BLM2031 YAPISAL PROGRAMLAMA – EKİM 2022
Sunan: Dr. Ahmet ELBİR
GENEL BİLGİLER

DERS GRUPLARI

- Gr.1 Dr. Öğretim Üyesi Yunus Emre SELÇUK (YES)
- Gr.2 Dr. Ahmet ELBİR (AEL) (Biz)

İLETİŞİM

- İletişim bilgileri
  - Oda : D-124
  - e-mail: aelbir@yildiz.edu.tr
  - İletişim için öncelikle e-mail gönderiniz, yüz yüze görüşme için randevu isteyiniz.

DERS NOTLARI ve KAYNAKLAR

- https://avesis.yildiz.edu.tr/yselcuk/dokumanlar
  - Önceki katkıları için Z. Cihan Tayşi, H. İrem Türkmen, Zeyneb Kurt hocalarımıza teşekkür ederim.
  - Darnell P. A. and Margolis P. E., C: A Software Engineering Approach, 3rd ed., Springer-Verlag, 1996 (notların oluşturulduğu asıl kaynaktır).

BLM2031 YAPISAL PROGRAMLAMA – GENEL BİLGİLER

BAŞARIM DEĞERLENDİRME

- Uygulama ve lab. çalışmaları:
  - 17/10/2022 itibariyle başlar, dönüşümlü yapılır.
  - Dersi alan tüm öğrenciler lab. çalışmalarına katılmak zorundadır.
- Ara sınav: 22/11/2022 (8.hafta)
- Proje ödevi: Ayrıntılar ileride duyurulacak
- Ara sınav mazereti: 03/01/2023 (14.hafta) (yönetmelik kuralları uyarınca)
- Final sınavı: Final haftasında
- Bölümün sayfasında duyuracağı vize ve final programlarına göre, haftalar ve hatta günler ile saatler değişebilir.
- Puanlama (değişebilir):
  - 1. Ara sınav %25, Lab. %15, Proje %20, Final %40
  - Yapıl(a)mayan değerlendirmenin not ağırlığı yapılanlara paylaştırılır.

BLM2031 YAPISAL PROGRAMLAMA – GENEL BİLGİLER

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

BLM2031 YAPISAL PROGRAMLAMA – GENEL BİLGİLER

ÖNEMLİ SENATO KARARLARI

- 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** (YN-027-YTÜ Önlisans ve Lisans Eğitim-Öğretim Yönetmeliği, Md. 26.e).
- Yarıyıl sonu sınavına girmeyen öğrenciler vize notuna bakılmaksızın ilgili dersten başarısız (FF) sayılırlar (YÖ-075-YTÜ Sınav Yönergesi, Md. 4.2.k).
- Bütün öğrencilere derslere devam zorunluluğu gelmiştir (dersi tekrar alanların önceki notu ne olursa olsun).
  - 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 (YÖ-075-YTÜ Sınav Yönergesi, Md. 4.2.h).

A Fast Review of C Essentials
Part I

Structural Programming

by Z. Cihan TAYSI

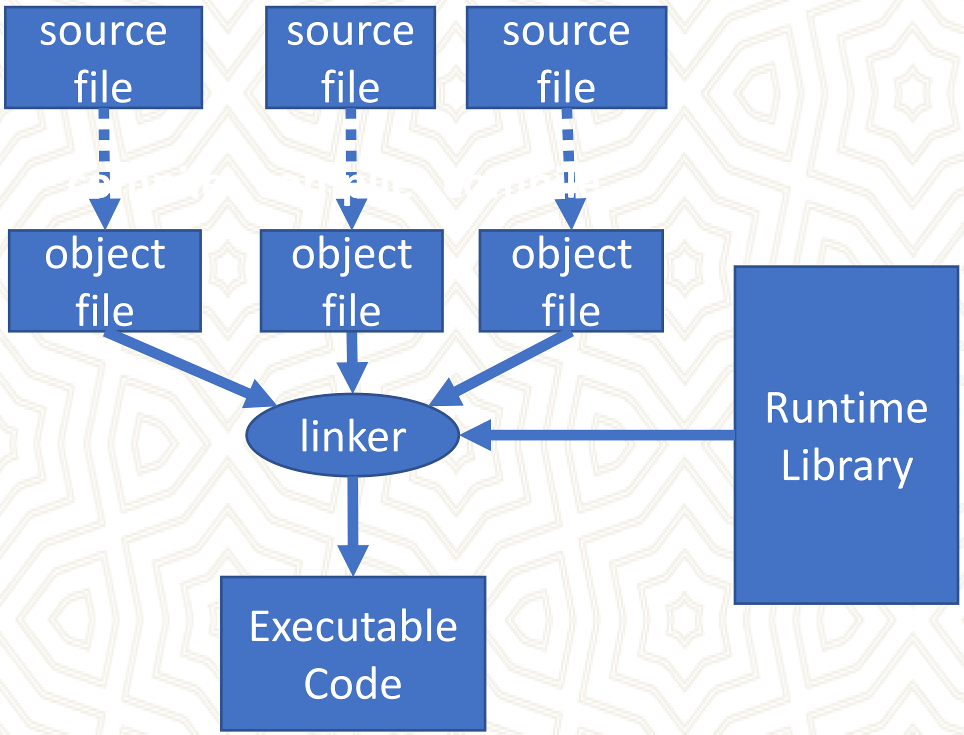
additions by Yunus Emre SELÇUK

Outline

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

Program Development

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

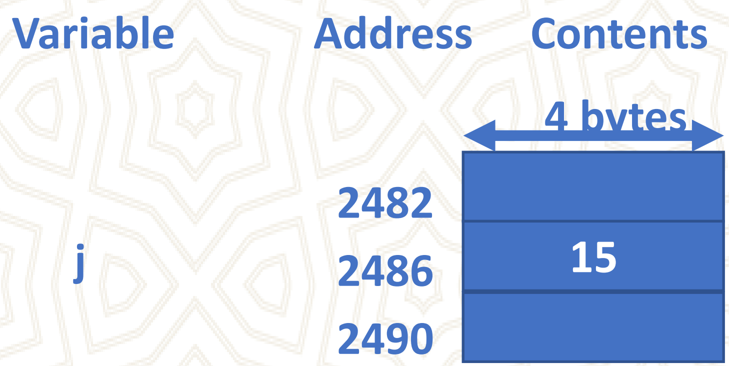


Program Development CONT'D

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

Variables & Constants

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



Names

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

Names cont’d

- **LEGAL NAMES**
  - j
  - j5
  - __sesquipedalial_name_system_name
  - UpPeR_aNd_LoWeR_cAsE_nAmE

- **ILLEGAL NAMES**
  - 5j
  - $name
  - int
  - bad%#*@name

Names cont’d

- reserved keywords = illegal names cont.’d:

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

Expressions

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

Assignment Statements

lvalue → = → rvalue → ;

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

```
answer = num * num;
```

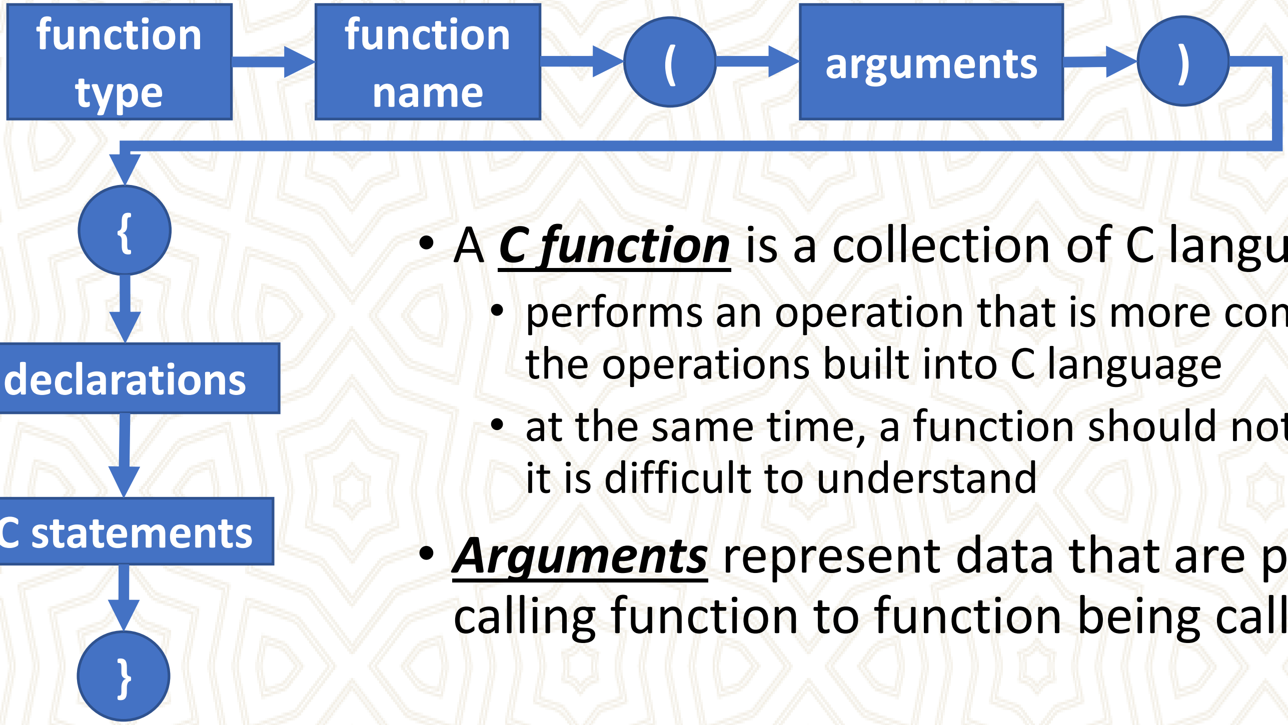
```
num * num = answer;
```

Comments

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

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

Functions



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

Functions

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

Functions

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

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

Functions

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

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

Formatting Source Code

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

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

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

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

The main() Function

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

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

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

printf() and scanf() Functions

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

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

Preprocessor

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

Preprocessor cont'd

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

Preprocessor cont'd

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

hello world!!!

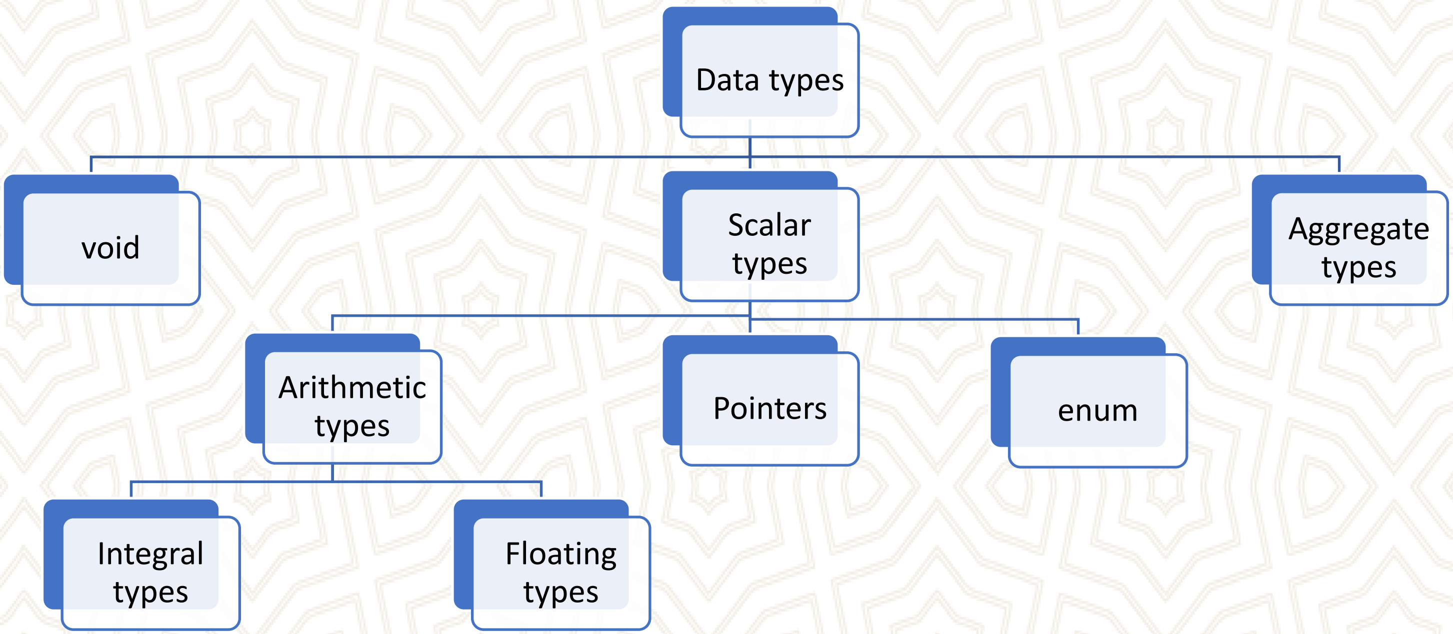
```
#include <stdio.h>

int main ( void )  {

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

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

Data Types



Data types
void
Scalar types
Aggregate types
Arithmetic types
Pointers
enum
Integral types
Floating 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 variables that have the same type in a single declaration
  - int j,k;
- ***<u>All declarations in a block must appear before any executable statements</u>***

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 ***<u>at least 8 bits long</u>***, and that ints must be ***<u>at least 16 bits long</u>*** and must represent the "***<u>natural</u>***" size for computer.
  - natural: the number of bits that the CPU usually handles in a single instruction

Type	Size (in bytes)	Value Range	Format String
int	4	-2^{31} to $2^{31}-1$	%d
unsigned int	4	0 to $2^{32}-1$	%u
short int	2	-2^{15} to $2^{15}-1$	%hi
long int	4	-2^{31} to $2^{31}-1$	%li
unsigned short int	2	0 to $2^{16}-1$	%hu
unsigned long int	4	0 to $2^{32}-1$	%lu
signed char	1	-2^{7} to $2^{7}-1$	%c
unsigned char (rather meaningless)	1	0 to $2^{8}-1$	%uc

Format Strings for Integers

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

```
#include <stdio.h>

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

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

Different Types of Integers cont’d

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

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

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

Floating Point Types

- to declare a variable capable of holding floating-point values
  - ***float (%f)***
  - ***Double (%lf)***
- The word ***double*** stands for double-precision
  - it is capable of representing about twice as much precision as a ***float***
  - A float generally requires ***4 bytes***, and a double generally requires ***8 bytes***
  - ***read more about limits in <limits.h>***

- Decimal point
  - 0.356
  - 5.0
  - 0.000001
  - .7
  - 7.
- ***Scientific notation (%e)***
  - 3e2
  - 5E-5

- Long double can be defined but they can become plain double in some computer platforms
- Refer to the source book and the Internet for different representation format modifiers (such as %5.7f)

Format Strings for Real Numbers

```
#include <stdio.h>

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

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

Initialization

- A declaration allocates memory for a variable, but it does not necessarily store an initial value at the location
  - ***<u>If you read the value of such a variable before making an explicit assignment, the results are unpredictable</u>***
- 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

- long double
- double
- float
- unsigned long int
- long int
- unsigned
- int

Implicit Conversions

- When the compiler encounters an ***expression***, it divides it into ***subexpressions***, where each expression consists of one operator and one or more objects, called ***operands***, that are **bound to the operator**.
- **Ex :** 1 + 2.5 # involves two types, an int and a double
- **Ex :** – 3 / 4 + 2.5 # The expression contains three operators –, /, +
- Each operator has its own rules for operand type agreement, but most binary operators require both operands to have the same type.
  - If the types differ, the compiler converts one of the operands to agree with the other one.
  - For this conversion, compiler resorts to the hierarchy of data types. **(Please remember previous slide)**

Mixing Signed and Unsigned Variables

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

Mixing Integers with Floating Types

- Invisible conversions

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

- **Loss of precision**

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

Explicit Conversions - Cast

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

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

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

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

Enumeration Data Type

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

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

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

void Data Type

- The void data type has two important purposes.

- The first is to indicate that a function does not return a value
  - void func (int a, int b);

- The second is to declare a generic pointer
  - **We will discuss it later !**

typedef

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

```
typedef long int INT32;

long int j;
INT32 j;
```

Bu yansı ders notlarının düzeni için boş bırakılmıştır.

A Fast Review of C Essentials Part II

Structural Programming & Control Flow

by Z. Cihan TAYSI

Outline

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

Expressions

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

Precedence & Associativity

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

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

- a + b – c;
- a = b = c;
- a < b < c

Precedence & Associativity

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

Precedence & Associativity

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

Parenthesis

- The compiler groups operands and operators that appear within the parentheses first, so you can use parentheses to specify a particular grouping order.

  - (2 – 3) * 4
  - 2 – (3 * 4)

- The inner most parentheses are evaluated first. The expression (3+1) and (8-4) are at the same depth, so they can be evaluated in either order.

  1 + ((3+1) / (8 – 4) – 5)

  1 + (4 / (8 – 4) – 5)

  1 + (4 / 4 – 5)

  1 + (1 – 5)

  1 + -4

  -3

Binary Arithmetic Operators

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

The Remainder Operator

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

Arithmetic Assignment Operators

Operator	Symbol	Form	Operation
assign	**=**	**a = b**	put the value of ***b*** into ***a***
add-assign	**+=**	**a += b**	put the value of ***a***+***b*** into ***a***
substract-assign	**-=**	**a -= b**	put the value of ***a***-***b*** into ***a***
multiply-assign	***=**	**a *= b**	put the value of ***a********b*** into ***a***
divide-assign	**/=**	**a /= b**	put the value of ***a***/***b*** into ***a***
remainder-assign	**%=**	**a %= b**	put the value of ***a***%***b*** into ***a***

Arithmetic Assignment Operators

int m = 3, n = 4;

float x = 2.5, y = 1.0;

m += n + x – y	m = (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

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

```
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;
}
```

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

Increment & Decrement Operators

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

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

Comma Operator

- Allows you to evaluate two or more distinct expressions wherever a single expression allowed!

- Ex : for (j = 0, k = 100; k – j > 0; j++, k--)

- Result is the value of the rightmost operand

Relational Operators

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

Relational Operators

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

float x=2.5, y=0.0;

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

Logical Operators

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

Logical Operators

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

float x=2.5, y=0.0;

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

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

Bit Manipulation Operators

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

Bit Manipulation Operators cont’d

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

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

Bit Manipulation Operators cont'd

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

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

Bit Manipulation Operators cont'd

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

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

Bitwise Assignment Operators

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

cast & sizeof Operators

- Cast operator enables you to convert a value to a different type
- One of the use cases of cast is to promote an integer to a floating point number of ensure that the result of a division operation is not truncated.
  - 3 / 2
  - (float) 3 / 2

- The ***sizeof*** operator accepts two types of operands: an expression or a data type
  - **the expression may not have type function or void or be a bit field !**
- ***sizeof*** returns the number of bytes that operand occupies in memory
  - sizeof (3+4) returns the size of int
  - sizeof(short)

Conditional Operator (? :)

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

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

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

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

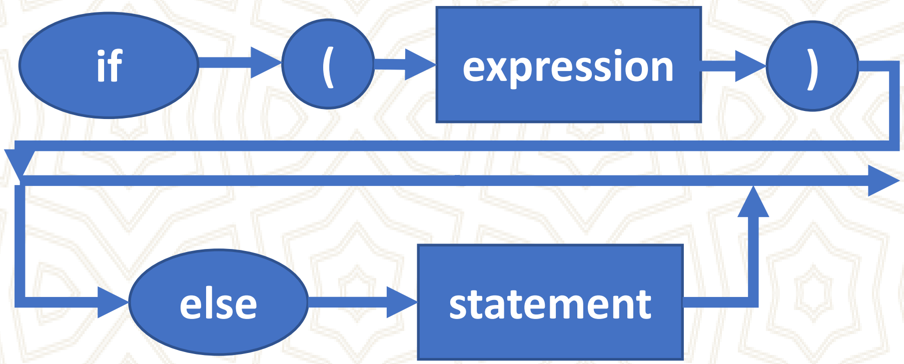
Memory Operators

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

Control Flow

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

The if...else statement



Ex1 :

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

Ex2:

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

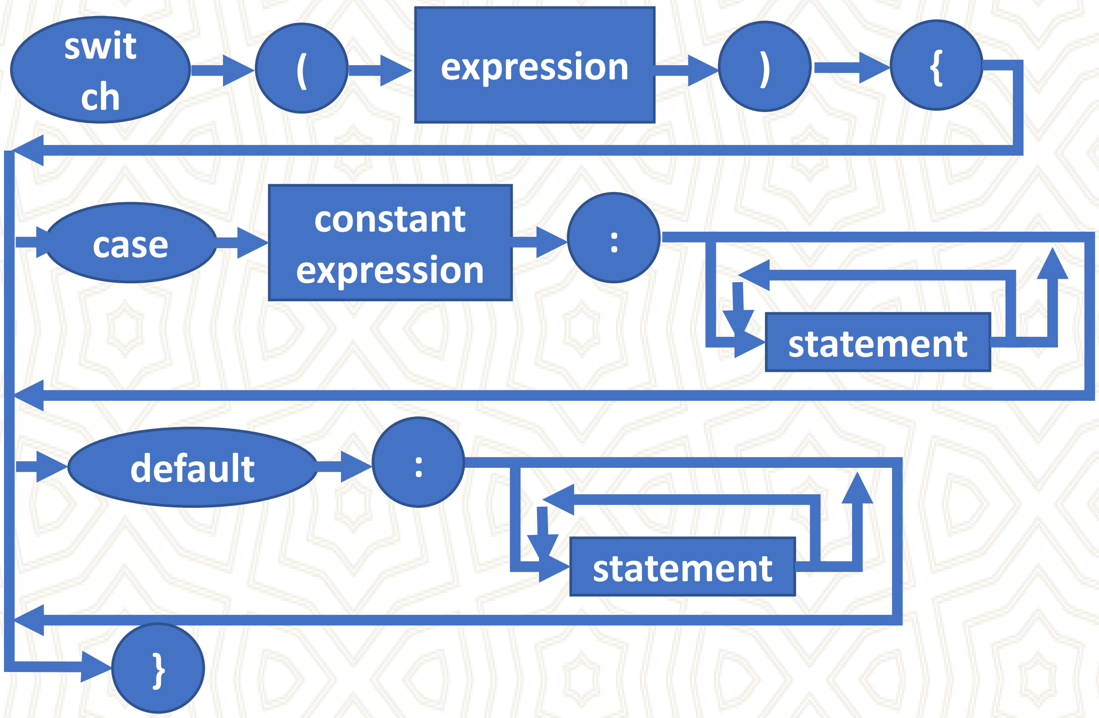
Nested if statements

- Note that when an ***else*** is immediately followed by an ***if***,
  - they are usually placed on the same line.
  - this is commonly called an ***else if*** statement.

- Nested if statements create the problem of matching each else phrase to the right if statement.
  - This is often called the ***dangling else*** problem !
  - An else is always associated with the nearest previous if.

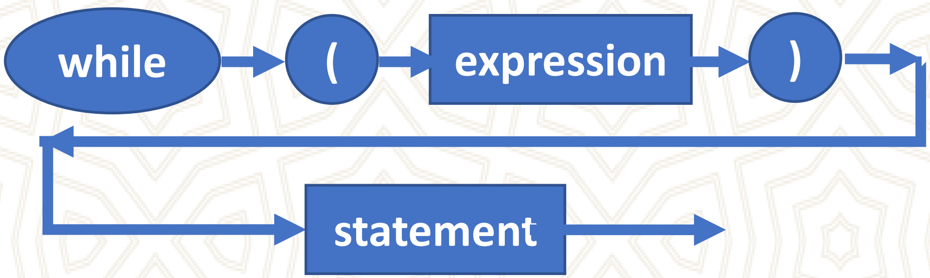
```
if(a<b)
   if(a<c)
          return a;
   else
          return c;
else if (b<c)
          return b;
else
   return c;
```

The switch Statement



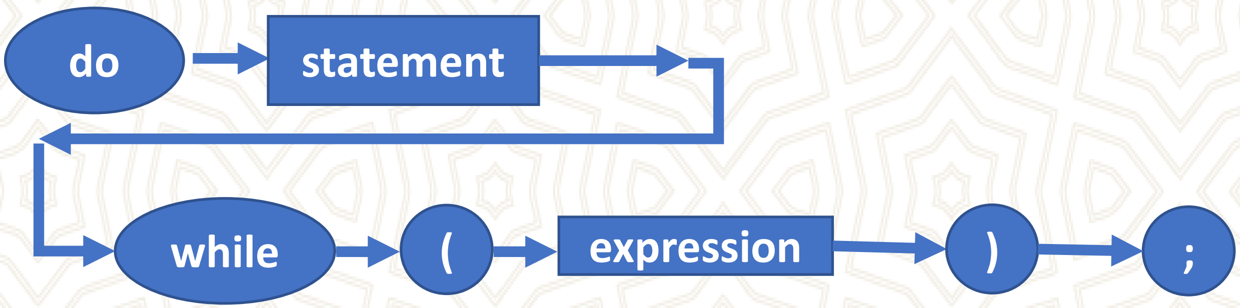
- The ***switch*** expression is evaluated,
  - if it matches one of the case labels, program flow continues with the statement that follows the matching case label.
  - If none of the case labels match the switch expression, program flow continues at the default label, **if exists!**
- No two case labels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

The while Statement



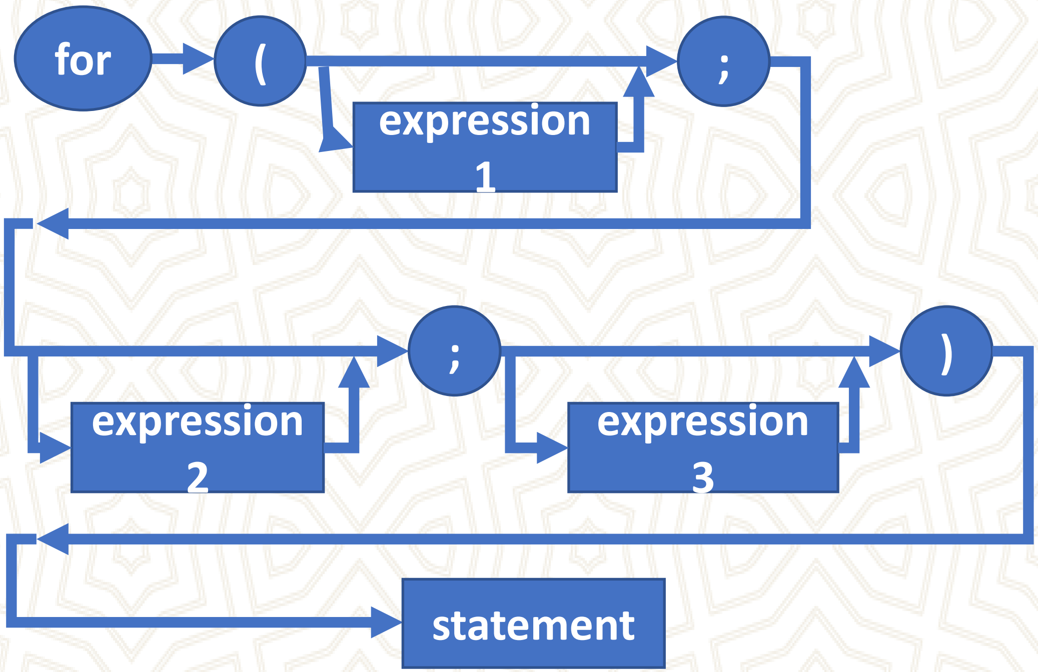
- First the expression is evaluated. If it is a ***nonzero*** value, statement is executed.
- After statement is executed, program control returns to the top of the while statement, and the process is repeated.
- This continues indefinitely until the expression evaluated to zero.

The do...while Statement



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

The for Statement



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

NULL Statements

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

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

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

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

Nested Loops

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

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

break & continue & goto

- ***break***
  - We have already talked about it in ***<u>switch statement</u>***
  - 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

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 !
- Try macro and corresponding function with x=6, y=7 and see the difference

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

```
void print_loc( )  {
        printf("This message 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 *macro_name*

#if !defined macro_name

#if defined (TEST)

#ifdef *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

Additions by Yunus E. SELÇUK

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**.

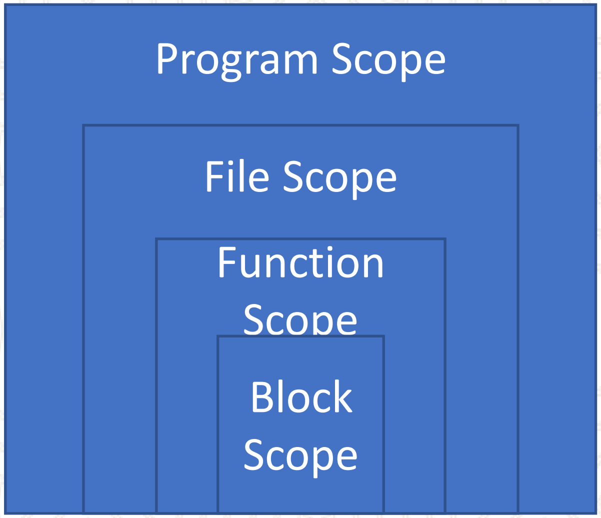
Scope – I

- 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

- A variable with a block scope can NOT be accessed outside its block.

```
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...
     ...
}
```

- 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 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\n", i, j);
    i += 2; j += 3; printf("i:%d j:%d\n", 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

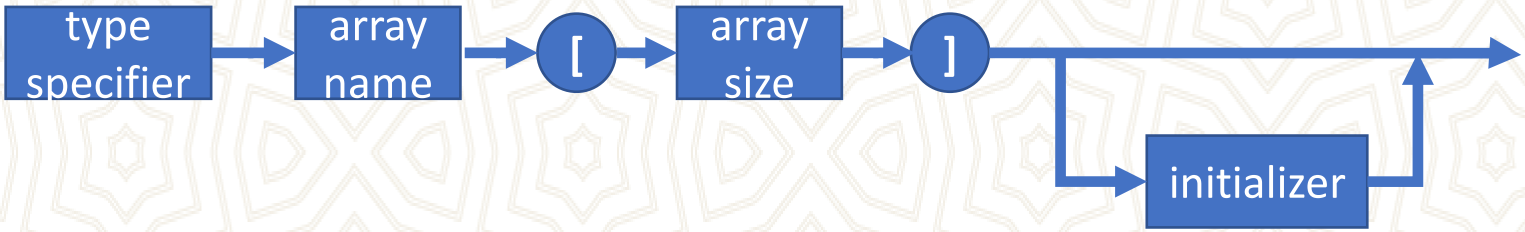
After statement 4:

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] = 18;

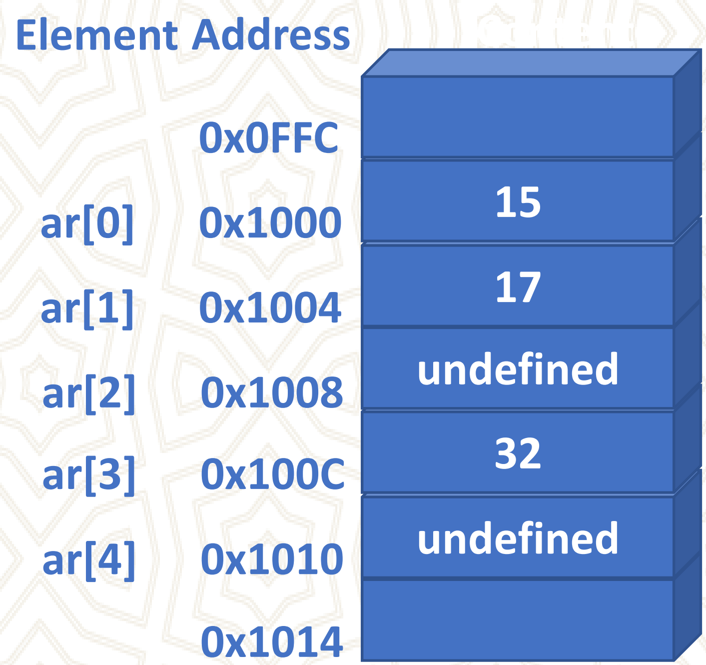
dailyTemp[1] = 23;

- **subscripts begin at 0, not 1 !**
- Type can be defined as void, as done in file operations. Void pointers will not be covered in detail.

How Arrays Stored in Memory

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

- **<u>Note that ar[2] and ar[4] have undefined values!</u>**
  - 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.
- **<u>Note that 3.5 is converted to the integer value 3!</u>**
- 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; **<u>is same as above assignment!</u>**
- *(p+3) refers to the same memory content as ar[3]
- p[3] is also same as *(p+3)

int ai[4];

scanf("%d",&ai[0]);

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

- Let's code without functions (not preferred) and with functions (preferred)
  - It will look like :

```
The unsorted array is:     6    5    3    1    8    7    2    4
The array has become :     5    6    3    1    8    7    2    4
The array has become :     3    6    5    1    8    7    2    4
The array has become :     1    6    5    3    8    7    2    4
The array has become :     1    5    6    3    8    7    2    4
The array has become :     1    3    6    5    8    7    2    4
The array has become :     1    2    6    5    8    7    3    4
The array has become :     1    2    5    6    8    7    3    4
The array has become :     1    2    3    6    8    7    5    4
The array has become :     1    2    3    5    8    7    6    4
The array has become :     1    2    3    4    8    7    6    5
The array has become :     1    2    3    4    7    8    6    5
The array has become :     1    2    3    4    6    8    7    5
The array has become :     1    2    3    4    5    8    7    6
The array has become :     1    2    3    4    5    7    8    6
The array has become :     1    2    3    4    5    6    8    7
The array has become :     1    2    3    4    5    6    7    8
The sorted array is  :     1    2    3    4    5    6    7    8
```

- What are the advantages of functions?
  - (codes\SortBu1.c) vs (codes\SortBu2.c)

Examples: Selection Sort

- Let's code with functions (codes\sortSE1.c)
  - It will look like :

```
The unsorted array is:     6     5     3     1     8     7     2     4
The array has become :     1     5     3     6     8     7     2     4
The array has become :     1     2     3     6     8     7     5     4
The array has become :     1     2     3     4     8     7     5     6
The array has become :     1     2     3     4     5     7     8     6
The array has become :     1     2     3     4     5     6     8     7
The array has become :     1     2     3     4     5     6     7     8
The sorted array is  :     1     2     3     4     5     6     7     8
```

- Compare Bubble Sort with Selection Sort:
  - 16 swaps vs. 6 swaps in this particular input array
  - 28 vs 28 comparisons (HW: How can you count?)
  - $O(n^2)$ vs $O(n^2)$
- Can we avoid global variables?
  - Yes, with function parameters (to be studied later)
- Can we sort an array of an arbitrary size?
  - Yes, with dynamic memory management and pointers (to be studied later)

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";        // not uniformly supported between different C standards
    char *ptr2;
    array = "not OK";              // can NOT assign to an address! Does not compile (☺)
    array[5] = 'A';                // Buggy1 because: Array is not populated yet. So, …
    array[0] = 'O';                // … Always begin from 0 and
    array[1] = 'K';
    array[2] = '\0';               // use null-terminated strings where necessary
    ptr1[8] = 'r';                 // creates a segment violation1. Buggy. See next slide.
    *ptr2 = "not OK";              // Type mismatch warning. Does not compile (☺)
    ptr2="OK";                     // not uniformly supported and would be buggy1
}
//see StrAsnP.prj & strAsgn.c
//for printouts.
```

[1] in DevCPP4, linker gives warning at first but if you make a second attempt, it compiles but see next slide for more discussion.

String Assignments

```
main ()  {
    char *ptr1="10 spaces";
    ptr1[8] = 'r'; //does not work
    *(ptr1+8) = 'r'; //does not work either
}
```

- This code does not work because char pointers that are assigned such constant strings are handled in C as constants/literals.
  - Literals can not modify their data, but they can modify their pointer (i.e. they are read-only).
  - This code results in a segment violation exception and crashes.
- What can we do?
  - We can use the string class of C++ defined in <string.h> and use string objects
    - We will learn object orientation later and in Java programming language
  - We can use regular arrays and live with their restrictions
  - We can try harder:

String Assignments

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char *ptr1= (char*) malloc(10);
    strcpy(ptr1, "10 spaces");
    ptr1[8] = 'r';
    printf("ptr1 :%s\n",ptr1);
    system("pause"); return 0;
}
```

- Allocating memory in a proper way, assigning initial value with strcpy function gives us a string that is not literal/constant.
    - strcpy and some other functions will be introduced shortly.

Strings vs. Chars

Chars

```
char ch = 'a', *p;     // one byte is allocated for 'a'
*p = 'a';              // OK
p = 'a';               // Illegal
printf("%s\n",*p);
```

Strings

```
char *p = "a", *p;     // two bytes allocated for "a"
*p = "a";              // INCORRECT
p = "a";               // OK
printf("%s\n",p);
```

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("You wrote:");
    printf("%s\n",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;
}
```

- The main method will be like :

```
int main () {
   char str1[MAX_CHAR];
   printf("Enter string:");
   scanf("%s",str1);
   printf("Length: %d", strLen(str1));
   return(0);
}
```

- Notice the underlined mappings!

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
  - **c**o**py**, con**cat**enate, **c**o**mp**are, data **t**ype

examine codes\MoreStringOps.c for more examples

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 occurrence
  - 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];
    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 5-element array of 3-element arrays

  int ar[5][3];

  - 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;
```

- examine codes\ArrayOfPointers.c
- codes\StringMatrixOps.c will also be a good example

Element	Address	Memory
	0x0FFC	
ar_of_p[0]	0x1000	2000
ar_of_p[1]	0x1004	2001
ar_of_p[2]	0x1008	undefined
ar_of_p[3]	0x100C	undefined
ar_of_p[4]	0x1010	undefined
	0x1014	

Element	Address	Memory
	0x1FFF	
c0	0x2000	‘a’
c1	0x2001	‘b’
	0x2002	

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

- Complete examples is left after learning dynamic memory management as they will make more sense.

Bu yansı ders notlarının düzeni için boş bırakılmıştır.

Dynamic Memory Allocation

Structural Programming

by Z. Cihan TAYŞİ

Additions by Yunus E. SELÇUK

Outline

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

Memory Allocation Functions (in stdlib.h)

- ***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(void *prev_ptr, total_size_in_bytes)***
  - Changes the size of a previously allocated block prev_ptr.
  - The function may move the memory block to a new location (whose address is returned by the function).
  - 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);
```

Matrix 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
```

Matrix Allocation (cont’d)

```
//use the memory and then
for(i = 0; i < n; i++)  {
        free(mat[i]);
}
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: Sorting (DynamicSorting.c)
  - ask number of elements in the array
  - allocate necessary space
  - ask for elements
  - sort the array
- Write a program: Matrix Multiplication (To do: Left as an exercise to code at home)
  1. ask dimensions of the matrices
  2. check if it is possible to multiple them !
  3. allocate necessary space
  4. ask for elements
  5. perform multiplication
  6. write the result matrix

Example 3

- String matrix operations (bkz. StringMatrixOps.c)

Functions

Structural Programming

by Z. Cihan TAYŞİ

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

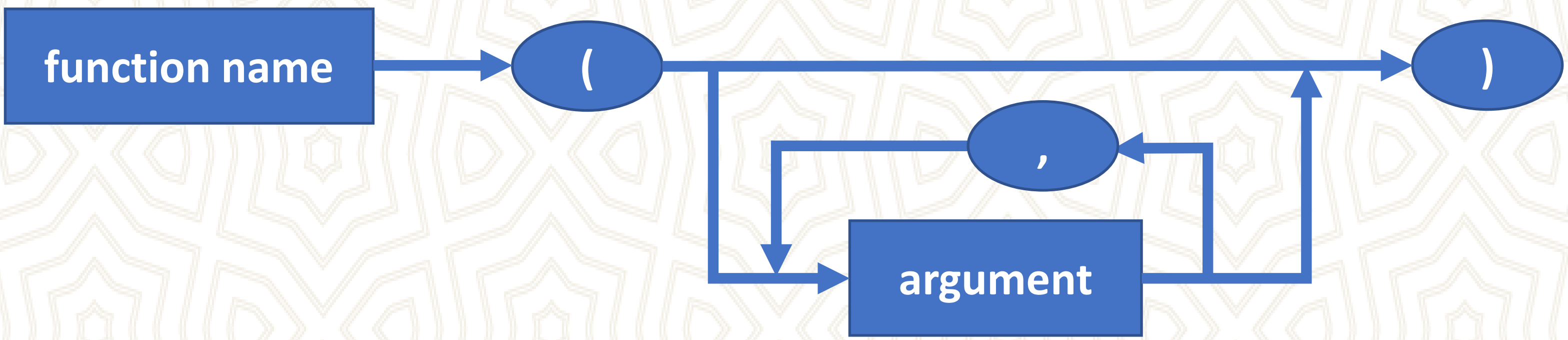
- A very simple example
  - no arguments
  - no return

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

- A relatively complex example
  - a function to calculate factorial n
  - PS: Beware of the value range of int

```
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

- 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

- 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>
#include <stdlib.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));
    free(ptrM);
    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>
#include <stdlib.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

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
- getopt(int argc, char *const argv[], const char *optstring)
  - Simply delegate the argc and argv parameters of the main function to the getopt function.
  - optstring is simply a list of characters, each representing a single character option.
    - : (full column) has special meaning that this option requires an additional argument.
    - “abc:d" accepts the options a, b, c, and d; c requires an additional and mandatory argument.
    - GNU C introduces double :: where the argument is optional, not mandatory.
- 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.

Fore more details and an example, please refer to: http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html

More on the Main Function

- A better way to handle command line arguments
  - getopt
  - Argp
  - Optopt
  - suboptions

```
I wanted to put some code here but getopt is
used most efficiently in Linux. As a result,
the exams will not cover this topic.
```

Structures and Unions

Structural Programming

by Z. Cihan TAYSI

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

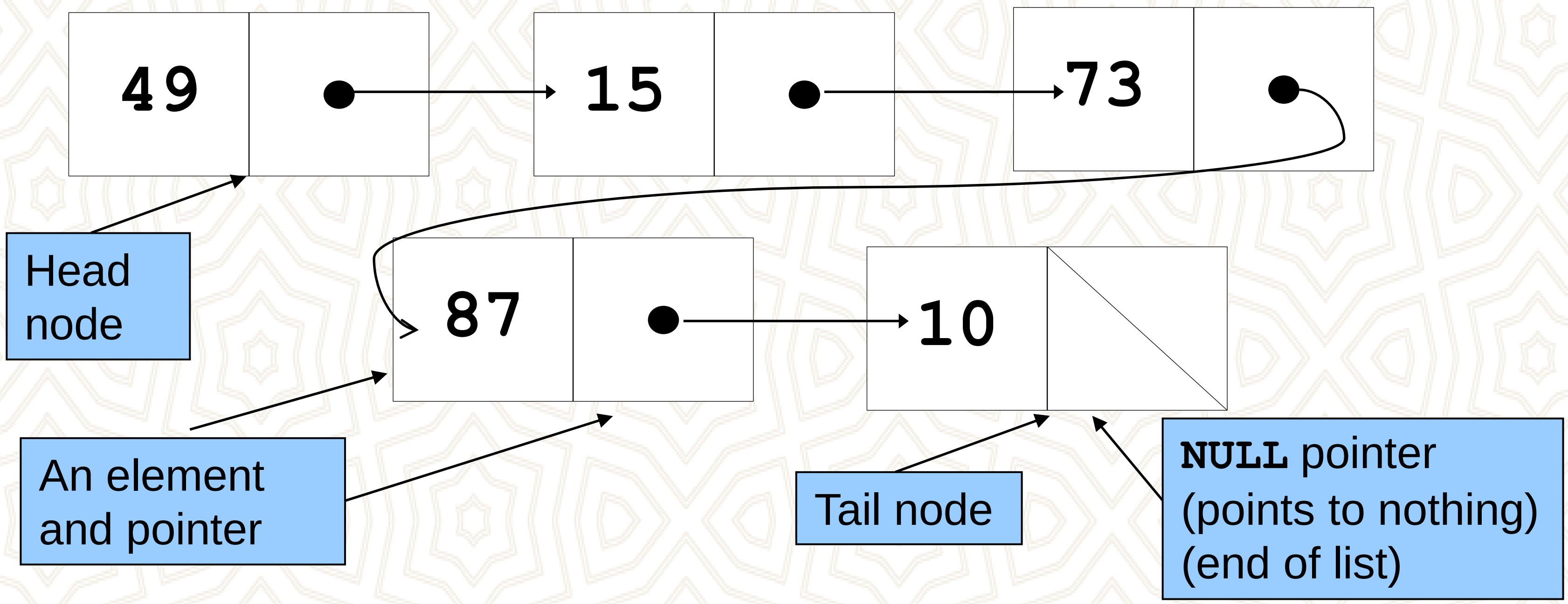
- There are two ways to pass structures as arguments:
  - pass the structure itself (called pass by value)
    ```
    EMPLOYEE emp;
    printReport(emp);
    ```
  - pass a pointer to the structure (called pass by reference)
    ```
    EMPLOYEE emp;
    increaseSalary(&emp);
    ```
- Passing by reference 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);
    //can’t get non-pointer struct fields directly in some platforms
    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

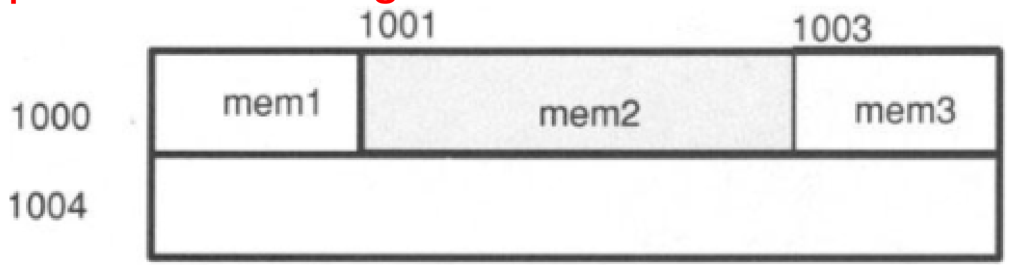
- Some computers require that any data object larger than a char must be assigned an address that is a multiple of a power of 2 (all objects larger than a char are to be stored at even addresses).
- Normally, these alignment restrictions are invisible to the programmer. However, they can create holes, 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;
```

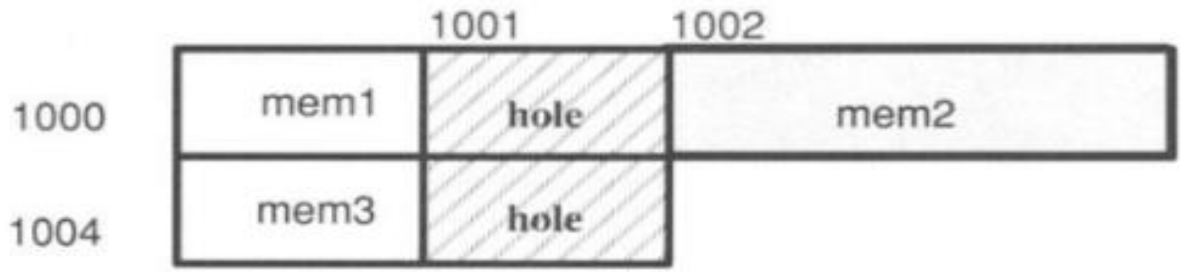
Structure Alignment

- 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 larger than a char to be stored at even addresses, s1 would be stored as:



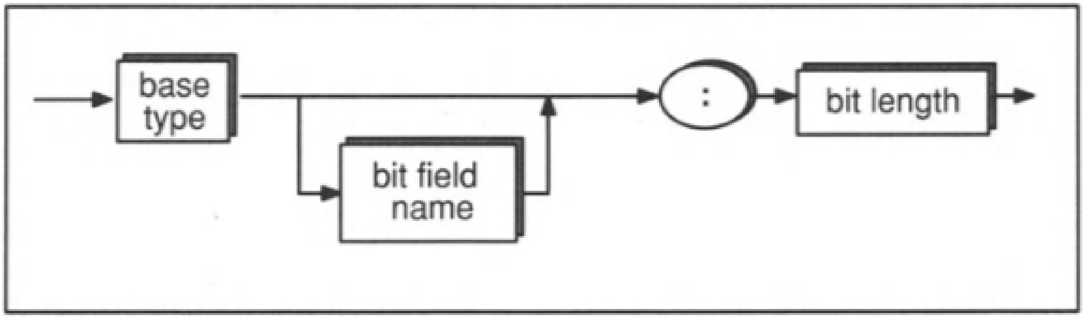
*This storage arrangement results in a l-byte hole between mem1 and mem2 and following mem3.

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:

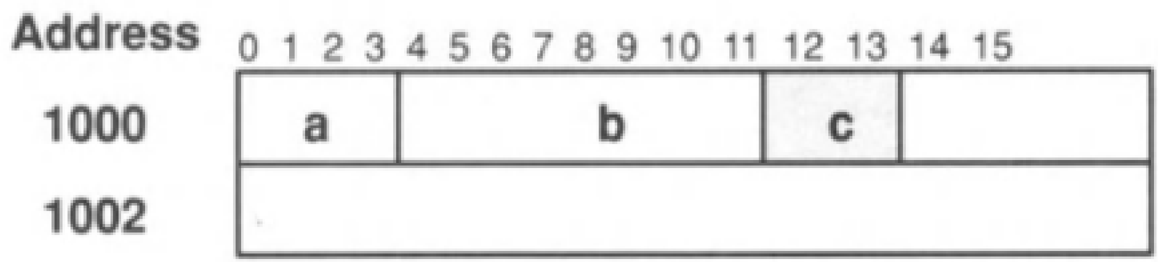


- The base type may be **int, unsigned int**, or **signed int**.
- If the bit field is declared as int, the implementation is free to decide whether it is an unsigned int or a signed int (**For portable code, use the signed or unsigned qualifier**).
- The *bit length* is an integer constant expression 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

- Assuming your compiler allocates 16-bits for a bit field, the following declarations would cause *a, b,* and *c* to be packed into a single 16-bit object:

```
struct
{
   int a : 3;
   int b : 7;
   int c : 2;
} s;
```

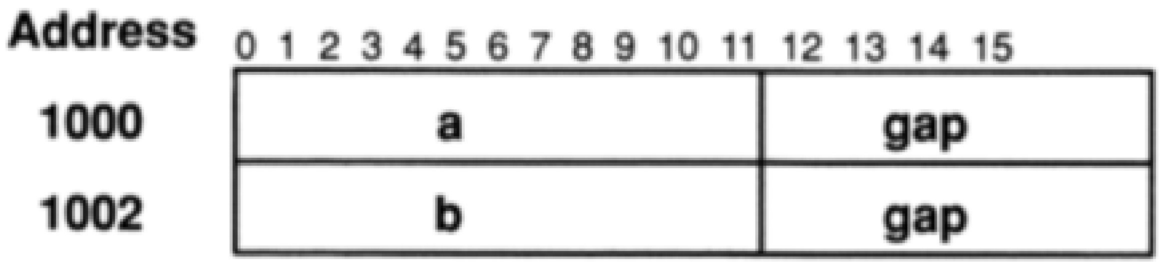


- PS: Each implementation is free to arrange the bit fields within the object in either increasing or decreasing order, depending on the compiler

Bit fields

- If a bit field would located in an **int** boundary, a new memory area may be allocated, depending on your compiler. For instance, the declaration might cause a new 16-bit area of memory to be allocated for *b:*

```
struct
{
   int a : 10;
   int b : 10;
} s;
```



Bit fields

- Consider *DATE* structure example:

```
struct DATE{
  unsigned int day : 5;
  unsigned int month : 4;
  unsigned int year : 11;
};
```

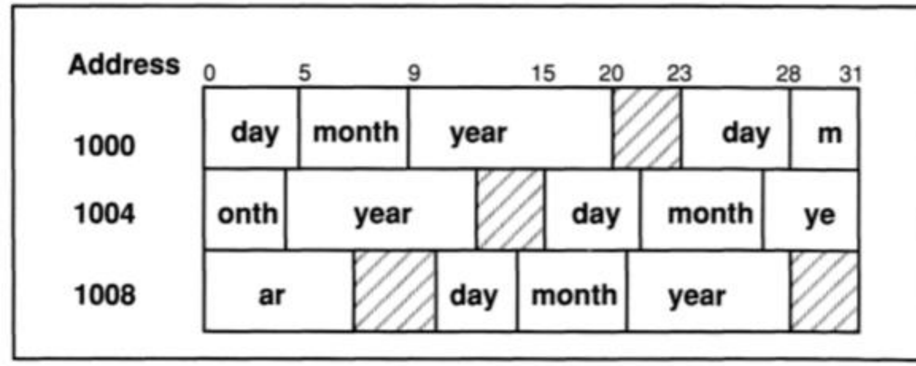
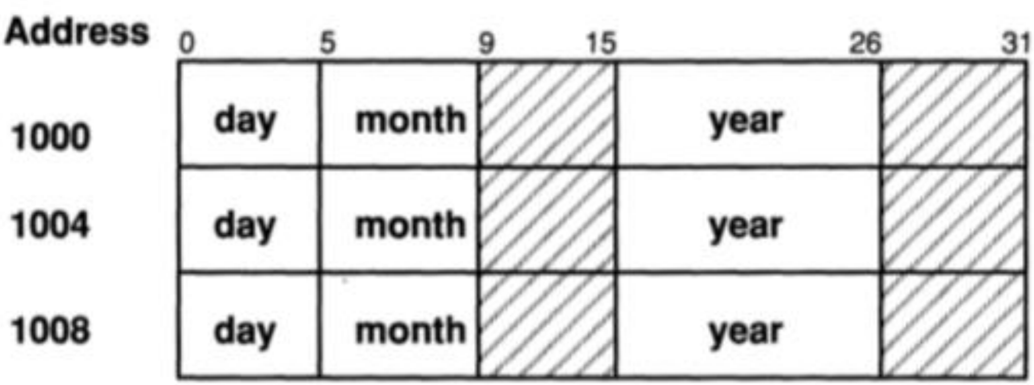


Figure 8-8. Storage of the DATE *Structure with Bit Fields. This figure assumes that the compiler packs bit fields to the nearest* **char** *and allows fields to span* **int** *boundaries.*



Alternative Storage of the DATE Structure with Bit Fields. The left figure assumes that the compiler packs bit fields to the nearest **short** and **does not allow fields to span int boundaries**.

- It can be seen that, in order to ensure optimum memory allocation, you need to know some details of your environment but you can always take some precautions by: changing the order of your fields, allocating a few extra bits in order to save more memory, etc.

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

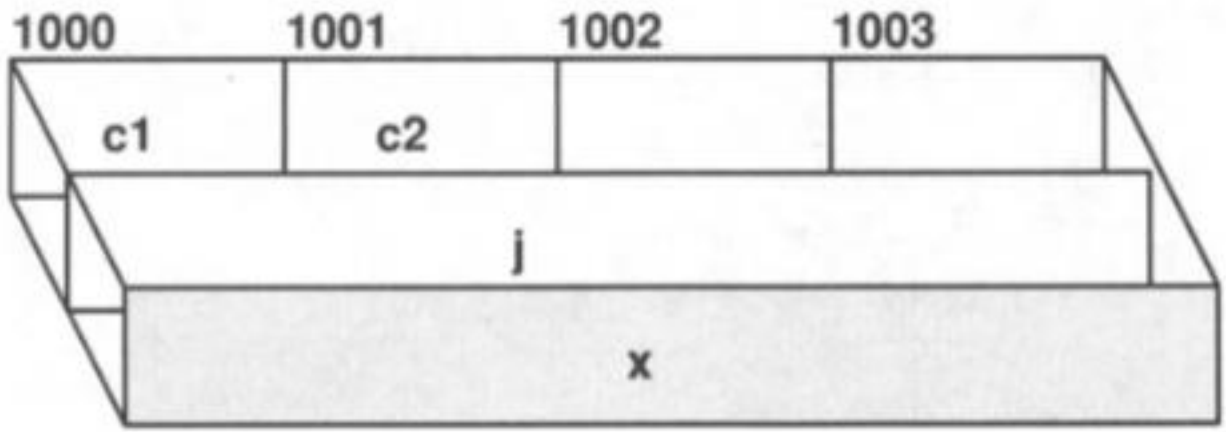
- Unions are similar to structures except that the members are overlaid one on top of another, so members share the same memory.
- There are two basic applications for unions:
  - Interpreting the same memory in different ways.
  - Creating flexible structures that can hold different types of data.

Unions

- Example:

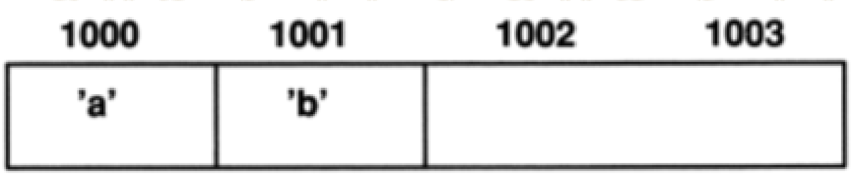
```
typedef union
{
  struct
  {
    char c1, c2
  } s;
  long j;
  float x;
} U;

U example;
```



- Usage:

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



- If you make the assignment:

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

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 : 4;
   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

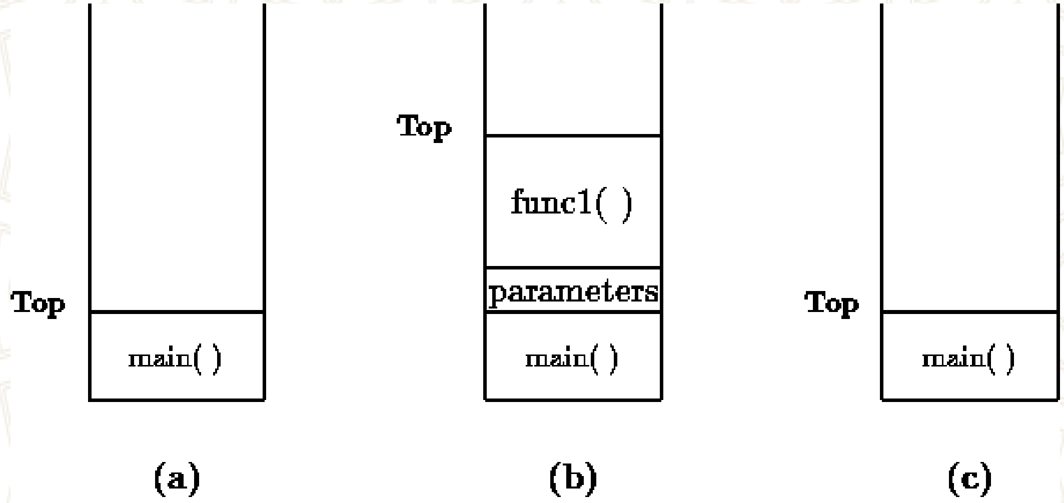
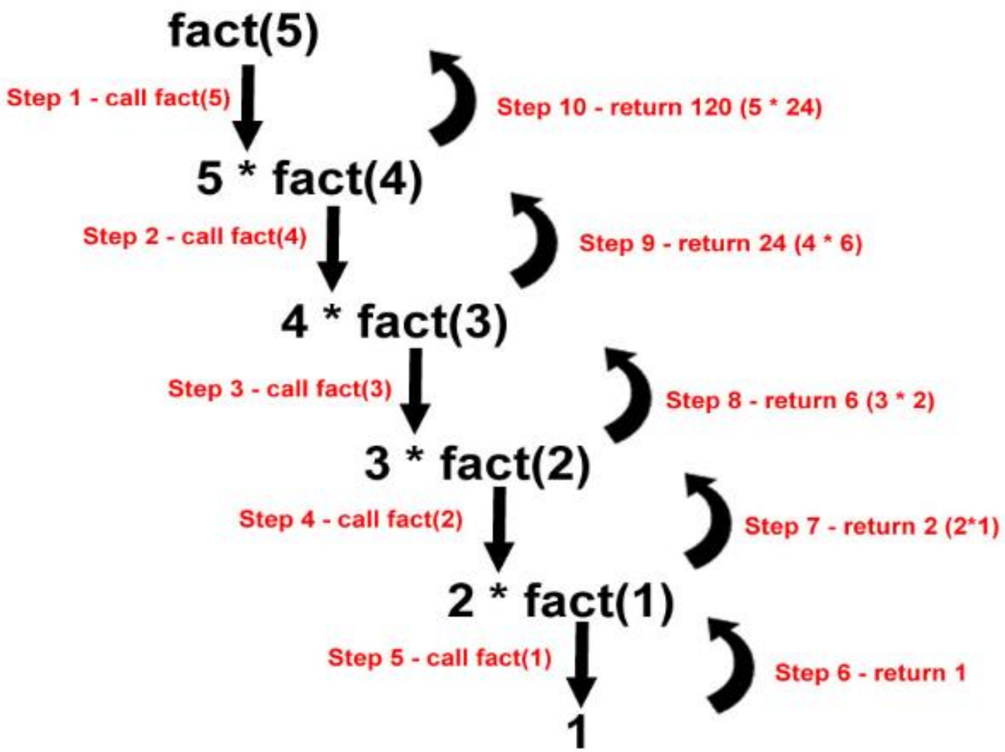


Figure 14.13: Organization of the Stack

- 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)**
- 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

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 <u>**(*pf)**</u> (); // 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 but compiler dependent:
                       &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;
}
```

Bu yansı ders notlarının düzeni için boş bırakılmıştır.

File Input-Output (I/O)

Structural Programming

by Zeyneb YAVUZ

Corrections and additions

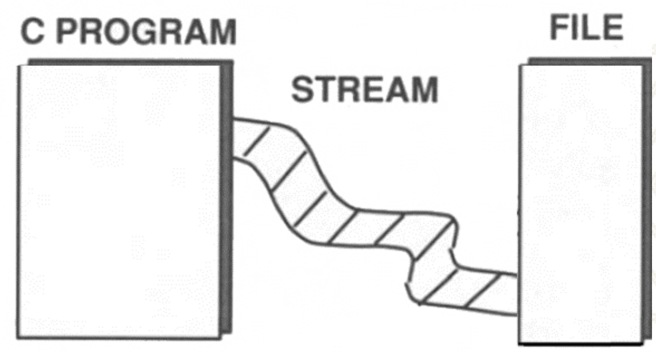
by Yunus Emre SELÇUK and Ahmet ELBİR

Input and Output (I/O)

- Operating systems (OSs) vary greatly in the way they allow access to data in files and devices.
- This variation makes it extremely difficult to design I/O programs that are portable.
- The C language performs I/O through a large set of runtime routines. Some of these functions were derived from the UNIX I/O library.
- However:
  - The "C library" deals mostly with buffered I/O while the UNIX library performs unbuffered I/O.
  - The UNIX OS treats binary and text files the same. In some other OSs, the distinction is extremely important.
- ANSI Committee preserved, deleted, and modified some functions:
  - The biggest change is: elimination of unbuffered I/O functions. In the ANSI library, all I/O functions are buffered, still you can change the buffer size.
  - The ANSI I/O functions make a distinction between accessing files in binary or text mode (to be examined in more detail shortly).

I/O and Streams

- C makes no distinction between devices such as a terminal or tape drive or files on a disk.
- In all cases, I/O is performed through *streams* that are associated with the files or devices.



- A stream consists of an ordered series of bytes (such as a one-dimensional array of characters, as shown in the Figure).
- Reading and writing to a file or device involves reading data from the stream or writing data onto the stream.
- To perform I/O operations, you must associate a stream with a file or a device.
- You do this by declaring a pointer to a structure type called *FILE*.
- *The FILE* structure, will be examined later in more detail.

Standard Streams

- There are three default streams that are automatically opened for every program:
  - *stdin, stdout and stderr.*
- Usually, these streams point to your terminal, but many operating systems permit you to redirect them (eg you might want error messages written to a file instead of the terminal).
- The I/O functions already introduced, eg printf() and scanf(), use these default streams.
- printf() writes to stdout, and scanf() reads from stdin.
- You could use these functions to perform I/O to files by making stdin and stdout point to files (with the freopen() function).
- However an easier method is to use the equivalent functions, fprintf() and fscanf(), which enable us to specify a particular stream.

Text and Binary Formats

- Data can be accessed in one of two formats: *text* or *binary*.
- A *text stream consists of a series of lines, where each line is* terminated by a newline ('\n') character.
- However, OSs may have other ways of storing lines on disks, so each line in a text file does not necessarily end with a newline character.
- E.g. many IBM systems, keep track of text lines through an index of pointers to the beginning of each line.
- In this scheme, the files stored on disk or tape may not contain newline characters even though they are logically composed of lines.
- However, when these lines are read into memory in *text mode*, the runtime functions automatically insert newlines into the text stream.

Text Format

- When lines are written from/to a text stream, the I/O functions may replace new lines in the stream with implementation-defined characters that get written to the I/O device.
- In this way, C text streams have a consistent appearance from one environment to another, even though the format of the data on the storage devices may vary.
- Despite this rule that promotes portability somewhat, be extremely careful when performing textual I/O: Programs that work on one system may not work exactly the same way on another.
- In particular, the rules described above hold true only for printable characters (e.g. tabs, form feeds, and newlines).
- If control characters (non-printable characters) appear in a text stream, they are interpreted in an implementation-defined manner.

Binary Format

- In binary format, the compiler performs no interpretation of bytes. It simply reads and writes bits exactly as they appear.
- Binary streams are used primarily for non-textual data, where there is no line structure and it is important to preserve the exact contents of the file.
- If you are more interested in preserving the line structure of a file, you should use a text stream.
- The 3 standard streams (*stdin, stdout, stderr)* are all opened in text mode.
- In UNIX environments the distinction between text and binary modes is superficial since UNIX treats all data as binary data.
- However, even when programming in a UNIX environment, you should beware of potential difficulties in porting to other systems

Using Streams via the FILE Structure

- To perform I/O operations, you must associate a stream with a file or a device.
- You do this by declaring a pointer to a structure type called *FILE.*
- *The FILE* structure, which is defined in the *stdio.h header file, contains several fields to* hold such information as the file's name, its access mode, and a pointer to the next character in the stream.
- These fields are assigned values when you open the stream and access it, but they are implementation dependent, so they vary from one system to another.
- The *FILE structures provide the OS some metadata information*, but our only chance to access to the stream is the pointer to the *FILE structure (called a file pointer).*

Using Streams via the FILE Structure

- *The file pointer, which you must declare in* your program, holds the stream identifier returned by the *fopen() function.*
- You use the file pointer to read from, write to, or close the stream.
- A program may have more than one stream open simultaneously, although each implementation imposes a limit on the number of concurrent streams.
- One of the fields in each *FILE structure is a file position indicator that points to* the byte where the next character will be read from or written to.
- As you read from or write to the file, the OS adjusts the file position indicator to point to the next byte.

Using Streams via the FILE Structure

- Although you can't directly access the file position indicator (at least not in a portable fashion), you can fetch and change its value through library functions, thus enabling you to access a stream in non-serial order.
- Do not confuse the file pointer with the file position indicator:
  - The file pointer identifies an open stream connected to a file or device.
  - The file position indicator refers to a specific byte position (i.e. next character) within a stream

The <stdio.h> Header File

- To use any of the I/O functions, include the stdio.h, which contains:
  - Prototype declarations for all the I/O functions.
  - Declaration of the FILE structure.
  - Several useful macro constants, including stdin, stdout, stderr, EOF, and NULL.
- stdin, stdout, stderr: Standard streams
- EOF: The value returned by many functions when the system reaches the end-of-file marker.
- NULL: The name for a null pointer. It can be defined in another header file called stddef.h.
- To use NULL, you must either include stdio.h or stddef.h

Opening a File: fopen function

- Before you can read from or write to a file, you must open it with the *fopen()* function.
- *fopen()* takes 2 arguments:
  - 1st parameter is the file name
  - 2nd parameter is the access mode
- There are two sets of access modes:
  - One for text streams and
  - One for binary streams.

Opening a File: fopen function

- Access modes for text streams is on the right side.
- The binary modes are exactly the same, except that they have a “b” appended to the mode name.
- For example to open a binary file with read access you would use "rb".

“r”	Open an existing text file for reading. Reading occurs at the beginning of the file.
“w”	Create a new text file for writing. If the file already exists, it will be truncated to zero length. The file position indicator is initially set to the beginning of the file.
“a”	Open an existing text file in append mode. You can write only at the end-of-file position. Even if you explicitly move the file position indicator, writing still occurs at the end-of-file.
“r+”	Open an existing text file for reading and writing. The file position indicator is initially set to the beginning of the file.
“w+”	Create a new text file for reading and writing. If the file already exists, it will be truncated to zero length.
“a+”	Open an existing file or create a new one in append mode. You can read data anywhere in the file, but you can write data only at the end-of-file marker.

Opening a File: fopen function

- File and Stream properties of fopen() modes:

	r	w	a	r+	w+	a+
File must exist before open	*			*		
Old file truncated to zero length		*			*	
Stream can be read	*			*	*	*
Stream can be written		*	*	*	*	*
Stream can be written only at end			*			*

Opening a File: fopen function

- *fopen()* returns a file pointer of type FILE that you can use to access the file later in the program (check the example code).
- *fopen()* returns a null pointer *(NULL)* if an error occurs.
- If *successful, fopen()* returns a non-zero file pointer.
- *fprintf()* is exactly like *printf(),* except that it takes an extra argument indicating which stream the output should be sent to.

I/O Operations: Reading and Writing Data

- Once you have opened a file, you use the file pointer to perform read and write operations.
- There are three ways to perform I/O operations on three different sizes of objects:
  - **One character at a time: getc and putc functions**
  - **One line at a time: fgets and fputs functions**
  - **One block at a time: fread and fwrite functions**
- Each of these methods has some pros and cons that will be discussed later.

I/O Operations: Reading and Writing Data

- One rule that applies to all levels of I/O is:
  - You cannot read from a stream and then write to it without an intervening call to *fseek(), rewind(),* or *fflush().*
  - The same rule holds for switching from write mode to read mode.
  - These three functions are the only I/O functions that flush the buffers.
- If you do not have memory shortage, you can read from input the stream, construct and keep the output data in the memory and finally write to the output stream
  - The input and output streams can point to the same file, but close the file that you have read before writing to it.

Closing a File: fclose function

- To close a file, you need to use the fclose() function: fclose(fp);
- Closing a file frees up the FILE structure that fp points to so, the OS can use the structure for a different file.
- It also flushes any buffers associated with the stream.
- Most OSs have a limit on the number of streams that can be open at once, so it's a good idea to close files when you're done with them.
- In any event, all open streams are automatically closed when the program terminates normally.
- Most OSs will close open files even when a program aborts abnormally, but you can't depend on this behavior.

I/O Example #1: Copy operation

- Reading and writing one character at a time in binary mode:

When the end-of-file is encountered, the feof() function returns a non-0 value

```
#include <stdio.h>
#define FAIL 0
#define SUCCESS 1
int copyfile(char * infile, char * outfile){
  FILE *fp1, *fp2;
  if ((fp1 = fopen(infile, "rb" )) == NULL) {
    printf("Could not open the input file\n"); return FAIL;
  }
  if ((fp2=fopen (outfile, "wb" )) == NULL) {
    printf("Could not open the output file\n"); fclose( fp1 ); return FAIL;
  }
  while (!feof( fp1 ))
    putc( getc(fp1), fp2 );
  fclose(fp1); fclose(fp2); return SUCCESS;
} //to be continued with the main method
```

I/O Example #1: Copy operation

- More on how to determine EOF:
- we cannot use the return value of getc() to test for an end-of-file character because the file is opened in the binary mode.
- For example, if we wrote:

```
int c;
while ((c = getc( fp1 )) != EOF)
    putc( getc(fp1), fp2 );
```

- the loop will exit whenever the character read has the same value as EOF.
- This may or may not be a true end-of-file condition in binary files.
- Only the feof() function will exactly provide us to check if we reach the enf-of-file.

I/O Example #1: Copy operation

- The rest of the example:

```
int main(){
   char infl[100], outfl[100];
   int result;
   printf("enter name of the input file\n");  scanf("%s", infl);
   printf("enter name of the output file\n"); scanf("%s", outfl);

   result=copyfile(infl, outfl);
   if(result == SUCCESS)
     printf("input file is copied to the output file \n");
   if(result == FAIL)
     printf("input file could not be copied to the output file \n");
   return 0;
}
```

Character I/O:

- Four functions that read and write one character to a stream:
  - *getc()* A macro that reads one character from a stream.
  - *fgetc()* Same as *getc(),* but implemented as a function.
  - *putc()* A macro that writes one character to a stream.
  - *fputc()* Same as *putc(),* but implemented as a function.
- *getc()* and *putc()* are usually implemented as macros whereas *fgetc()* and *fputc()* are guaranteed to be functions.
- Because *putc* and *getc* are implemented as macros, they usually run much faster. They are almost twice as fast as *fgetc* and *fputc*
- However since they are macros, they are susceptible to side effect problem e.g. this is a dangerous call that may not work as expected: putc(’x', fp[j++]);
- If an argument contains side effect operators, you should use *fgetc()* or *fputc(),* which are guaranteed to be implemented as functions.

I/O Example #2: Copy operation

- Reading and writing one line at a time in **text mode**:

The difference is in the while loop and its body

```
#include <stdio.h>
//#include <stddef.h>
#define FAIL 0
#define SUCCESS 1
#define LINESIZE 100
int copyfile(char * infile, char * outfile){
  FILE *fp1, *fp2; char line[LINESIZE];
  if ((fp1 = fopen(infile, "r" )) == NULL) {
    printf("Could not open the input file\n"); return FAIL;
  }
  if ((fp2=fopen (outfile, "w" )) == NULL) {
    printf("Could not open the output file\n"); fclose( fp1 ); return FAIL;
  }
  while (fgets ( line, LINESIZE-1, fp1 ) != NULL)
    fputs( line, fp2 );
  fclose(fp1); fclose(fp2); return SUCCESS;
} //the main method will be the same as the previous example
```

Line I/O:

- There are two line-oriented *I/0 functions-fgets()* and *fputs().*
- The prototype for *fgets()* is: char *fgets(char *s, int n, FILE stream);
- The three arguments of fgets():
  - *s:* A pointer to the 1^{st} element of an array to which characters will be written.
  - *N:* An integer representing the max number of characters to read.
  - *stream* The stream from which to read.
- *fgets()* reads characters until it reaches a newline, or the end-of-file, or the maximum number of characters specified.
- *fgets()* automatically inserts a null character after the last character written to the array.
- So, we specify the "n" parameter to be one less than the "s" array #.
- *fgets()* returns *NULL* when it reaches the end-of-file.
- Otherwise, it returns the first argument ("s" string).

Line I/O:

- The prototype for *fputs()* is: fputs(char *s, FILE stream)
- *fputs()* writes the array identified by the 1st argument to the stream identified by the 2nd argument.
- In the code example, copying a binary file with line I/O produced a corrupt file.

fgets() vs gets():

- *gets()* is the function that reads lines from *stdin.*
- Both functions append a null character ('\0') after the last character written.
- However, *gets()* does not write the terminating newline character to the input array. *fgets()* does include the terminating newline character (or an EOF if it just got the last line of the file).
- Also, *fgets()* allows you to specify a maximum number of characters to read, whereas *gets()* reads characters indefinitely until it encounters a newline or EOF.

Block I/O:

- We can also access data in lumps called *blocks,* where each block is stored in an array.
- When you read or write a block, you need to specify the number of elements in the block and the size of each element.
- The two block I/O functions are: *fread()* and *fwrite().*
- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
- void fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
  - size_t: an integer *type* defined in stdio.h
  - *ptr:* A pointer to an array (mostly char), in which to store data.
  - *size* The size of each element in the array.
  - *nmemb* The number of elements to read.
  - *stream* The file pointer.

Block I/O:

- *fread()* returns the number of elements actually read, which should be the same as the 3rd argument unless an error occurs or an EOF condition is encountered.
- The *fwrite()* is the mirror of *fread()*, takes the same arguments, but instead of reading elements from the stream to the array, it writes elements from the array to the stream.
- The block sizes in *fread()* and *fwrite()*, do not affect the number of device I/O operations performed
  - The buffer size, for instance, might be 1024 bytes. If the block size specified in a read operation is only 512 bytes, the OS will still fetch 1024 bytes from the disk and store them in memory.
  - But, only the first 512 bytes will be available to the *fread()*.
  - On the next *fread()* call, the OS will fetch the remaining 512 bytes from memory rather than performing another disk access.

I/O Example #2: Copy operation

- Reading and writing by blocks in binary mode:

```
#include <stdio.h>
#include <stddef.h>
#define FAIL 0
#define SUCCESS 1
#define BLOCKSIZE 512
typedef char DATA;
//continued in the next slide
```

I/O Example #2: Copy operation

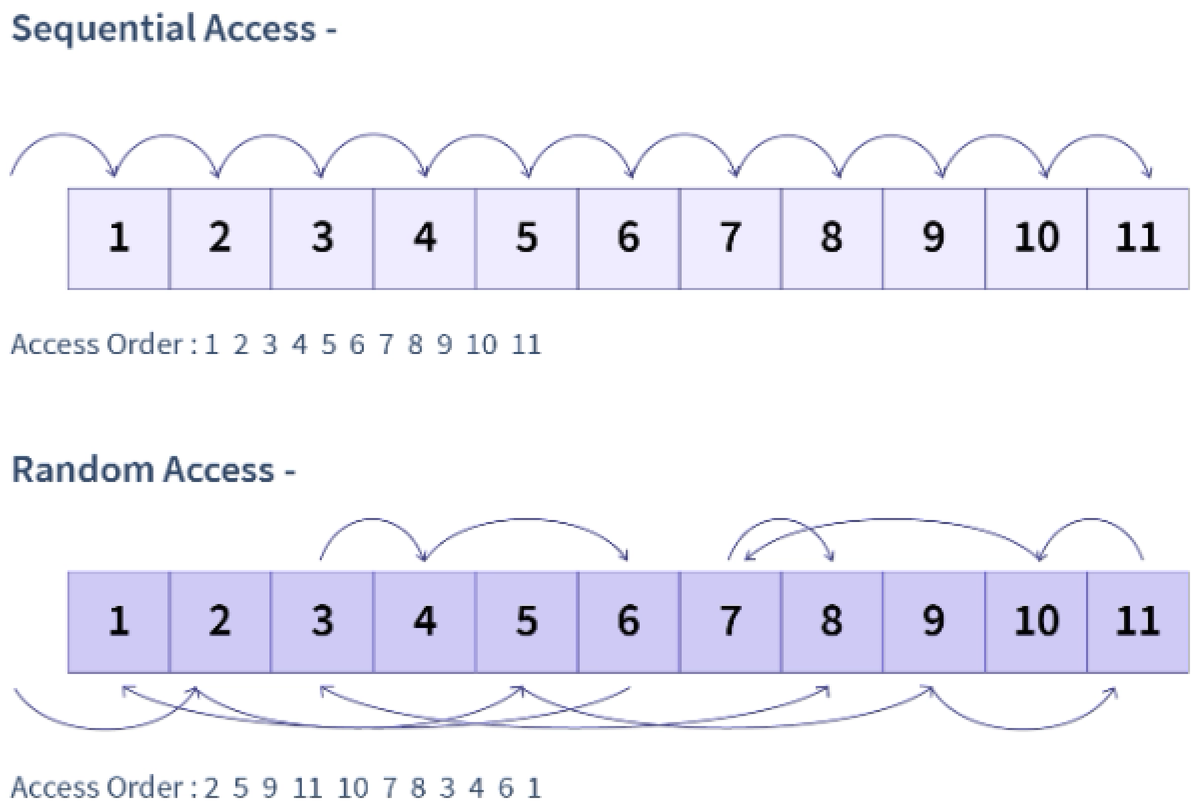
- Reading and writing by blocks in binary mode:

```
int copyfile(char * infile, char * outfile){
  FILE *fp1, *fp2; int num_read; char block[BLOCKSIZE];
  if ((fp1 = fopen( infile, "rb" )) == NULL){
    printf( "Error opening file %s for input.\n", infile ); return FAIL;
  }
  if ((fp2 = fopen( outfile, "wb" )) == NULL){
    printf( "Error opening file %s for output.\n", outfile );
    fclose(fp1); return FAIL;
  }
  while ((num_read = fread( block, sizeof(DATA), BLOCKSIZE, fp1 )) == BLOCKSIZE)
    fwrite( block, sizeof(DATA), num_read, fp2 );
  fwrite( block, sizeof(DATA), num_read, fp2 ); //notice this line!
  fclose(fp1); fclose(fp2);
  if (ferror(fp1)) { printf( "Error reading file %s\n", infile ); return FAIL; }
  return SUCCESS;
} //the main method will be the same as the previous example
```

Random Access:

- So far we accessed files sequentially, beginning with the 1st byte and accessing each successive byte in order.
- For some applications this can be reasonable.
- However, for some applications, you need to access particular bytes in the middle of the file.
- In this case, we use 2 random access functions: fseek() and ftell().

Random Access:



Random Access: fseek

- The fseek() moves the file position indicator to a specified character in a stream:

  ```
  int fseek( FILE *stream, long int offset, int whence);
  ```

- stream: A file pointer
- offset: An offset measured in characters (can be negative).
  - Binary: # of bytes.
  - Text: Either 0, or a value returned by ftell().
- whence: The starting position from which to count the offset.
- 3 choices for whence (defined in stdio.h):
  - SEEK_SET: The beginning of the file.
  - SEEK_CUR: The current position of the file position indicator
  - SEEK_END: The end-of-file position.

Random Access: fseek

- For example: stat = fseek(fp, 10, SEEK_SET);
  - Moves the file position indicator to character 10 of the stream. This will be the next character read or written.
  - Streams, like arrays, start at the 0-th position, so character 10 is actually the 11-th character in the stream.
  - The value returned by *fseek()* is 0 if the request is legal. Otherwise, it returns a non-0 value.
    - This can happen for a variety of reasons, the following is illegal if *fp* is opened for read-only access because it attempts to move the file position indicator beyond the end-of-file position: stat = fseek(fp, 1, SEEK_END)
- If *SEEK_END* is used with read-only files, the offset value must be less than or equal to 0.
- Similarly, if *SEEK_SET* is used, the offset value must be greater than or equal to 0.
- Do not push the file position indicator out of the file

Random Access: ftell

- The *ftell()* takes just one argument, which is a file pointer, and returns the current position of the file position indicator.
- *ftell()* is used to return to a specified file position after performing one or more *I/O* operations
- The position returned by ftell() is measured from the beginning of the file:
  - For binary streams, the value returned by ftell() represents the actual number of characters from the beginning of the file.
  - For text streams, the value returned by ftell() represents an implementation-defined value that has meaning only when used as an offset to an fseek() call.

How to use the ftell() function in C

Highlights:

1. ftell() is used to find the position of the file pointer from the starting of the file.

2. Its syntax is as follows:

```
ftell(FILE *fp)
```

In C, the function ftell() is used to determine the file pointer's location relative to the file's beginning. ftell() has the following syntax:

```
pos = ftell(FILE *fp);
```

Ftell.c

How to use the rewind() function in C

Highlights:

1. rewind() is used to move the file pointer to the beginning of the file.

2. Its syntax is as follows:

```
rewind(FILE *fp);
```

The file pointer is moved to the beginning of the file using this function. It comes in handy when we need to update a file. The following is the syntax:

Rewind.c

How to use the fseek() function in C

Highlights:

1. The fseek() function moves the file position to the desired location.

2. Its syntax is:

```
int fseek(FILE *fp, long displacement, int origin);
```

To shift the file position to a specified place, use the fseek() function.

Syntax:

```
int fseek(FILE *fp, long displacement, int origin);
```

fseek.c

The various components are as follows:

- *fp* – file pointer.
- *displacement* - represents the number of bytes skipped backwards or forwards from the third argument's location. It's a long integer that can be either positive or negative.
- *origin* – It's the location relative to the displacement. It accepts one of the three values listed below.

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

Operation	Description
fseek(fp, 0, 0)	This takes us to the beginning of the file.
fseek(fp, 0, 2)	This takes us to the end of the file.
fseek(fp, N, 0)	This takes us to (N + 1)th bytes in the file.
fseek(fp, N, 1)	This takes us N bytes forward from the current position in the file.
fseek(fp, -N, 1)	This takes us N bytes backward from the current position in the file.
fseek(fp, -N, 2)	This takes us N bytes backward from the end position in the file.

Random Access Example

- Consider a large data file composed of records, where each record is a *PERSONALSTAT* structure, as declared earlier weeks.
- Suppose that the records are arranged randomly, but we want to print them alphabetically by the *surname* field. First, you need to sort the records.
- We want to avoid sorting as it will take a lot of time and I/O operations.
- Instead of sorting, let’s create an index and sort only the index

Random Access Example

Suppose that the first five records have the following values.

```
Jordan, Larry      043-12-7895       5-11-1954
Bird, Michael      012-45-4721       3-24-1952
Erving, Isiah      065-23-5553       11-01-1960
Thomas, Earvin     041-92-1298       1-21-1949
Johnson, Julius    012-22-3365       7-15-1957
```

The key/index pairs would be

```
index     key
0         Jordan, Larry
1         Bird, Michael
2         Erving, Isiah
3         Thomas, Earvin
4         Johnson, Julius
```

Instead of physically sorting the entire records, we can sort the key/index pairs by index value:

```
1         Bird, Michael
2         Erving, Isiah
4         Johnson, Julius
0         Jordan, Larry
3         Thomas, Earvin
```

Random Access Example

- Let’s create the data file first:

```
#include <stdio.h>
typedef struct {
        unsigned int day : 5;
        unsigned int month : 4;
        unsigned int year : 11;
} DATA;
typedef struct {
    char ps_name[19], ps_tcno[11];
    DATE ps_birth_date;
} PERSONALSTAT;
//to be continued in the next slide
```

Random Access Example

```
int main(){
    int j; FILE *fp; unsigned int val;
    PERSONALSTAT ps2[4];
    fp=fopen("records.dat", "wb");
    for(j=0; j<4; j++){
        printf("Enter name of the %d-th person:\n", j+1);
        scanf("%s", ps2[j].ps_name);
        printf("Enter tcno of the %d-th person:\n", j+1);
        scanf("%s", ps2[j].ps_tcno);
        printf("Enter the birth-date (day) of the %d-th person:\n", j+1);
        // NOT ALLOWED: scanf("%u", &ps2[j].ps_birth_date.day);
        scanf("%u", &val); ps2[j].ps_birth_date.day=val;
        printf("Enter the birth-date (month) of the %d-th person:\n", j+1);
        scanf("%u", &val); ps2[j].ps_birth_date.month=val;
        printf("Enter the birth-date (year) of the %d-th person:\n", j+1);
        scanf("%u", &val); ps2[j].ps_birth_date.year=val;
        fwrite(&ps2[j], sizeof(PERSONALSTAT), 1, fp);
     }
     fclose(fp); return 0;
}
```

Random Access Example

- Now let’s create and sort the index:

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h> // Header file for qsort()
#include <string.h> // for the strcmp function()
#define MAX_REC_NUM 1000
#define NAME_LEN 19
typedef struct {
        unsigned int day : 5;
        unsigned int month : 4;
        unsigned int year : 11;
} DATE;
typedef struct {
   char ps_name[NAME_LEN], ps_tcno[11];
   DATE ps_birth_date;
} PERSONALSTAT;

// structure definition for the index files for our records:
typedef struct {
   int index;
   char key[NAME_LEN];
} INDEX;
//to be continued in the next slide
```

Random Access Example

```
/* Reads up to max_rec_num records from a file and stores the key field of each record in an index
array. Returns the number of key fields stored. */
int get_records(FILE* data_file, INDEX names_index[], int max_rec_num){
   int offset = 0, counter = 0, k, nm = NAME_LEN;
   char namei[NAME_LEN];

   // get only the name of the 1st PERSON: (the first 19 chars is for name field)
   nm=fread(namei, 1, NAME_LEN, data_file);
   for (k = 0; counter < max_rec_num && nm== NAME_LEN; k++){
      strcpy(names_index[k].key, namei);
      // jump into the beginning of the next person's (next record's) starting point:
      offset += sizeof(PERSONALSTAT);
      fseek(data_file, offset, SEEK_SET);
      counter++;
      // get only the name of the i-th PERSON: (the first 19 chars for each person/record)
      nm=fread(namei, 1, NAME_LEN, data_file);
   }
   return counter;
}
//to be continued in the next slide
```

Random Access Example

```
/* Sort an array of NAMES_INDEX structures by the name field. There are index count elements to be
sorted. Returns a pointer to the sorted array. This function will be required for the qsort function
to provide a means of comparison. */
int compare_func(INDEX *p, INDEX *q ){
   return strcmp( p->key, q->key);
        /* <0: the first character that does not match has a lower value in ptr1 than in ptr2
            0:  the contents of both strings are equal
           >0: the first character that does not match has a greater value in ptr1 than in ptr2 */
}

void sort_index(INDEX names_index[], int index_count) {
   int j;
   int (*pf) (); pf = compare_func;
   // Assign values to the index field of each structure:
   for (j = 0; j < index_count; j++)
      names_index[j].index = j;
   qsort(names_index, index_count, sizeof(INDEX), pf);
}
//to be continued in the next slide
```

Random Access Example

```
/* Print the records in a file in the order * indicated by the index array. */
void print_indexed_records(FILE * data_file, INDEX index[], int index_count ){
  PERSONALSTAT ps;
  int j;
  for (j = 0; j < index_count; j++){
    if (fseek( data_file, sizeof(PERSONALSTAT) * index[j].index, SEEK_SET))
      exit( 1 );
    fread(&ps, 1, sizeof(PERSONALSTAT), data_file);
    printf( "%20s, %u, %u, %u, %12s\n", ps.ps_name, ps.ps_birth_date.day,
                ps.ps_birth_date.month, ps.ps_birth_date.year, ps. ps_tcno);
  }
}
//to be continued in the next slide
```

Random Access Example

```
/* To make this program complete, we need a main() function that calls these other functions. We
have written main() so the filename can be passed as an argument.*/
int main(int argc, char *argv[]){
  FILE *data_file, *index_file; static INDEX index[MAX_REC_NUM]; int num_recs_read; char *filename;
  if (argc < 2) {
    printf( "Error: must enter index filename\n" );
    printf( "Filename: " );   filename=malloc(60);   scanf( "%s", filename );
  }
  else filename = argv[1];

  if ((data_file = fopen( filename, "rb" )) == NULL){
    printf( "Error opening file %s.\n", filename); exit(1);
  }
  num_recs_read = get_records(data_file, index, MAX_REC_NUM ) ;
  printf("num_recs_read: %d\n", num_recs_read);
  sort_index(index, num_recs_read);
  printf("After the sorting\n");
  print_indexed_records (data_file, index, num_recs_read);
  fclose(data_file);
  return 0;
}
```

File Management Functions

- remove(): Deletes a file
  - int		remove (const char* szFileName);
- rename(): Renames a file
  - int		rename (const char* szOldFileName, const char* szNewFileName);
- tmpfile(): Creates a temporary binary file
  - FILE*	tmpfile ();
- tmpnam():
  - char*	tmpnam (char caName[]);
  - Generates a string that can be used as the name of a temporary file. Can return unsafe characters such as \s therefore it should be sanitized.

Splitting a Program & Makefile

Multithread Programming

by Zeyneb YAVUZ and Ahmet ELBİR

Splitting Your C Program

- At least one of the files must have a main() function.
- To use functions from another file,
  - make a .h file with the function prototypes,
  - and use #include to include those .h files within your .c files.
- Be sure no 2 files have functions with the same name in it.
  - The compiler will get confused.
- Similarly, if you use global variables in your program, be sure no two files define the same global variables.

Splitting Your C Program

- If you use global variables, be sure only one of the files defines them, and declare them in your .h as follows:
  - **extern int globalvar;**

REMINDER!!

- When you define a variable, it looks like this:
  - **int globalvar;**
- When you declare a variable, it looks like this:
  - **extern int globalvar;**
- The main difference is that a variable definition creates the variable, while a declaration indicates that the variable is defined elsewhere. A definition implies a declaration.

An EXAMPLE

salute.h (header file contains prototypes):

```
#ifndef __salute_h__
#define __salute_h__
void salute( void );
#endif
```

salute.c (implement the .h in a .c file):

```
#include <stdio.h>
void salute( void ) {
	printf("\n\n HELLO!!! \n\n");
}
```

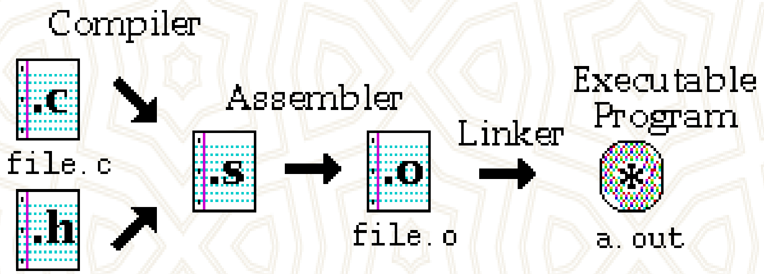
main.c :

```
#include <stdio.h>
#include "salute.h"
int main ( void ) {
	salute();
	return 0;
}
```

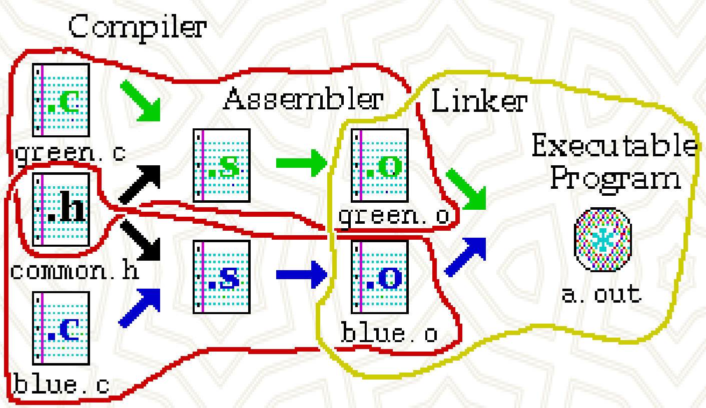
```
cc -c salute.c #this will give you salute.o
cc -c main.c #this will give you main.o
cc -o main main.o salute.o #this will create the main executable
```

Compiling with Several Files

- The command to perform this task is simply
  - **gcc file.c**
- There are 3 steps to obtain the final executable program
  - **Compiler stage**
  - **Assembler stage**
  - **Linker stage**
- Side-note: pre-processor takes place before the compiler stage



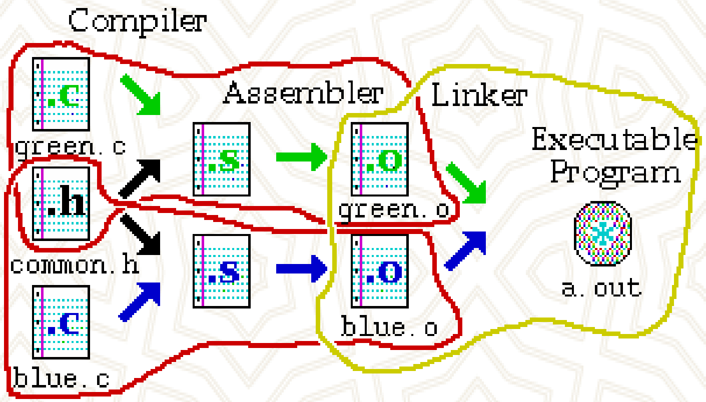
Compiling with Several Files



- You can use the -c option with gcc to create the corresponding object (.o) file from a .c file.
  - **gcc -c green.c**
- will not produce an a.out file, but the compiler will stop after the assembler stage, leaving you with a green.o file.

Compiling with Several Files

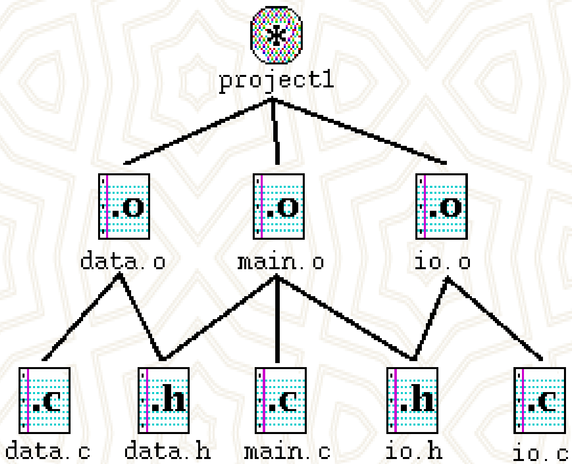
- The three different tasks required to produce the executable program are as follows:
- Compile green.o:
  - **gcc -c green.c**
- Compile blue.o:
  - **gcc -c blue.c**
- Link the parts together:
  - **gcc green.o blue.o** **//** will create an exe with the name ***a.out*** (by default)
  - **If you want to give another name to your exe, use –o flag**



The Make Command

- helps you to manage large programs or groups of programs
- keeps track of which portions of the entire program have been changed
- compiles only those parts of the program which have changed since the last compile.

How does make do it?



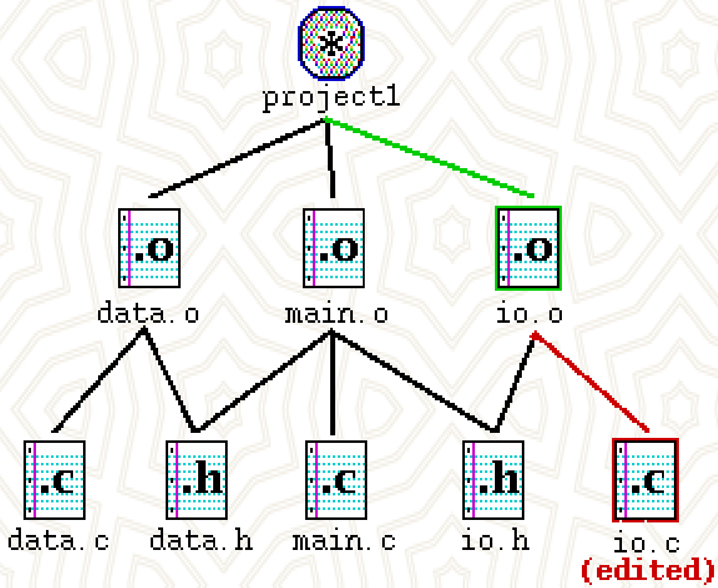
Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

The make program gets its dependency "graph" from a text file called makefile or Makefile, which resides in the same directory as the source files

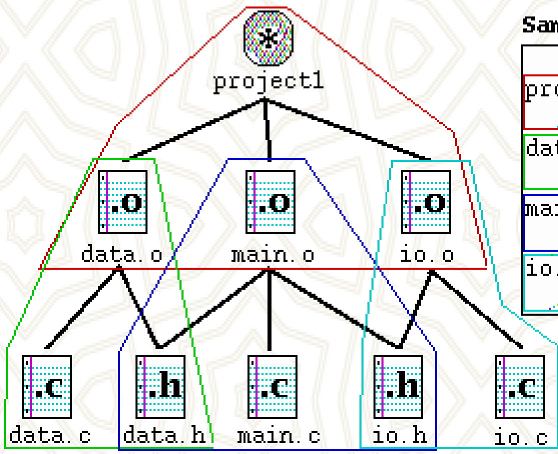
make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, it runs the compiler accordingly.

How Dependency Works



- Case : while you are testing the program, you realize that one function in io.c has a bug in it.
- You edit io.c to fix the bug.
- notice that io.o needs to be updated because io.c has changed. Similarly, because io.o has changed, project1 needs to be updated as well.

Translating The Dependency Graph



Sample Makefile

```
project1: data.o main.o io.o
	cc data.o main.o io.o -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

- Each dependency shown in the graph is circled with a corresponding color in the Makefile, and each uses the following format:
- target : source file(s)
  - command (must be preceded by a tab)

Listing Dependencies

- Note that in the Makefile shown on the right, the .h files are listed, but there are no references in their corresponding commands.
- This is because the .h files are referred within the corresponding .c files through the #include "file.h".
- If you do not explicitly include these in your Makefile, your program will not be updated if you make a change to your header (.h) files.

Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Using the Makefile with make

- Once you have created your Makefile and your corresponding source files, you are ready to use make.
- If you have named your Makefile either Makefile or makefile, *make* command will recognize it.
- If you do not wish to call your Makefile one of these names, you can use *make -f mymakefile*.

Macros in make

- The make program allows you to use macros, which are similar to variables, to store names of files. The format is:
  - OBJECTS = data.o io.o main.o
- Whenever you want to have make expand these macros out when it runs, type the corresponding string $(OBJECTS)

Here is our sample *Makefile* again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o: data.c data.h
        cc -c data.c
main.o: data.h io.h main.c
        cc -c main.c
io.o: io.h io.c
        cc -c io.c
```

Special macros, which are used by the make program

- In addition to those macros which you can create yourself, there are a few macros, which are used internally by the make program. Here are some of those, listed below:

CC	Contains the current C compiler. Defaults to cc.
CFLAGS	Special options which are added to the built-in C rule.
$@	Full name of the current target.
$?	A list of files for current dependency which are out-of-date.
$<	The source file of the current (single) dependency.

References & more Reading

- References
  - http://www.eng.hawaii.edu/Tutor/Make/index.html
- More reading on make command
  - http://www.cs.duke.edu/~ola/courses/programming/Makefiles/Makefiles.html
  - http://www.hsrl.rutgers.edu/ug/make_help.html
  - http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/make.htm

Multithread Programming

Multithread Programming

by Zeyneb YAVUZ and Ahmet ELBİR

Outline

- What is a thread?
- Why do we need threads?
- Difference between threads and processes
- Problems with Threads
- Identifying a thread
- Creating a thread
- Terminating a thread
- Examples

Threading in C

- Threads/ Processes are the mechanism by which you can run multiple code segments at a time,
- A thread of execution is the smallest sequence of program instructions that can be managed independently by a scheduler
- A process can have multiple threads of execution.
- Threads appear to run concurrently; the kernel schedules them asynchronously, interrupting each thread from time to time to give others chance to execute.
- This asynchronous execution brings in the capability of each thread handling a particular work or service independently.
- Multiple threads running in a process handle their services which overall constitutes the complete capability of the process.

Use of Threads - Examples

- Graphical User Interfaces (GUIs)
  - The GUI is usually put on a separate thread from the “app engine”
  - ⍰ GUI remains responsive even if app blocks for processing
- ⍰ Web Browser Tabs
  - Each tab is managed by a separate thread for rendering
  - Web pages render “simultaneously” (e.g. while one page is printed out, another page can be downloaded concurrently)
  - Note: Google Chrome actually uses a separate process per tab

Why Threads are Required?

- Why do we need multiple threads in a process? Why can't a process with only one (default) main thread be used in every situation.
- To answer this lets consider an example:
  - Suppose there is a process, that receiving real time inputs and corresponding to each input it has to produce a certain output.
  - If the process does not involve multiple threads, then the whole processing in the process becomes synchronous.
  - This means that the process takes an input, processes it, and produces an output.

Why Threads are Required -2

- The limitation in the above design is that the process cannot accept an input until its done processing the earlier one.
- In case processing an input takes longer than expected, then accepting further inputs goes on hold.
- We could solve the above example with a socket server process that can accept input connection, process them and provide the socket client with output.
- While processing any input, if the server process takes more than expected time and in the meantime another input (connection request) comes to the socket server then the server process would not be able to accept the new input connection as its already stuck in processing the old input connection.
- This may lead to a connection time out at the socket client which is not at all desired.
- This shows that synchronous model of execution cannot be applied everywhere hence the asynchronous model of execution would be required, which is implemented by using threads.

Difference between threads and processes

- Processes do not share their memory space, while threads executing under same process share the memory space.
- Processes have independent open file descriptors, while threads have shared open file descriptors
- Processes execute independent of each other and the synchronization between processes is taken care by kernel only; on the other hand, thread synchronization has to be taken care by the process under which the threads are executing
- Context switching between threads is fast as compared to context switching between processes
- The interaction between 2 processes is achieved only through the standard inter-process communication, while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

Problems with Threads -1

- Many operating system does not implement threads as processes rather they see threads as part of parent process.
- What would happen if a thread execs a new binary (exe)?
- This scenario may have dangerous consequences e.g. the whole parent process could get replaced with the address space of the newly exec'd binary.
- So, an exec from any of the thread would stop all the threads in the parent process. This is not at all desired.
- This problem is a design issue and design for applications should be done in a way that least problems of this kind arise.
- Debugging with threads is difficult.
- Too many threads may reduce the performance.

Problems with Threads -2

- Another problem that may arise is the concurrency:
  - Since threads share all the segments (except the stack) and can be preempted at any stage by the scheduler before any global variable or data structure that can be left in inconsistent state by preemption of one thread could cause severe problems when the next high priority thread executes the same function & uses the same variables or data structures.
  - For the above problem: using locking mechanisms programmer can lock a chunk of code inside a function so, when a context switch* happens, next thread is not able to execute the same code until the locked code block inside the function is unlocked by the previous thread.
- *context switch: is the process of storing the state of a thread, so, it can be restored and execution resumed from the same point later. This allows multiple threads to share a single CPU.

Identifying a thread

- Each thread is identified by an ID, which is known as Thread ID
- Thread ID is quite different from Process ID.
- A Thread ID is unique in the context of current process, while a Process ID is unique across the system.
- A process ID is an integer value but the thread ID is not necessarily an integer value. It could be a structure and represented by type pthread_t.
- A process ID can be printed very easily while a thread ID is not easy to print.
- The header file, which needs to be included to access thread functions and pthread_t type, is: #include<pthread.h>

Creating a Thread (pthread_create)

- pthread_create function in pthread.h file, is used to create a thread.
- The syntax and parameters details are given as follows:

  *int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine) (void *), void *arg);*

- pthread_t *thread: It is the pointer to a pthread_t variable which is used to store thread id of new created thread.
- const pthread_attr_t *attr: It is the pointer to a thread attribute object which is used to set thread attributes, NULL can be used to create a thread with default arguments.
- void *(*start_routine) (void *): It is the pointer to a thread function; this function contains the code segment which is executed by the thread.
- void *arg: It is the thread functions argument of the type void*, you can pass what is necessary for the function using this parameter.
- int (return type): If thread created successfully, return value will be 0 otherwise pthread_create will return an error number of type int.

Example

```c
#include <stdio.h>
#include <pthread.h>
/*thread (worker) function definition*/
void* threadFunction(void* args){
    while(1)
        printf("I am threadFunction.\n");
}
int main(){   /*creating a thread id in the main function (main thread) */
   pthread_t id;
   int ret;
   /*creating thread*/
   ret=pthread_create(&id, NULL, &threadFunction, NULL);
   if(ret==0)
        printf("Thread is created successfully.\n");
   else{
        printf("Thread is not created.\n");
        return 0; /*return from main*/
    }
    while(1)
        printf("I am main function.\n");
    return 0;
}
```

* https://www.thegeekstuff.com/2012/03/linux-threads-intro/

How to compile & execute?

- To compile:

$ gcc <file-name.c> -o <output-file-name> -lpthread

- To run:

$ ./<output-file-name>

Example

```c
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];

void* doSomeThing(void *arg){
   unsigned long i = 0;
   pthread_t id = pthread_self();

   if(pthread_equal(id,tid[0]))
      printf("\n First thread
   processing\n");
   else
      printf("\n Second thread
   processing\n");
   for(i=0; i<(0xFFFFFFFF);i++);
   return NULL;
}
```

```c
int main(void){
   int i = 0;
   int err;

   while(i < 2) {
      err = pthread_create(&(tid[i]),
               NULL, &doSomeThing, NULL);
      if (err != 0)
         printf("\ncan't create thread: %s",
               strerror(err));
      else
         printf("\n Thread created
                              successfully\n");
      i++;
   }
   sleep(5); // this is important!!
   return 0;
}
```

*https://www.thegeekstuff.com/2012/04/create-threads-in-linux/

➢ int pthread_equal(pthread_t tid1, pthread_t tid2); takes two thread IDs and returns a non-0 value if both IDs are equal, else it returns 0.

➢ pthread_t pthread_self(void); // It is used by a thread for printing its own thread ID.

- Without sleep() function:

```
$ ./threads
Thread created successfully
First thread processing
Thread created successfully
```

- With sleep() function:

```
$ ./threads
Thread created successfully
First thread processing
Thread created successfully
Second thread processing
```

- Without the sleep* function, we did not see the message of "Second thread processing":
- Because just before the second thread is about to be scheduled, the parent thread, from which the two threads were created, completed its execution.
- So that the default thread in which the main() function was running got completed and hence the process terminated as main() returned.
- * sleep: sleep for the specified number of sec. Defined in <unistd.h>

Terminating a thread

- (See the example code) In the code:
- We created two threads using pthread_create()
- The start function for both the threads is same: doSomeThing()
- The threads exit from the start function using the pthread_exit() function with a return value (this is called inside the doSomeThing() function).
- In the main function, after the threads are created, the pthread_join() is called to wait for each thread to complete.
- Once both the threads are complete, their return value is accessed by the second argument in the pthread_join() call.

```c
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int ret1,ret2;
void* doSomeThing(void *arg) {
   unsigned long i = 0;
   pthread_t id = pthread_self();
   for(i=0; i<(0xFFFFFFFF);i++);
   if(pthread_equal(id,tid[0])) {
      printf("\n 1st thread processing done\n");
      ret1  = 100;
      pthread_exit(&ret1);
   }
   else   {
      printf("\n 2nd thread processing done\n");
      ret2  = 200;
      pthread_exit(&ret2);
   }
   return NULL;
}
```

```c
int main(void)
{
    int i = 0, err;
    int *ptr[2];

    while(i < 2) {
        err = pthread_create(&(tid[i]), NULL,
                &doSomeThing, NULL);
        if (err != 0)
            printf("\ncan't create thread :[%s]",
                strerror(err));
        else
            printf("\n Thread created
 successfully\n");
        i++;
    }
    pthread_join(tid[0], (void**)&(ptr[0]));
    pthread_join(tid[1], (void**)&(ptr[1]));
    printf("\n return value from first thread is
            %d\n", *ptr[0]);
    printf("\n return value from second thread is
            %d\n", *ptr[1]);
    return 0;
}
```

https://www.thegeekstuff.com/2012/04/terminate-c-thread/