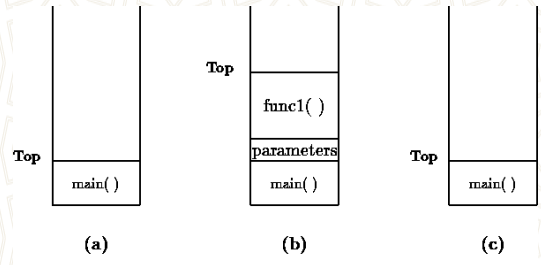
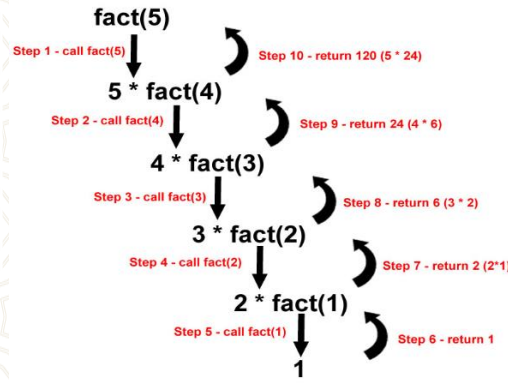


Figure 14.13: Organization of the Stack

- If main() calls a function, func1(), additional storage is allocated for the variables in func1() at the top of the stack **Figure 14.13(b)**
  - Notice that the parameters passed by main() to func1() are also stored on the stack.
- When func1() returns, storage for its local variables is deallocated, and the top of the stack returns to the 1st position **Figure 14.13(c)**
- As can be seen, the memory allocated in the stack area is used and reused during program execution.
  - *It should be clear that memory allocated in this area will contain garbage values left over from previous usage.*

Recursion Example: Factorial Calculation

```
int fact( int n ) {
        if( n <= 1 )
                return 1;
        else
                return n*fact(n-1);
}
main() {
        printf("5! is %d\n", fact(5));
}
```



- A few other examples to solve with recursion (left as exercises at home):
  - Fibonacci numbers – $F_{n+1} = F_n + F_{n-1}$
  - Binary search
  - Depth-first search

Function Pointers

Structural Programming

by Zeyneb YAVUZ

Corrections and additions

by Yunus Emre SELÇUK

More on the Main Function

- It is possible to pass arguments to the main function so that the program begins with initial prior data.
- The compiler treats the main() function like any other function, except that at runtime the host environment is responsible for providing two arguments
  - **argc** – number of arguments that are presented at the command line
  - **argv** – an array of pointers to the command line arguments

```
int main(int argc, char *argv[])  {
    while(--argc > 0 )
          printf("%s\n",  *++argv);
    exit(0);
}
```


More on the Main Function

- getopt: A better way to handle command line arguments
- The getopt() function parses the command-line arguments. Its arguments *argc* and *argv* are the argument count and array as passed to the main() function on program invocation.
- The variable *optind* is the index of the next element to be processed in *argv*. The system initializes this value to 1. If there are no more option characters, getopt() returns -1.
- The variable *optstring* is a string containing all options characters
- Fore more details and an example, please refer to: http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html

Function Pointers

- We can use some functions as arguments to other functions through the function pointers
  - This possibility opens new doors in terms of flexibility for coding.
- Definition:
  - int **(*pf)** (); // pf is a pointer to a function returning an int.
  - The () around *pf are necessary for correct grouping. Because:
  - int *pf(); // this is a function allusion returning an int pointer

Function Pointers

- Assignments to function pointers:

```
extern int f1();
int main( ) {
   int (*pf) (); // pf is a pointer to a function returning an int.
   pf=f1;        // assign the address of f1 to pf
   pf=f1();      // ILLEGAL, f1 returns an int, but pf is a pointer
   pf=&f1();     /* ILLEGAL,  cannot take the address of a function
                    result */
   pf=&f1;       /* ILLEGAL, &f1 is a pointer to a pointer, but pf
                    is a pointer to an int */
}
```


Function Pointers

- Return types:

```
extern int   if1(), if2(), (*pif)();
extern float ff1(), (*pff)();
extern char  cf1(), (*pcf)();

int main( ) {
  pif = if1;  // Legal:   Types match
  pif = cf1;  // ILLEGAL: Type mismatch
  pff = if2;  // ILLEGAL: Type mismatch
  pcf = cf1;  // Legal:   Types match
  if1 = if2;  // ILLEGAL: Assign to a constant
}
```


Function Pointers

- Example function call via a function pointer:

```
#include <stdio.h>
extern int f1(int); //could be defined externally but we have coded it below
int main() {
    int n;
    int (*pf) ();
    int answer;
    printf("Bir sayi giriniz: "); scanf("%d",&n);

    pf=f1;
    answer=(*pf)(n); // calls f1() with argument a => f1(a)

    printf("%d", answer);
    return 0;
}
int f1( int a ) {
    return a+1;
}
```