
Course Material

on

Introduction to the C Programming
Language

Table of Contents

- [Introduction](#)
- [C Program Structure](#)
- [Variables, Expressions, & Operators](#)
- [Input and Output](#)
- [Program Looping](#)
- [Decision Making Statements](#)
- [Array Variables](#)
- [Strings](#)
- [Math Library Functions](#)
- [User-defined Functions](#)
- [Formatted Input and Output](#)
- [Pointers](#)
- [Structures](#)
- [Unions](#)
- [File Input and Output](#)
- [Dynamic Memory Allocation](#)
- [Command Line Arguments](#)
- [Operator Precedence Table](#)

Introduction

- [Why Learn C?](#)

- Compact, fast, and powerful
- “Mid-level” Language
- Standard for program development (**wide acceptance**)
- It is everywhere! (**portable**)
- Supports modular programming style
- Useful for all applications
- C is the native language of UNIX
- Easy to interface with system devices/assembly routines
- C is terse

C Program Structure

- [Canonical First Program](#)
- [Header Files](#)
- [Names in C](#)
- [Comments](#)
- [Symbolic Constants](#)

Canonical First Program

- The following program is written in the C programming language:

```
#include <stdio.h>
main()
{
    /* My first program */
    printf("Hello World! \n");
}
```

- **C is case sensitive.** All commands in C must be lowercase.
- **C has a free-form line structure.** **End of each statement must be marked with a semicolon.** Multiple statements can be on the same line. *White space* is ignored. Statements can continue over many lines.

Canonical First Program Continued

```
#include <stdio.h>
main()
{
    /* My first program */
    printf("Hello World! \n");
}
```

- The C program starting point is identified by the word `main()`.
- This informs the computer as to where the program actually starts. The parentheses that follow the keyword `main` indicate that there are no arguments supplied to this program (this will be examined later on).
- **The two braces, { and }, signify the begin and end segments of the program.** In general, braces are used throughout C to enclose a block of statements to be treated as a unit. ***COMMON ERROR: unbalanced number of open and close curly brackets!***

More on the Canonical First Program

```
#include <stdio.h>
main()
{
    /* My first program */
    printf("Hello World! \n");
}
```

- The purpose of the statement `#include <stdio.h>` is to allow the use of the `printf` statement to provide program output. For each function built into the language, an associated *header* file must be included. Text to be displayed by `printf()` must be enclosed in double quotes. The program only has the one `printf()` statement.
- `printf()` is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes `" "`, it is printed without modification. There are some exceptions however. This has to do with the `\` and `%` characters. These characters are modifiers, and for the present the `\` followed by the `n` character represents a *newline* character.

Canonical First Program Output & Comments

- Thus the program prints

Hello World!

- And the cursor is set to the beginning of the next line. As we shall see later on, what follows the `\` character will determine what is printed (i.e., a tab, clear screen, clear line, etc.)

`/* My first program */`

- **Comments** can be inserted into C programs by bracketing text with the `/*` and `*/` delimiters. As will be discussed later, comments are useful for a variety of reasons. Primarily they serve as *internal documentation* for program structure and functionality.

Header Files

- **Header files contain definitions of functions and variables** which can be incorporated into any C program by using the pre-processor **#include** statement. Standard header files are provided with each compiler, and cover a range of areas: string handling, mathematics, data conversion, printing and reading of variables, etc.
- To use any of the standard functions, the appropriate header file should be included. This is done at the beginning of the C source file. For example, to use the function **printf()** in a program, the line

```
#include <stdio.h>
```

- should be at the beginning of the source file, because the declaration for **printf()** is found in the file **stdio.h**. All header files have the extension **.h** and generally reside in the **/usr/include** subdirectory.

```
#include <string.h>
```

```
#include <math.h>
```

```
#include "mylib.h"
```

- The use of angle brackets **<>** informs the compiler to search the compiler's include directories for the specified file. The use of the double quotes **" "** around the filename informs the compiler to start the search in the current directory for the specified file.

Names in C

- Identifiers in C **must begin with a character or underscore**, and may be followed by any combination of characters, underscores, or the digits 0-9.

<code>summary</code>	<code>exit_flag</code>	<code>i</code>
<code>Jerry7</code>	<code>Number_of_moves</code>	<code>_id</code>

- You should ensure that you use **meaningful (but short) names** for your identifiers. The reasons for this are to make the program easier to read and self-documenting. Example:

```
distance = speed * time;
```

- Some users choose to adopt the **convention** that variable names are all lower case while symbolic names for constants are all upper case.
- Keywords** are reserved identifiers that have strict meaning to the C compiler. C only has 29 keywords. Example keywords are:

```
if, else, char, int, while
```

Comments

- The addition of comments inside programs is desirable. These may be added to C programs by enclosing them as follows,

`/*`

`Computational Kernel: In this section of code we implement the
Runge-Kutta algorithm for the numerical solution of the
differential Einstein Equations.`

`*/`

- Note that the `/*` opens the comment field and the `*/` closes the comment field. Comments may span multiple lines. Comments may not be nested one inside the another.

`/* this is a comment. /* this comment is inside */ wrong */`

- In the above example, the first occurrence of `*/` closes the comment statement for the entire line, meaning that the text `wrong` is interpreted as a C statement or variable, and in this example, generates an error.

Why use comments?

- Documentation of variables and functions and their usage
- Explaining difficult sections of code
- Describes the program, author, date, modification changes, revisions...

Best programmers comment as they write the code, not after the fact.

Symbolic Constants

- Names given to values that cannot be changed. Implemented with the `#define` preprocessor directive.

```
#define N 3000
#define FALSE 0
#define PI 3.14159
#define FIGURE "triangle"
```

- Note that preprocessor statements begin with a `#` symbol, and are NOT terminated by a semicolon. Traditionally, preprocessor statements are listed at the beginning of the source file.
- Preprocessor statements are handled by the compiler (or preprocessor) before the program is actually compiled. All `#` statements are processed first, and the **symbols** (like `N`) which occur in the C program **are replaced by their value** (like `3000`). Once this substitution has taken place by the preprocessor, the program is then compiled.
- In general, preprocessor constants are written in UPPERCASE. This acts as a form of internal documentation to **enhance program readability and reuse**.
- In the program itself, values cannot be assigned to symbolic constants.

Use of Symbolic Constants

- Consider the following program which defines a constant called **TAXRATE**.

```
#include <stdio.h>
#define TAXRATE 0.10
main ()    {
    float balance;
    float tax;
    balance = 72.10;
    tax = balance * TAXRATE;
    printf("The tax on %.2f is %.2f\n",balance, tax);
}
```

The tax on 72.10 is 7.21

- The whole point of using **#define** in your programs is to make them easier to read and modify. Considering the above program as an example, what changes would you need to make if the **TAXRATE** was changed to 20%?

Use of Symbolic Constants

- Obviously, the answer is one, where the `#define` statement which declares the symbolic constant and its value occurs. You would change it to read
`#define TAXRATE 0.20`
- Without the use of symbolic constants, you would hard code the value `0.20` in your program, and this might occur several times (or tens of times).

Variables, Expressions, and Operators

- [Declaring Variables](#)
- [Basic Format](#)
- [Basic Data Types: Integer](#)
- [Basic Data Types: Float](#)
- [Basic Data Types: Double](#)
- [Basic Data Types: Character](#)
- [Expressions and Statements](#)
- [Assignment Operator](#)
- [Assignment Operator Evaluation](#)
- [Initializing Variables](#)
- [Initializing Variables Example](#)
- [Arithmetic Operators](#)
- [Increment/Decrement Operators](#)
- [Prefix versus Postfix](#)
- [Advanced Assignment Operators](#)
- [Precedence & Associativity of Operators](#)
- [Precedence & Associativity of Operators Examples](#)
- [The **int** Data Type](#)
- [The **float** and **double** Data Types](#)
- [The **char** Data Type](#)
- [ASCII Character Set](#)
- [Automatic Type Conversion](#)
- [Automatic Type Conversion with Assignment Operator](#)
- [Type Casting](#)

Declaring Variables

- A variable is a **named memory location** in which data of a certain type can be stored. The *contents of a variable can change*, thus the name. User defined variables must be declared before they can be used in a program. It is during the declaration phase that the actual memory for the variable is reserved. **All variables in C must be declared before use.**
- Get into the habit of declaring variables using lowercase characters. Remember that C is case sensitive, so even though the two variables listed below have the same name, they are considered different variables in C.

`sum`

`Sum`

- The declaration of variables is done after the opening brace of main().

```
main()    {  
    int sum;
```

- It is possible to declare variables elsewhere in a program, but let's start simply and then get into variations later on.

Basic Format

- The basic format for declaring variables is

```
data_type var, var, ...;
```

- where *data_type* is one of the four basic types, an integer, character, float, or double type. Examples are

```
int i,j,k;
```

```
float length,height;
```

```
char midinit;
```

Basic Data Types: INTEGER

- **INTEGER:** These are whole numbers, both positive and negative. Unsigned integers(positive values only) are also supported. In addition, there are short and long integers. These specialized integer types will be discussed later.
- The keyword used to define integers is

`int`

- An example of an integer value is 32. An example of declaring an integer variable called `age` is

`int age;`

Basic Data Types: FLOAT

- **FLOATING POINT:** These are numbers which contain fractional parts, both positive and negative, and can be written in scientific notation.
- The keyword used to define float variables is

`float`

- Typical floating point values are 1.73 and 1.932e5 (1.932×10^5). An example of declaring a float variable called `x` is

`float x;`

Basic Data Types: DOUBLE

- **DOUBLE:** These are floating point numbers, both positive and negative, which have a higher precision than float variables.
- The keyword used to define double variables is

`double`

- An example of declaring a double variable called `voltage` is

`double voltage;`

Basic Data Types: CHAR

- **CHARACTER:** These are single characters.
- The keyword used to define character variables is

`char`

- Typical character values might be the letter A, the character 5, the symbol “, etc. An example of declaring a character variable called `letter` is

```
char letter;
```

Expressions and Statements

- An **expression** in C is some **combination of constants, variables, operators and function calls**. Sample expressions are:

```
a + b  
3.0*x - 9.66553  
tan(angle)
```

- Most expressions have a value based on their contents.
- A **statement** in C is just an **expression terminated with a semicolon**. For example:

```
sum = x + y + z;  
printf("Go Buckeyes!");
```


The Assignment Operator

- In C, the assignment operator is the equal sign `=` and is used to give a variable the value of an expression. For example:

```
i=0;  
x=34.8;  
sum=a+b;  
slope=tan(rise/run);  
midinit='J';  
j=j+3;
```

- When used in this manner, the equal sign should be read as “gets”. Note that when assigning a character value the character should be **enclosed in single quotes**.

The Assignment Operator Evaluation

- In the assignment statement

`a=7;`

- two things actually occur. **The integer variable a gets the value of 7**, and **the expression a=7 evaluates to 7**. This allows a shorthand for multiple assignments of the same value to several variables in a single statement. Such as

`x=y=z=13.0;`

Initializing Variables

- C Variables may be initialized with a value when they are declared. Consider the following declaration, which declares an integer variable `count` which is initialized to `10`.

```
int count = 10;
```

- In general, the user should not assume that variables are initialized to some default value “automatically” by the compiler. Programmers must ensure that variables have proper values before they are used in expressions.

Initializing Variables Example

- The following example illustrates the two methods for variable initialization:

```
#include <stdio.h>
main () {
    int sum=33;
    float money=44.12;
    char letter;
    double pressure;
    letter='E'; /* assign character value */
    pressure=2.01e-10; /*assign double value */
    printf("value of sum is %d\n",sum);
    printf("value of money is %f\n",money);
    printf("value of letter is %c\n",letter);
    printf("value of pressure is %e\n",pressure);
}
```

- which produces the following output:

```
value of sum is 33
value of money is 44.119999
value of letter is E
value of pressure is 2.010000e-10
```

Arithmetic Operators

- The primary arithmetic operators and their corresponding symbols in C are:

Negation	-	Modulus	%
Multiplication	*	Addition	+
Division	/	Subtraction	-

- When the / operator is used to perform **integer division** the resulting integer is obtained by discarding (or truncating) the fractional part of the actual floating point value. For example:

$$1/2 \rightarrow 0$$

$$3/2 \rightarrow 1$$

- The **modulus operator** % only works with integer operands. The expression **a%b** is read as “a modulus b” and evaluates to the remainder obtained after dividing **a** by **b**. For example

$$7 \% 2 \rightarrow 1$$

$$12 \% 3 \rightarrow 0$$

Increment/Decrement Operators

- In C, specialized operators have been set aside for the incrementing and decrementing of integer variables. The increment and decrement operators are `++` and `--` respectively. These operators allow a form of shorthand in C:

`++i;` is equivalent to `i=i+1;`

`--i;` is equivalent to `i=i-1;`

- The above example shows the **prefix form** of the increment/decrement operators. They can also be used in **postfix form**, as follows

`i++;` is equivalent to `i=i+1;`

`i--;` is equivalent to `i=i-1;`

Prefix versus Postfix

- The difference between prefix and postfix forms shows up when the operators are used as part of a larger expression.
 - If **++k** is used in an expression, **k** is incremented *before* the expression is evaluated.
 - If **k++** is used in an expression, **k** is incremented *after* the expression is evaluated.

- Assume that the integer variables **m** and **n** have been initialized to zero. Then in the following statement

a=++m + ++n; m → 1, n → 1, then a → 2

- whereas in this form of the statement

a=m++ + n++; a → 0 then m → 1, n → 1

Advanced Assignment Operators

- A further example of C shorthand are operators which combine an arithmetic operation and a assignment together in one form. For example, the following statement

`k=k+5;` can be written as `k += 5;`

- The general syntax is

`variable = variable op expression;`

- can alternatively be written as

`variable op= expression;`

- common forms are:

`+=` `-=` `*=` `/=` `%=`

- Examples:

`j=j*(3+x);` `j *= 3+x;`
`a=a/(s-5);` `a /= s-5;`

Precedence & Associativity of Operators

- The precedence of operators determines the order in which operations are performed in an expression. Operators with higher precedence are employed first. If two operators in an expression have the same precedence, associativity determines the direction in which the expression will be evaluated.
- C has a built-in **operator hierarchy** to determine the precedence of operators. Operators higher up in the following diagram have higher precedence. The associativity is also shown.

-	++	--	R	→	L
*	/	%	L		R
+	-		L		R
=			R		L

Precedence & Associativity of Operators Examples

- This is how the following expression is evaluated

```
1 + 2 * 3 - 4
1 + 6 - 4
7 - 4
3
```

- The programmer can **use parentheses to override the hierarchy** and force a desired order of evaluation. **Expressions enclosed in parentheses are evaluated first.** For example:

```
(1 + 2) * (3 - 4)
3 * -1
-3
```

The `int` Data Type

- A typical `int` variable is in the range $\pm 32,767$. This value differs from computer to computer and is thus **machine-dependent**. It is possible in C to specify that an integer be stored in more memory locations thereby increasing its effective range and allowing very large integers to be stored. This is accomplished by declaring the integer variable to have type `long int`.

`long int national_debt;`

- `long int` variables typically have a range of $\pm 2,147,483,648$.
- There are also **short int** variables which may or may not have a smaller range than normal `int` variables. All that C guarantees is that a **short int** will not take up more bytes than `int`.
- There are **unsigned** versions of all three types of integers. Negative integers cannot be assigned to **unsigned** integers, only a range of positive values. For example

`unsigned int salary;`

- typically has a range of 0 to 65,535.

The float and double Data Types

- As with integers the different floating point types available in C correspond to different ranges of values that can be represented. More importantly, though, the number of bytes used to represent a real value determines the precision to which the real value is represented. The more bytes used the higher the number of decimal places of accuracy in the stored value. The actual ranges and accuracy are machine-dependent.
- The three C floating point types are:

`float`

`double`

`long double`

- In general, the accuracy of the stored real values increases as you move down the list.

The char Data Type

- Variables of type char take up exactly one byte in memory and are used to store printable and non-printable characters. The ASCII code is used to associate each character with an integer (see next page). For example the ASCII code associates the character 'm' with the integer 109. Internally, **C treats character variables as integers.**

ASCII Character Set

Ctrl	Decimal	Code	Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
^@	0	NUL	32	sp	32	sp	64	@	96	`
^A	1	SOH	33	!	33	!	65	A	97	a
^B	2	STX	34	"	34	"	66	B	98	b
^C	3	ETX	35	#	35	#	67	C	99	c
^D	4	EOT	36	\$	36	\$	68	D	100	d
^E	5	ENQ	37	%	37	%	69	E	101	e
^F	6	ACK	38	&	38	&	70	F	102	f
^G	7	BEL	39		39		71	G	103	g
^H	8	BS	40	(40	(72	H	104	h
^I	9	HT	41)	41)	73	I	105	i
^J	10	LF	42	*	42	*	74	J	106	j
^K	11	VT	43	+	43	+	75	K	107	k
^L	12	FF	44	,	44	,	76	L	108	l
^M	13	CR	45	-	45	-	77	M	109	m
^N	14	SOH	46	.	46	.	78	N	110	n
^O	15	ST	47	/	47	/	79	O	111	o
^P	16	SLE	48	0	48	0	80	P	112	p
^Q	17	CS1	49	1	49	1	81	Q	113	q
^R	18	DC2	50	2	50	2	82	R	114	r
^S	19	DC3	51	3	51	3	83	S	115	s
^T	20	DC4	52	4	52	4	84	T	116	t
^U	21	NAK	53	5	53	5	85	U	117	u
^V	22	SYN	54	6	54	6	86	V	118	v
^W	23	ETB	55	7	55	7	87	W	119	w
^X	24	CAN	56	8	56	8	88	X	120	x
^Y	25	EM	57	9	57	9	89	Y	121	y
^Z	26	SIB	58	:	58	:	90	Z	122	z
^[27	ESC	59	;	59	;	91	[123	{
^\[28	FS	60	<	60	<	92	\	124	
^]	29	GS	61	=	61	=	93]	125	}
^^	30	RS	62	>	62	>	94	^	126	~
^_	31	US	63	?	63	?	95	_	127	DEL

Automatic Type Conversion

- How does C evaluate and type expressions that contain a **mixture of different data types**? For example, if **x** is a double and **i** an integer, what is the type of the expression

x+i

- In this case, **i** will be converted to type double and the expression will evaluate as a double. NOTE: the value of **i** stored in **memory is unchanged**. A temporary copy of **i** is converted to a double and used in the expression evaluation.
- This automatic conversion takes place in two steps. First, all floats are converted to double and all characters and shorts are converted to ints. In the second step “lower” types are promoted to “higher” types. The expression itself will have the type of its highest operand. The **type hierarchy** is as follows

long double
double
unsigned long
long
unsigned
int

Automatic Type Conversion with Assignment Operator

- Automatic conversion even takes place if the operator is the assignment operator. This creates a method of type conversion. For example, if **x** is double and **i** an integer, then

x=i;

- **i** is promoted to a double and resulting value given to **x**
- On the other hand say we have the following expression:

i=x;

- A conversion occurs, but result is machine-dependent

Type Casting

- Programmers can override automatic type conversion and explicitly cast variables to be of a certain type when used in an expression. For example,

```
(double) i
```

- will force `i` to be of type double. The general syntax is

```
(type) expression
```

- Some examples,

```
(char) 3 + 'A'  
x = (float) 77;  
(double) k * 57
```

Input and Output

- [Basic Output](#)
- [printf Function](#)
- [Format Specifiers Table](#)
- [Common Special Characters for Cursor Control](#)
- [Basic Output Examples](#)
- [Basic Input](#)
- [Basic Input Example](#)

Basic Output

- Now, let us look more closely at the `printf()` statement. In a previous program, we saw this example

```
printf("value of sum is %d\n",sum);
```

- which produced this output:

```
value of sum is 33
```

- The first argument of the `printf` function is called the **control string**. When the `printf` is executed, it starts printing the text in the control string until it encounters a `%` character. The `%` sign is a special character in C and marks the beginning of a **format specifier**. A format specifier controls how the value of a variable will be displayed on the screen. When a format specifier is found, `printf` looks up the next argument (in this case `sum`), displays its value and continues on. The `d` character that follows the `%` indicates that a (d)ecimal integer will be displayed. At the end of the control statement, `printf` reads the special character `\n` which indicates print the new line character.

printf Function

- General form of `printf` function

`printf(control string, argument list);`

- where the `control string` consists of 1) literal text to be displayed, 2) format specifiers, and 3) special characters. The arguments can be variables, constants, expressions, or function calls -- anything that produces a value which can be displayed. **Number of arguments must match the number of format identifiers**. Unpredictable results if argument type does not “match” the identifier.

Format Specifiers Table

- The following table show what format specifiers should be used with what data types:

<i>Specifier</i>	<i>Type</i>
%c	character
%d	decimal integer
%o	octal integer (leading 0)
%x	hexadecimal integer (leading 0x)
%u	unsigned decimal integer
%ld	long int
%f	floating point
%lf	double or long double
%e	exponential floating point
%s	character string

Common Special Characters for Cursor Control

- Some common special characters for cursor control are:

<code>\n</code>	newline
<code>\t</code>	tab
<code>\r</code>	carriage return
<code>\f</code>	form feed
<code>\v</code>	vertical tab
<code>\b</code>	backspace
<code>\"</code>	Double quote (\ acts as an “escape” mark)
<code>\nnn</code>	octal character value

Basic Output Examples

<code>printf("ABC");</code>	ABC (cursor after the C)
<code>printf("%d\n",5);</code>	5 (cursor at start of next line)
<code>printf("%c %c %c",'A','B','C');</code>	A B C
<code>printf("From sea ");</code> <code>printf("to shining ");</code> <code>printf ("C");</code>	From sea to shining C
<code>printf("From sea \n");</code> <code>printf("to shining \n");</code> <code>printf ("C");</code>	From sea to shining C
<code>leg1=200.3; leg2=357.4;</code> <code>printf("It was %f</code> <code>miles",leg1+leg2);</code>	It was 557.700012 miles
<code>num1=10; num2=33;</code> <code>printf("%d\t%d\n",num1,num2);</code>	10 33
<code>big=11e+23;</code> <code>printf("%e \n",big);</code> <code>printf("%c \n",'?');</code>	1.100000e+24 ?
<code>printf("%d \n",'?');</code>	63
<code>printf("\007 That was a beep\n");</code>	try it yourself

Basic Input

- There is a function in C which allows the programmer to accept input from a keyboard. The following program illustrates the use of this function.

```
#include <stdio.h>
main()    {
    int pin;
    printf("Please type in your PIN\n");
    scanf("%d",&pin);
    printf("Your access code is %d\n",pin);}
```

- What happens in this program? An integer called **pin** is defined. A prompt to enter in a number is then printed with the first **printf** statement. The **scanf** routine, which accepts the response, has a **control string** and an **address list**. In the control string, the format specifier **%d** shows what data type is expected. The **&pin** argument specifies the memory location of the variable the input will be placed in. After the **scanf** routine completes, the variable **pin** will be initialized with the input integer. This is confirmed with the second **printf** statement. The **& character** has a very special meaning in C. It **is the address operator**. (Much more with & when we get to pointers...)

Basic Input Example

```
#include <stdio.h>
main()    {
    int pin;
    printf("Please type in your PIN\n");
    scanf("%d",&pin);
    printf("Your access code is %d\n",pin);}
```

- A session using the above code would look like this

```
Please type your PIN
4589
Your access code is 4589
```

- The format identifier used for a specific C data type is the same as for the **printf** statement, with one exception. If you are inputting values for a **double** variable, use the **%lf** format identifier.
- White space is skipped over in the input stream (including carriage return) except for character input. **A blank is valid character input.**

Program Looping

- [Introduction to Program Looping](#)
- [Relational Operators](#)
- [Relational Operators Table](#)
- [for Loop](#)
- [for Loop Example](#)
- [for Loop Diagram](#)
- [General Comments about for Loop](#)
- [General Comments about for Loop Continued](#)
- [while Loop](#)
- [while Loop Example](#)
- [do while Loop](#)
- [do while Loop Example](#)
- [do while Loop Example: Error Checking](#)

Introduction to Program Looping

- Program looping is often desirable in coding in any language to have the ability to repeat a block of statements a number of times. In C, there are statements that allow iteration of this type. Specifically, there are two classes of program loops -- unconditional and conditional. An **unconditional loop** is repeated a set number of times. In a **conditional loop** the iterations are halted when a certain condition is true. Thus the actual number of iterations performed can vary each time the loop is executed.

Relational Operators

- Our first use of these operators will be to set up the condition required to control a conditional loop. Relational operators allow the **comparison of two expressions**. Such as

a < 4

- which reads **a “less than” 4**. If **a** is less than **4**, this expression will evaluate to TRUE. If not it will evaluate to FALSE.
- Exactly what does it mean to say an expression is TRUE or FALSE? C uses the following definition
 - **FALSE** means evaluates to **ZERO**
 - **TRUE** means evaluates to any **NON-ZERO** integer(even negative integers)

Relational Operators Table

- The following table shows the various C relational operators

<i>Operator</i>	<i>Meaning</i>	<i>Example</i>
==	Equal to	count == 10
!=	Not equal to	flag != DONE
<	Less than	a < b
<=	Less than or equal to	i <= LIMIT
>	Greater than	pointer > end_of_list
>=	Greater than or equal to	lap >= start

- The relational operators have a precedence **below** the arithmetic operators.

for Loop

- The for loop is C's form of an unconditional loop. The basic syntax of the for statement is,

```
for (initialization expression; test expr; increment expr)  
program statement;
```

- Here is an example

```
sum=10;  
for (i=0; i<6; ++i)  
    sum = sum+i;
```

- The operation for the loop is as follows
 - 1) The initialization expression is evaluated.
 - 2) The test expression is evaluated. If it is TRUE, body of the loop is executed. If it is FALSE, exit the for loop.
 - 3) Assume test expression is TRUE. Execute the program statements making up the body of the loop.
 - 4) Evaluate the increment expression and return to step 2.
 - 5) When test expression is FALSE, exit loop and move on to next line of code.

for Loop Example

- Sample Loop:

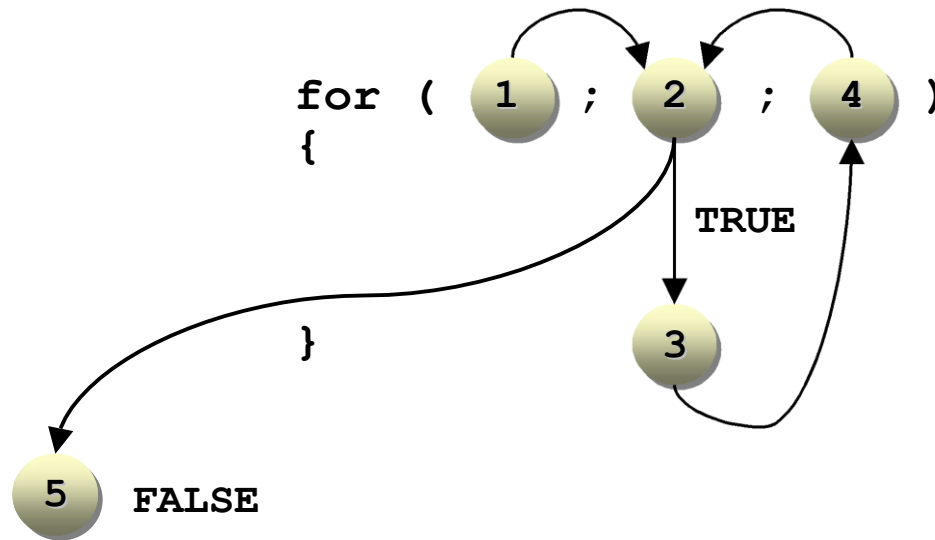
```
sum = 10;  
for (i=0; i<6; ++i)  
    sum=sum+i;
```

- We can trace the execution of the sample loop as follows

<i>Iteration</i>	<i>i</i>	<i>i<6</i>	<i>sum</i>
1 st	0	TRUE	10
2 nd	1	TRUE	11
3 rd	2	TRUE	13
4 th	3	TRUE	16
5 th	4	TRUE	20
6 th	5	TRUE	25
7 th	6	FALSE	25

for Loop Diagram

- The following diagram illustrates the operation of a for loop



General Comments about for Loop

- Some general comments regarding the use of the **for** statement:
 - Control expressions are separated by ; not ,
 - If there are multiple C statements that make up the loop body, enclose them in brackets (USE INDENTATION FOR READABILITY)

```
for (x=100; x!=65; x-=5)    {  
    z=sqrt(x);  
    printf("The square root of %d is %f\n",x,z);  
}
```

- Control expressions can be any valid expression. Don't necessarily have to perform initialization, testing, and incrementation.
- Any of the control expressions can be omitted (but always need the two semicolons for syntax sake).

```
product=1;  
for (i=1;i<=6;)   
    product*=i++;
```

General Comments about for Loop Continued

- Some general comments regarding the use of the `for` statement:
 - Since test performed at beginning of loop, body may never get executed
- Can string together multiple expressions in the `for` statement by separating them by commas

```
x=10;  
for (y=10;y!=x;++y)  
    printf ("%d",y);
```

```
for (x=1,y=5;x+y<100;++x)  
    z=x%y;
```

while Loop

- The **while** loop provides a mechanism for repeating C statements while a condition is true. Its format is

```
while(control expression)  
    program statement;
```

- The **while** statement works as follows:
 - 1) Control expression is evaluated (“entry condition”)
 - 2) If it is FALSE, skip over the loop.
 - 3) If it is TRUE, loop body is executed.
 - 4) Go back to step 1

while Loop Example

- Example while loop

```
i=1; factorial=1;
while (i<=n) {
    factorial *= i;
    i=i+1;
}
```

- **Programmer is responsible for initialization and incrementation.** At some point in the body of the loop, the control expression must be altered in order to allow the loop to finish. Otherwise: **infinite loop**.

- Will this loop end?

```
j=15;
while (j--)
    ...;
```

do while Loop

- The **do while** statement is a variant of the while statement in which the condition test is performed at the “bottom” of the loop. This guarantees that the loop is executed at least once.
- The syntax of the **do while** statement is

```
do
    program statement;
while (control expression);
```
- and it works as follows
 - 1) The body of the loop is executed.
 - 2) The control expression is evaluated (“exit condition”).
 - 3) If it is TRUE, go back to step 1. If it is FALSE, exit loop.

do while Loop Example

- Here is a sample program that reverses an integer with a **do while** loop:

```
main() {
    int value, r_digit;
    printf("Enter the number to be reversed.\n");
    scanf("%d", &value);
    do {
        r_digit = value % 10;
        printf("%d", r_digit);
        value = value / 10;
    } while (value != 0);
    printf("\n");
}
```

do while Loop Example: Error Checking

- A common use of the **do while** statement is input error checking. A simple form is shown here

```
do {  
    printf("\n Input a positive integer: ");  
    scanf("%d",&n);  
} while (n<=0);
```

- The user will remain in this loop continually being prompted for and entering integers until a positive one is entered. A sample session using this loop looks like this

```
Input a positive integer: -4  
Input a positive integer: -34  
Input a positive integer: 6
```

Decision Making Statements

- [Introduction to Decision Making Statements](#)
- [if Statement](#)
- [if Statement Examples](#)
- [if-else Statement](#)
- [if-else Ladder](#)
- [switch Statement](#)
- [switch Statement Example](#)
- [switch Statement Operation](#)
- [switch Statement Example: Characters](#)
- [switch Statement Example: Menus](#)
- [Conditional Operator](#)
- [Conditional Operator Examples](#)
- [Logical Operators](#)
- [Logical Operators Precedence](#)

Introduction to Decision Making Statements

- Used to have a program execute different statements depending on certain conditions. In a sense, makes a program “smarter” by allowing different choices to be made. In C, there are three decision making statements.

if	execute a statement or not
if-else	choose to execute one of two statements
switch	choose to execute one of a number of statements

if Statement

- The **if** statement allows branching (decision making) depending upon a condition. **Program code is executed or skipped**. The basic syntax is

```
if (control expression)  
    program statement;
```

- If the control expression is TRUE, the body of the **if** is executed. If it is FALSE, the body of the **if** is skipped.
- There is **no “then”** keyword in C!
- Because of the way in which floating point types are stored, it makes it very difficult to compare such types for equality. **Avoid trying to compare real variables for equality**, or you may encounter unpredictable results.

if Statement Examples

- These code fragments illustrate some uses of the **if** statement

- **Avoid division by zero**

```
if (x!=0)
    y/=x;
```

- **Customize output**

```
if (grade>=90)
    printf("\nCongratulations!");
printf("\nYour grade is \"%d\",grade);
```

- **Nested ifs**

```
if (letter>='A')
    if (letter>='Z')
        printf("The letter is a capital \n");
```

if-else Statement

- **Used to decide between two courses of action.** The syntax of the **if-else** statement is

```
if (expression)
    statement1;
else
    statement2;
```

- If the expression is TRUE, *statement1* is executed; *statement2* is skipped.
- If the expression is FALSE, *statement2* is executed; *statement1* is skipped.
- Some examples

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

```
if (letter == 'e') {
    ++e_count;
    ++vowel_count; }
else
    ++other_count;
```

if-else Ladder

- What if we wanted to extend the task shown in the previous example and not just counts how many e's there are in a piece of text, but also make counts of the other vowels? This is possible by nesting **if-else** statements together to make what is called an **if-else ladder**. For example, consider the following code

```
if (letter == 'a')
    ++a_count;
else if (letter == 'e')
    ++e_count;
else if (letter == 'i')
    ++i_count;
else if (letter == 'o')
    ++o_count;
else if (letter == 'u')
    ++u_count;
else
    ++const_count;
```

- As soon as a TRUE control expression is found, the statement associated with it is executed and the rest of the ladder is bypassed. If no control expressions are found to be TRUE, the final **else** statement acts as a default.

switch Statement

- The **switch** statement is a better way of writing a program which employs an **if-else** ladder. It is C's built-in **multiple branch decision statement**. The syntax for the **switch** statement is as follows:

```
switch (integer expression) {  
    case constant1:  
        statement1;  
        break;  
    case constant2:  
        statement2;  
        break;  
    ...  
    default:  
        statement;  
}
```

- The keyword **break** should be included at the end of each case statement. In general, whenever a **break** statement is encountered in C, it interrupts the normal flow of control. In the **switch** statement, it causes an exit from the switch shunt. The **default** clause is optional. The right brace at the end marks the end of **switch** statement.

switch Statement Example

- Here is a simple example of a **switch** statement:

```
switch(n) {  
    case 12:  
        printf("value is 12\n");  
        break;  
    case 25:  
        printf("value is 25\n");  
        break;  
    case 99:  
        printf("value is 99\n");  
        break;  
    default:  
        printf("number is not part of the Xmas date\n");  
}
```

switch Statement Operation

- The **switch** statement works as follows
 - 1) Integer control expression is evaluated.
 - 2) A match is looked for between this expression value and the **case constants**. If a match is found, execute the statements for that **case**. If a match is not found, execute the **default** statement.
 - 3) Terminate **switch** when a **break** statement is encountered or by “falling out the end”.
- Some things to be aware of when using a switch statement:
 - **case** values must be unique (How to decide otherwise?)
 - **switch** statement only tests for equality
 - The *control expression* can be of type character since they are internally treated as integers

switch Statement Example: Characters

```
switch(ch) {  
    case 'a':  
        ++a_count;  
        break;  
    case 'b':  
        ++b_count;  
        break;  
    case 'c':  
    case 'C':    /* multiple values, same statements */  
        ++c_count; }  

```

switch Statement Example: Menus

- A common application of the **switch** statement is to control menu-driven software:

```
switch(choice) {  
    case 'S':  
        check_spelling();  
        break;  
    case 'C':  
        correct_errors();  
        break;  
    case 'D':  
        display_errors();  
        break;  
    default:  
        printf("Not a valid option\n"); }  
}
```

Conditional Operator

- Short-hand notation for an **if-else statement that performs assignments**. This conditional expression operator takes THREE operands. The two symbols used to denote this operator are the **?** and the **:**. The first operand is placed before the **?**, the second operand between the **?** and the **:**, and the third after the **:**. The general syntax is thus

condition ? expression1 : expression2;

- If the result of *condition* is TRUE (non-zero), *expression1* is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then *expression2* is evaluated and its result becomes the result of the operation. Consider the example on the next page:

Conditional Operator Examples

```
s = (x<0) ? -1 : x*x;
```

- If **x** is less than zero, then **s=-1**. If **x** is greater than or equal to zero, then **s=x*x**.
- The following code sets the logical status of the variable **even**

```
if (number%2==0)
    even=1;
else
    even=0;
```

- Identical, short-hand code to perform the same task is

```
even = (number%2==0) ? 1 : 0;
```

Logical Operators

- These operators are used to create more sophisticated conditional expressions which can then be used in any of the C looping or decision making statements we have just discussed. When expressions are combined with a logical operator, either TRUE (i.e., 1) or FALSE (i.e., 0) is returned.

Operator	Symbol	Usage	Operation
LOGICAL AND	&&	exp1 && exp2	Requires both exp1 and exp2 to be TRUE to return TRUE. Otherwise, the logical expression is FALSE.
LOGICAL OR	 	exp1 exp2	Will be TRUE if either (or both) exp1 or exp2 is TRUE. Otherwise, it is FALSE.
LOGICAL NOT	!	!exp	Negates (changes from TRUE to FALSE and visa versa) the expression.

Logical Operators Precedence

- The negation operator, `!`, has the highest precedence and is always performed first in a mixed expression. The remaining logical operators have a precedence below relational operators.
- Some typical examples using logical operators:

```
if (year<1900 && year>1799)
    printf("Year in question is in the 19th century\n");
```

```
if (ch=='a' || ch=='e' || ch=='i' || ch=='o' || ch=='u')
    ++vowel_count;
```

```
done=0;
while(!done) {
    ...
}
```

Array Variables

- [Introduction to Array Variables](#)
- [Array Variables Example](#)
- [Array Elements](#)
- [Declaring Arrays](#)
- [Initializing Arrays during Declaration](#)
- [Using Arrays](#)
- [Multi-dimensional Arrays](#)
- [Multi-dimensional Array Illustration](#)
- [Initializing Multi-dimensional Arrays](#)
- [Using Multi-dimensional Arrays](#)

Introduction to Array Variables

- Arrays are a data structure which **hold multiple values of the same data type**. Arrays are an example of a **structured variable** in which 1) there are a number of pieces of data contained in the variable name, and 2) there is an ordered method for extracting individual data items from the whole.
- Consider the case where a programmer needs to keep track of the ID numbers of people within an organization. Her first approach might be to create a specific variable for each user. This might look like

```
int id1 = 101;      int id2 = 232;      int id3 = 231;
```

- It becomes increasingly more difficult to keep track of the IDs as the number of variables increase. Arrays offer a solution to this problem.

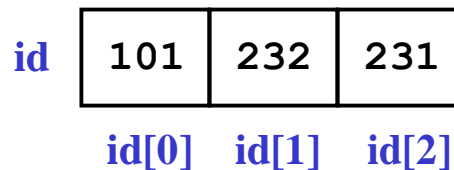
Array Variables Example

- An array is a multi-element box, a bit like a filing cabinet, and uses an indexing system to find each variable stored within it. **In C, indexing starts at zero**. Arrays, like other variables in C, must be declared before they can be used.
- The replacement of the previous example using an array looks like this:

```
int id[3];    /* declaration of array id */
id[0] = 101;
id[1] = 232;
id[2] = 231;
```
- In the first line, we declared an array called **id**, which has space for three integer variables. Each piece of data in an array is called an **element**. Thus, array **id** has three elements. After the first line, each element of **id** is initialized with an ID number.

Array Elements

- The syntax for an element of an array called **a** is
a[i]
- where **i** is called the index of the array element. **The array element id[1] is just like any normal integer variable and can be treated as such.**
- In memory, one can picture the array id as in the following diagram:



Declaring Arrays

- Arrays may consist of any of the valid data types. Arrays are declared along with all other variables in the declaration section of the program and the following syntax is used

```
type    array_name[n] ;
```

- where *n* is the **number of elements** in the array. Some examples are

```
int      final[160] ;  
float    distance[66] ;
```

- During declaration **consecutive memory locations** are reserved for the array and all its elements. After the declaration, you cannot assume that the elements have been initialized to zero. Random junk is at each element's memory location.

Initializing Arrays during Declaration

- If the declaration of an array is preceded by the word `static`, then the array can be initialized at declaration. The initial values are enclosed in braces. e.g.,

```
static int    value[9] = {1,2,3,4,5,6,7,8,9};  
static float  height[5]={6.0,7.3,2.2,3.6,19.8};
```

- Some rules to remember when initializing during declaration
 - 1 If the list of initial elements is **shorter** than the number of array elements, the **remaining elements are initialized to zero**.
 - 2 If a static array is **not initialized** at declaration manually, **its elements are automatically initialized to zero**.
 - 3 If a static array is declared **without a size** specification, its **size equals the length of the initialization list**. In the following declaration, **a** has size 5.

```
static int    a[]={-6,12,18,2,323};
```

Using Arrays

- Recall that indexing is the method of accessing individual array elements. Thus `grade[89]` refers to the 90th element of the `grades` array. A common programming error is **out-of-bounds array indexing**. Consider the following code:

```
int    grade[3];  
grade[5] = 78;
```

- The result of this mistake is unpredictable and machine and compiler dependent. You could write over important memory locations, for example. Often run-time errors result.
- Array variables and for loops often work hand-in-hand** since the for loop offers a convenient way to successively access array elements and perform some operation with them. Basically, the for loop counter can do double duty and act as an index for the array, as in the following summation example:

```
int total=0,i;  
int grade[4]={93,94,67,78};  
for (i=0; i<4; ++i)  
    total += grade[i];
```

Multi-Dimensional Arrays

- Multi-dimensional arrays have two or more index values which are used to specify a particular element in the array. For this 2D array element,

`image[i][j]`

- the first index value `i` specifies a row index, while `j` specifies a column index. Declaring multi-dimensional arrays is similar to the 1D case:

```
int a[10];           /* declare 1D array */
float b[3][5];       /* declare 2D array */
double c[6][4][2];   /* declare 3D array */
```

- Note that it is quite easy to allocate a large chunk of **consecutive memory** with multi-dimensional arrays. Array `c` contains **6x4x2=48 doubles**.

Multi-Dimensional Array Illustration

- A useful way to picture a 2D array is as a grid or matrix. Picture array **b** as

	0 th column	1 st column	2 nd column	3 rd column	4 th column
0 th row	b[0][0]	b[0][1]	b[0][2]	b[0][3]	b[0][4]
1 st row	b[1][0]	b[1][1]	b[1][2]	b[1][3]	b[1][4]
2 nd row	b[2][0]	b[2][1]	b[2][2]	b[2][3]	b[2][4]

- In C, **2D arrays are stored by row**. Which means that in memory the **0th row** is put into its memory locations, the **1st row** then takes up the next memory locations, the **2nd row** takes up the next memory locations, and so on.

Initializing Multi-Dimensional Arrays

- This procedure is entirely analogous to that used to initialize 1D arrays at their declaration. For example, this declaration

```
static int age[2][3]={4,8,12,19,6,-1};
```

- will fill up the array age as it is stored in memory. That is, the array is initialized row by row. Thus, the above statement is equivalent to:

```
age[0][0]=4; age[0][1]=8; age[0][2]=12;  
age[1][0]=19; age[1][1]=6; age[1][2]=-1;
```

- As before, if there are fewer initialization values than array elements, the remainder are initialized to zero.
- To make your program more readable, you can explicitly put the values to be assigned to the same row in inner curly brackets:

```
static int age[2][3]={ {4,8,12}, {19,6,-1} };
```

- In addition if the number of rows is omitted from the actual declaration, it is set equal to the number of inner brace pairs:

```
static int age[][3]= ]={ {4,8,12}, {19,6,-1} };
```


Using Multi-Dimensional Arrays

- Again, as with 1D arrays, **for** loops and multi-dimensional arrays often work hand-in-hand. In this case, though, **loop nests** are what is most often used. Some examples

Summation of array elements

```
double temp[256][3000],sum=0;
int i,j;
for (i=0; i<256; ++i)
    for (j=0; j<3000; ++j)
        sum += temp[i][j];
```

Trace of Matrix

```
int voxel[512][512][512];
int i,j,k,trace=0;
for (i=0; i<512; ++i)
    for (j=0; j<512; ++j)
        for (k=0; k<512; ++k)
            if (i==j && j==k)
                trace += voxel[i][j][k];
```

Strings

- [Arrays of Characters](#)
- [Initializing Strings](#)
- [Copying Strings](#)
- [String I/O Functions](#)
- [More String Functions](#)
- [More String Functions Continued](#)
- [Examples of String Functions](#)
- [Character I/O Functions](#)
- [More Character Functions](#)
- [Character Functions Example](#)

Arrays of Characters

- Strings are **1D arrays of characters**. Strings must be terminated by the null character `'\0'` which is (naturally) called the **end-of-string character**. Don't forget to remember to count the end-of-string character when you calculate the size of a string.
- As will all C variables, strings must be declared before they are used. Unlike other 1D arrays the **number of elements set** for a string set during declaration is only an **upper limit**. The actual strings used in the program can have fewer elements. Consider the following code:

```
static char name[18] = "Ivanova";
```

- The string called `name` actually has only 8 elements. They are
`'I' 'v' 'a' 'n' 'o' 'v' 'a' '\0'`
- Notice another interesting feature of this code. String constants **marked with double quotes** automatically include the end-of-string character. The **curly braces are not required** for string initialization at declaration, but can be used if desired (**but don't forget the end-of-string character**).

Initializing Strings

- Initializing a string can be done in three ways: 1) at declaration, 2) by reading in a value for the string, and 3) by using the **strcpy** function. **Direct initialization using the = operator is invalid.** The following code would produce an error:

```
char name[34];  
name = "Erickson";    /* ILLEGAL */
```

- To read in a value for a string use the **%s** format identifier:

```
scanf ("%s", name) ;
```

- Note that the address operator **&** is not needed for inputting a string variable (explained later). The end-of-string character will automatically be appended during the input process.

Copying Strings

- The **strcpy function** is one of a set of built-in string handling functions available for the C programmer to use. To use these functions be sure to include the **string.h** header file at the beginning of your program. The syntax of **strcpy** is

strcpy(string1, string2) ;

- When this function executes, **string2** is copied into **string1** at the beginning of **string1**. The previous contents of **string1** are overwritten.
- In the following code, **strcpy** is used for string initialization:

```
#include <string.h>
main ()    {
    char job[50];
    strcpy(job, "Professor");
    printf("You are a %s \n", job);
}
```

You are a Professor

String I/O Functions

- There are special functions designed specifically for string I/O. They are

`gets(string_name);`

`puts(string_name);`

- The **gets** function reads in a string from the keyboard. When the user hits a carriage return the string is inputted. The carriage return is not part of the string and the end-of-string character is automatically appended.
- The function **puts** displays a string on the monitor. It does not print the end-of-string character, but does output a carriage return at the end of the string. Here is a sample program demonstrating the use of these functions:

```
char phrase[100];  
printf("Please enter a sentence\n");  
gets(phrase);  
puts(phrase);
```

- A sample session would look like this

```
Please enter a sentence  
The best lack all conviction, while the worst are passionate.  
The best lack all conviction, while the worst are passionate.
```

More String Functions

- Included in the **string.h** are several more string-related functions that are free for you to use. Here is a brief table of some of the more popular ones

Function	Operation
strcat	Appends to a string
strchr	Finds first occurrence of a given character
strcmp	Compares two strings
strcmpi	Compares two, strings, non-case sensitive
strcpy	Copies one string to another
strlen	Finds length of a string
strncat	Appends <i>n</i> characters of string
strncmp	Compares <i>n</i> characters of two strings
strncpy	Copies <i>n</i> characters of one string to another
strnset	Sets <i>n</i> characters of string to a given character
strrchr	Finds last occurrence of given character in string
strspn	Finds first substring from given character set in string

More String Functions Continued

- Most of the functions on the previous page are self-explanatory. **The UNIX man pages provide a full description of their operation.** Take for example, **strcmp** which has this syntax

`strcmp(string1, string2);`

- It returns an integer that is less than zero, equal to zero, or greater than zero depending on whether *string1* is less than, equal to, or greater than *string2*.
- String comparison is done character by character using the ASCII numerical code

Examples of String Functions

- Here are some examples of string functions in action:

```
static char s1[]="big sky country";
static char s2[]="blue moon";
static char s3[]="then falls Caesar";
```

<i>Function</i>	<i>Result</i>
<code>strlen(s1)</code>	15 /* e-o-s not counted */
<code>strlen(s2)</code>	9
<code>strcmp(s1,s2)</code>	<i>negative number</i>
<code>strcmp(s3,s2)</code>	<i>positive number</i>
<code>strcat(s2," tonight")</code>	blue moon tonight

Character I/O Functions

- Analogous to the gets and puts functions there are the getchar and putchar functions specially designed for character I/O. The following program illustrates their use:

```
#include <stdio.h>
main() {
    int n; char lett;
    putchar('?');
    n=45;
    putchar(n-2);
    lett=getchar();
    putchar(lett);
    putchar('\n');
}
```

- A sample session using this code would look like:

```
?+f◀
f
```

More Character Functions

- As with strings, there is a library of functions designed to work with character variables. The file **ctype.h** defines additional routines for manipulating characters. Here is a partial list

<i>Function</i>	<i>Operation</i>
isalnum	Tests for alphanumeric character
isalpha	Tests for alphabetic character
isascii	Tests for ASCII character
iscntrl	Tests for control character
isdigit	Tests for 0 to 9
isgraph	Tests for printable character
islower	Tests for lowercase character
isprint	Tests for printable character
ispunct	Tests for punctuation character
isspace	Tests for space character
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal
toascii	Converts character to ASCII code
tolower	Converts character to lowercase
toupper	Converts character to upper

Character Functions Example

- In the following program, character functions are used to convert a string to all uppercase characters:

```
#include <stdio.h>
#include <ctype.h>
main() {
    char name[80];
    int loop;
    printf ("Please type in your name\n");
    gets(name);
    for (loop=0; name[loop] !=0; loop++)
        name[loop] = toupper(name[loop]);
    printf ("You are %s\n",name);
}
```

- A sample session using this program looks like this:

```
Please type in your name
Dexter Xavier
You are DEXTER XAVIER
```

Math Library Functions

- [“Calculator-class” Functions](#)
- [Using Math Library Functions](#)

“Calculator-class” Library Functions

- You may have started to guess that there should be a header file called **math.h** which contains definitions of useful “calculator-class” mathematical functions. Well there is! Some functions found in **math.h** are

`acos asin atan`

`cos sin tan`

`cosh sinh tanh`

`exp log log10`

`pow sqrt`

`ceil floor`

`erf`

`gamma`

`j0 j1 jn`

`y0 y1 yn`

Using Math Library Functions

- The following code fragment uses the Pythagorean theorem $c^2 = a^2 + b^2$ to calculate the length of the hypotenuse given the other two sides of a right triangle:

```
double c, a, b
c=sqrt(pow(a,2)+pow(b,2)) ;
```

- Typically, to use the math functions declared in the **math.h** include file, the user must explicitly load the math library during compilation. On most systems the compilation would look like this:

```
cc myprog.c -lm
```

User-defined Functions

- [Introduction to User-defined Functions](#)
- [Reasons for Use](#)
- [User-defined Functions Usage](#)
- [Function Definition](#)
- [User-defined Function Example 1](#)
- [User-defined Function Example 2](#)
- [return Statement](#)
- [return Statement Example](#)
- [Using Functions](#)
- [Considerations when Using Functions](#)
- [Using Functions Example](#)
- [Introduction to Function Prototypes](#)
- [Function Prototypes](#)
- [Recursion](#)
- [Storage Classes](#)
- [auto Storage Class](#)
- [extern Storage Class](#)
- [extern Storage Class Example](#)
- [static and register Storage Class](#)

Introduction to User-defined Functions

- A function in C is a **small “sub-program”** that performs a particular task, and supports the concept of **modular programming design** techniques. In modular programming the various tasks that your overall program must accomplish are assigned to individual functions and the main program basically calls these functions in a certain order.
- We have already been exposed to functions. The main body of a C program, identified by the keyword **main**, and enclosed by left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system. We have also seen examples of **library functions** which can be used for I/O, mathematical tasks, and character/string handling.
- But can the user define and use their own functions? **Absolutely YES!**

Reasons for Use

- There are many good reasons to program in a modular style:
 - Don't have to **repeat the same block of code** many times in your code. Make that code block a function and call it when needed.
 - **Function portability**: useful functions can be used in a number of programs.
 - Supports the **top-down technique** for devising a program algorithm. Make an outline and hierarchy of the steps needed to solve your problem and create a function for each step.
 - **Easy to debug**. Get one function working well then move on to the others.
 - **Easy to modify and expand**. Just add more functions to extend program capability
 - For a **large programming project**, you will code only a small fraction of the program.
 - Make program **self-documenting** and readable.

User-defined Function Usage

- In order to use functions, the programmer must do three things
 - **Define** the function
 - **Declare** the function
 - **Use** the function in the main code.
- In the following pages, we examine each of these steps in detail.

Function Definition

- The function definition is the C code that implements what the function does. Function definitions have the following syntax

```
return_type function_name (data type variable name list)
{
    local declarations;
    function statements;
}
```

The diagram illustrates the syntax of a function definition. It shows the following components with arrows pointing to them:

- function header**: Points to the first line of the function definition, *return_type function_name (data type variable name list)*.
- function body**: Points to the block of code inside the curly braces, specifically to the *local declarations;* and *function statements;* lines.

- where the *return_type* in the function header tells the type of the value returned by the function (default is **int**)
- where the *data type variable name list* tells what arguments the function needs when it is called (and what their types are)
- where *local declarations* in the function body are local constants and variables the function needs for its calculations.

Function Definition Example 1

- Here is an example of a function that calculates $n!$

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

Function Definition Example 2

- Some functions will not actually return a value or need any arguments. For these functions the keyword **void** is used. Here is an example:

```
void write_header(void) {  
    printf("Navier-Stokes Equations Solver ");  
    printf("v3.45\n");  
    printf("Last Modified: ");  
    printf("12/04/95 - viscous coefficient added\n");  
}
```

- The **1st void** keyword indicates that **no value will be returned**.
- The **2nd void** keyword indicates that **no arguments** are needed for the function.
- This makes sense because all this function does is print out a header statement.

return Statement

- A function returns a value to the calling program with the use of the **keyword return**, followed by a data variable or constant value. The return statement can even contain an expression. Some examples

```
return 3;
```

```
return n;
```

```
return ++a;
```

```
return (a*b) ;
```

- When a **return** is encountered the following events occur:
 - 1 execution of the function is terminated and control is passed back to the calling program, and
 - 2 the function call evaluates to the value of the ***return expression***.
- If there is no **return** statement control is passed back when the closing brace of the function is encountered (“falling off the end”).

return Statement Examples

- The data type of the *return expression* **must match** that of the declared *return_type* for the function.

```
float add_numbers (float n1, float n2) {  
    return n1 + n2; /*legal*/  
    return 6;       /*illegal, not the same data type*/  
    return 6.0;     /*legal*/ }
```

- It is possible for a function to have **multiple return statements**. For example:

```
double absolute(double x) {  
    if (x>=0.0)  
        return x;  
    else  
        return -x;  
}
```


Using Functions

- This is the easiest part! To invoke a function, **just type its name** in your program and be sure to supply arguments (if necessary). A statement using our factorial program would look like

```
number=factorial(9);
```

- To invoke our write_header function, use this statement

```
write_header();
```

- When your program encounters a function invocation, control passes to the function. When the function is completed, control passes back to the main program. In addition, if a value was returned, the function call takes on that return value. In the above example, upon return from the **factorial** function the statement

```
factorial(9) → 362880
```

- and that integer is assigned to the variable **number**.

Considerations when using Functions

- Some points to keep in mind when calling functions (your own or library's):
 - The **number** of arguments in the function call **must match** the **number** of arguments in the function definition.
 - The **type** of the arguments in the function call **must match** the **type** of the arguments in the function definition.
 - The **actual arguments** in the function call are **matched up in-order** with the **dummy arguments** in the function definition.
 - The actual arguments are **passed by-value** to the function. The dummy arguments in the function are initialized with the present values of the actual arguments. *Any changes made to the dummy argument in the function will NOT affect the actual argument in the main program.*

Using Function Example

- The independence of actual and dummy arguments is demonstrated in the following program.

```
#include <stdio.h>
int compute_sum(int n) {
    int sum=0;
    for(;n>0;--n)
        sum+=n;
    printf("Local n in function is %d\n",n);
    return sum; }
main() {
    int n=8,sum;
    printf ("Main n (before call) is %d\n",n);
    sum=compute_sum(n);
    printf ("Main n (after call) is %d\n",n);
    printf ("\nThe sum of integers from 1 to %d is %d\n",n,sum);}
```

```
Main n (before call) is 8
```

```
Local n in function is 0
```

```
Main n (after call) is 8
```

```
The sum of integers from 1 to 8 is 36
```

Introduction to Function Prototypes

- Function prototypes are **used to declare a function** so that it can be used in a program before the function is actually defined. Consider the program on the previous page. In some sense, it reads “backwards”. All the secondary functions are defined first, and then we see the main program that shows the major steps in the program. This example program can be rewritten using a function prototype as follows:

```
#include <stdio.h>
int compute_sum(int n); /* Function Prototype */
main() {
    int n=8,sum;
    printf ("Main n (before call) is %d\n",n);
    sum=compute_sum(n);
    printf ("Main n (after call) is %d\n",n);
    printf ("\nThe sum of integers from 1 to %d is %d\n",n,sum);}
int compute_sum(int n) {
    int sum=0;
    for(;n>0;--n)
        sum+=n;
    printf("Local n in function is %d\n",n);
    return sum; }
```

Function Prototypes

- Now the program reads in a "natural" order. You know that a function called **compute_sum** will be defined later on, and you see its immediate use in the main program. Perhaps you don't care about the details of how the sum is computed and you won't need to read the actual function definition.
- As this example shows, a function prototype is simply the **function header** from the function definition **with a semi-colon attached to the end**. The prototype tells the compiler the number and type of the arguments to the function and the type of the return value. Function prototypes should be placed **before the start of the main program**. The function definitions can then follow the main program. In fact, if you look at one of the include files -- say **string.h** -- you will see the prototypes for all the string functions available!
- In addition to making code more readable, the use of function prototypes offers **improved type checking** between actual and dummy arguments. In some cases, the type of actual arguments will automatically be coerced to match the type of the dummy arguments.

Recursion

- Recursion is the process in which a function repeatedly calls itself to perform calculations. Typical applications are games and sorting trees and lists. Recursive algorithms are not mandatory, usually an iterative approach can be found.
- The following function calculates factorials recursively:

```
int factorial(int n) {  
    int result;  
  
    if (n<=1)  
        result=1;  
    else  
        result = n * factorial(n-1);  
  
    return result;  
}
```

Storage Classes

- Every variable in C actually has two attributes: its data type and its storage class. The storage class refers to the **manner in which memory is allocated** for the variable. The storage class also determines the **scope of the variable**, that is, what parts of a program the variable's name has meaning. In C, the four possible Storage classes are
 - auto
 - extern
 - static
 - register

auto Storage Class

- This is the **default classification** for all variables declared within a function body [including **main()**].
- Automatic variables are truly **local**.
- They exist and their names have meaning only while the function is being executed.
- They are unknown to other functions.
- When the function is exited, the values of automatic variables are not retained.
- They are normally implemented on a stack.
- They are recreated each time the function is called.

extern Storage Class

- In contrast, extern variables are **global**.
- If a variable is declared at the beginning of a program outside all functions [including **main()**] it is classified as an external by default.
- **External variables can be accessed and changed by any function in the program.**
- Their storage is in permanent memory, and thus never disappear or need to be recreated.

What is the advantage of using global variables?

It is a method of transmitting information between functions in a program without using arguments.

extern Storage Class Example

- The following program illustrates the global nature of extern variables:

```
#include <stdio.h>
int a=4,b=5,c=6;    /* default extern */
int sum(void); int prod(void);
main() {
    printf ("The sum is %d\n",sum());
    printf ("The product is %d\n",prod());
}
int sum(void) {
    return (a+b+c); }
int prod(void) {
    return (a*b*c); }
```

```
The sum is 15
The product is 120
```

- There are two disadvantages of global variables versus arguments. **First**, the function is much less portable to other programs. **Second**, is the concept of **local dominance**. If a local variable has the **same name** as a global variable, only the local variable is changed while in the function. Once the function is exited, the global variable has the same value as when the function started.

static and register Storage Class

static Storage Class

- A static variable is a local variable that **retains its latest value** when a function is recalled. Its scope is still local in that it will only be recognized in its own function. Basically, static variables are created and initialized once on the first call to the function. With clever programming, one can use static variables to enable a function to do different things depending on how many times it has been called. (Consider a function that counts the number of times it has been called).

register Storage Class

- It is often true that the time bottleneck in computer calculations is the time it takes to fetch a variable from memory and store its value in a register where the CPU can perform some calculation with it. So for performance reasons, it is sometimes advantageous to store variables directly in registers. This strategy is most often used with loop counter variables, as shown below.

```
register int i;  
for (i=0; i<n; ++i)  
    ...
```

Formatted Input and Output

- [Formatted Output](#)
- [char and int Formatted Output Example](#)
- [f Format Identifier](#)
- [e Format Identifier](#)
- [Real Formatted Output Example](#)
- [s Format Identifier](#)
- [Strings Formatted Output Example](#)
- [Formatted Input](#)
- [Formatted Input Examples](#)

Formatted Output

- Can you control the appearance of your output on the screen? Or do you have to accept the default formatting provided by the C compiler? It turns out you can format your output in a number of ways.
- You can control how many columns will be used to output the contents of a particular variable by specifying the **field width**. The desired field width is inserted in the format specifier after the % and before the letter code indicating the data type. Thus the format specifier **%5d** is interpreted as use 5 columns to display the integer. Further examples:

%3c display the character in 3 columns

%13x display the hexadecimal integer in 13 columns

- Within the field, the argument value is **right-adjusted** and **padded with blanks**. If left adjustment is preferred use the syntax **%-3c**. If you wish to pad with zeros use the syntax **%04d**.

Nice Feature:

If the value to be printed out takes up more columns than the specified field width, the field is **automatically expanded**.

char and int Formatted Output Example

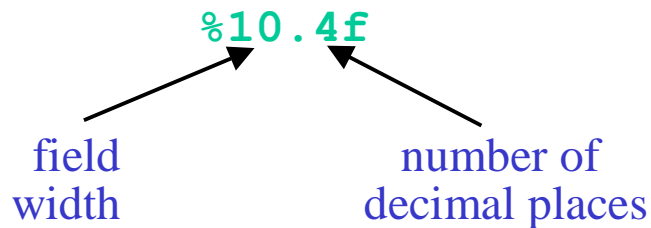
- This program and its output demonstrate various-sized field widths and their variants.

```
#include <stdio.h>
main() {
    char lett='w';
    int i=1, j=29;
    printf ("%c\n", lett);
    printf ("%4c\n", lett);
    printf ("% -3c\n\n", lett);
    printf ("%d\n", i);
    printf ("%d\n", j);
    printf ("%10d\n", j);
    printf ("%010d\n", j);
    printf ("% -010d\n", j);
    printf ("%2o\n", j);
    printf ("%2x\n", j);
}
```

```
w
  w
w
1
29
           29
0000000029
29
35
1d
```

f Format Identifier

- For floating-point values, in addition to specifying the field width, the **number of decimal places** can also be controlled. A sample format specifier would look like this



- Note that a period separates the two numbers in the format specifier. Don't forget to count the column needed for the decimal point when calculating the field width. We can use the above format identifier as follows:

```
printf("%10.4f", 4.0/3.0); → ----1.3333
```

- where `-` indicates the blank character.

e Format Identifier

- When using the e format identifier, the second number after the decimal point determines **how many significant figures** (SF) will be displayed. For example

`printf("%10.4e", 4.0/3.0);` → `_1.333e+10`

↙ number of significant figures

- Note that only 4 significant figures are shown. Remember that now the field size must include the actual numerical digits as well as columns for ‘.’, ‘e’, and ‘+00’ in the exponent.
- It is possible to print out as many SFs as you desire. But it only makes sense to print out as many SFs as match the precision of the data type. The following table shows a rough guideline applicable to some machines:

<i>Data Type</i>	<i># Mantissa bits</i>	<i>Precision (#SF)</i>
float	16	~7
double	32	~16
long double	64	~21

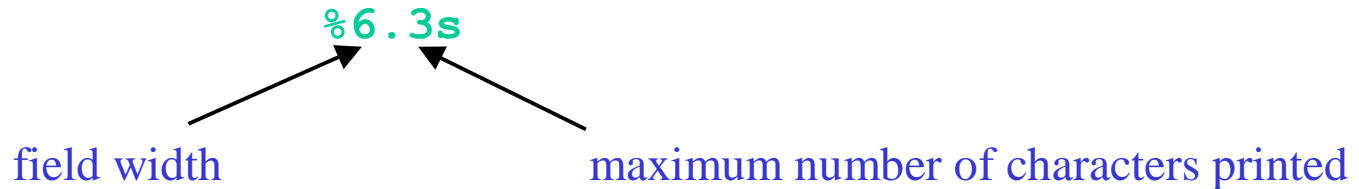
Real Formatted Output Example

```
#include <stdio.h>
main() {
    float x=333.123456;
    double y=333.1234567890123456;
    printf ("%f\n",x);
    printf ("%1f\n",x);
    printf ("%20.3f\n",x);
    printf ("%20.3f\n",x);
    printf ("%20.3f\n",x);
    printf ("%f\n",y);
    printf ("%9f\n",y);
    printf ("%20f\n",y);
    printf ("%20.4e\n",y);
}
```

```
333.123444
333.1
          333.123
333.123
00000000000000333.123
333.123457
333.123456789
333.12345678901232304270
          3.331e+02
```

s Format Identifier

- For strings, the **field length** specifier works as before and will **automatically expand** if the string size is bigger than the specification. A more sophisticated string format specifier looks like this



- where the value after the decimal point specifies the **maximum number of characters** printed.
- For example;

```
printf("3.4s\n", "Sheridan"); → Sher
```

Strings Formatted Output Example

```
#include <stdio.h>
main() {
    static char s[]="an evil presence";
    printf ("%s\n",s);
    printf ("%7s\n",s);
    printf ("%20s\n",s);
    printf ("% -20s\n",s);
    printf ("% .5s\n",s);
    printf ("% .12s\n",s);
    printf ("%15.12s\n",s);
    printf ("% -15.12s\n",s);
    printf ("%3.12s\n",s);
}
```

```
an evil presence
an evil presence
   an evil presence
an evil presence
an ev
an evil pres
   an evil pres
an evil pres
an evil pres
```

Formatted Input

- Modifications can be made to the control string of the **scanf** function which enable more sophisticated input. The formatting features that can be inserted into the control string are
 - **Ordinary characters (not just format identifiers) can appear in the `scanf` control string. They must exactly match corresponding characters in the input. These “normal” characters will not be read in as input.**
 - **An asterisk can be put after the `%` symbol in an input format specifier to suppress the input.**
 - **As with formatted output, a field width can be specified for inputting values. The field width specifies the number of columns used to gather the input.**

Formatted Input Examples

```
#include <stdio.h>
main() {
    int i; char lett; char word[15];
    scanf("%d , %*s %c %5s",&i,&lett,word);
    printf("%d \n %s \n %s\n",i,lett,word);
}
```

```
45 , ignore_this C read_this
45
C
read_
```

```
#include <stdio.h>
main() {
    int m,n,o;
    scanf("%d : %d : %d",&m,&n,&o);
    printf("%d \n %d \n %d\n",m,n,o);
}
```

```
10 : 15 : 17
10
15
17
```

Pointers

- [Introduction to Pointers](#)
- [Memory Addressing](#)
- [The Address Operator](#)
- [Pointer Variables](#)
- [Pointer Arithmetic](#)
- [Indirection Operator](#)
- [“Call-by-Reference” Arguments](#)
- [“Call-by-Reference” Example](#)
- [Pointers and Arrays](#)
- [Pointers and Arrays Illustration](#)
- [Pointers and Arrays Examples](#)
- [Arrays as Function Arguments](#)
- [Arrays as Function Arguments Example](#)
- [Pointers and Character Strings](#)
- [Pointers and Character Strings Example](#)

Introduction to Pointers

- Pointers are an **intimate part of C** and separate it from more traditional programming languages. Pointers **make C more powerful** allowing a wide variety of tasks to be accomplished. Pointers enable us to
 - **effectively represent sophisticated data structures**
 - **change values of actual arguments passed to functions (“call-by-reference”)**
 - **work with memory which has been dynamically allocated**
 - **more concisely and efficiently deal with arrays**
- On the other hand, pointers are usually difficult for new C programmers to comprehend and use. If you remember the following simple statement, working with pointers should be less painful...

POINTERS CONTAIN MEMORY ADDRESSES, NOT DATA VALUES!

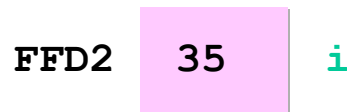
Memory Addressing

POINTERS CONTAIN MEMORY ADDRESSES, **NOT** DATA VALUES!

- When you declare a simple variable, like
`int i;`
- a memory location with a certain address is set aside for any values that will be placed in i. We thus have the following picture:



- After the statement `i=35;` the location corresponding to `i` will be filled



The Address Operator

- You can find out the **memory address of a variable** by simply using the address operator **&**. Here is an example of its use:

&v

- The above expression should be read as “address of **v**”, and it returns the memory address of the variable **v**.
- The following simple program demonstrates the difference between the contents of a variable and its memory address:

```
#include <stdio.h>
main() {
    float x;
    x=2.171828;
    printf("The value of x is %f\n",x);
    printf("The address of x is %X\n",&x); }
```

```
The value of x is 2.171828
The address of x is EFFFFBA4
```

Pointer Variables

- A pointer is a C variable that contains memory addresses. Like all other C variables, pointers must be declared before they are used. The syntax for **pointer declaration** is as follows:

```
int *p;  
double *offset;
```

- Note that the prefix ***** defines the variable to a pointer. In the above example, **p** is the type “pointer to integer” and **offset** is the type “pointer to double”.
- Once a pointer has been declared, it can be assigned an address. This is usually done with the address operator. For example,

```
int *p;  
int count;  
p=&count;
```

- After this assignment, we say that **p** is “**referring to**” the variable **count** or “**pointing to**” the variable **count**. The pointer **p** contains the memory address of the variable **count**.

Pointer Arithmetic

- A limited amount of pointer arithmetic is possible. The "unit" for the arithmetic is the size of the variable being pointed to in bytes. Thus, incrementing a pointer-to-an-int variable automatically adds to the pointer address the number of bytes used to hold an int (on that machine).
 - Integers and pointers can be added and subtracted from each other, and
 - incremented and decremented.
 - In addition, different pointers can be assigned to each other
- Some examples,

```
int *p, *q;  
p=p+2;  
q=p;
```

Indirection Operator

- The indirection operator, *****, can be considered as the complement to the address operator. It **returns the contents of the address stored in a pointer variable**. It is used as follows:

***p;**

- The above expression is read as “contents of **p**”. What is returned is the value stored at the memory address **p**.
- Consider the sample code:

```
#include <stdio.h>
main() {
    int a=1,b=78,*ip;
    ip=&a;
    b=*ip;      /* equivalent to b=a */
    printf("The value of b is %d\n",b); }
```

```
The value of b is 1
```

- Note that **b** ends up with the value of **a** but it is done **indirectly**; by using a pointer to **a**.

“Call-by-Reference” Arguments

- We learned earlier that if a variable in the main program is used as an actual argument in a function call, its value won't be changed no matter what is done to the corresponding dummy argument in the function.
- What if we would like the function to change the main variable's contents?
 - To do this we use pointers as dummy arguments in functions and indirect operations in the function body. (The actual arguments must then be addresses)
 - Since the **actual argument variable and the corresponding dummy pointer refer to the same memory location**, changing the contents of the dummy pointer will- by necessity- change the contents of the actual argument variable.

“Call-by-Reference” Example

- The classic example of “call-by-reference” is a **swap function** designed to exchange the values of two variables in the main program. Here is a swapping program:

```
#include <stdio.h>
void swap(int *p,int *q);

main() {
    int i=3,j=9876;
    swap(&i,&j);
    printf("After swap, i=%d j=%d\n",i,j);
}

void swap(int *p,int *q) {
    int temp;
    temp=*p;
    *p=*q;
    *q=temp;
}
```

```
After swap, i=9876 j=3
```

Pointers and Arrays

- Although this may seem strange at first, in C an **array name is an address**. In fact, it is the **base address** of all the consecutive memory locations that make up the entire array.
- We have actually seen this fact before: when using scanf to input a character string variable called name the statement looked like
- `scanf ("%s", name) ;` **NOT** `scanf ("%s", &name) ;`
- Given this fact, we can use pointer arithmetic to access array elements.

Pointers and Arrays Illustration

- Given the following array declaration

```
int a[467];
```

- The following two statements **do the exact same thing**:

```
a[5]=56;
```

```
*(a+5)=56;
```

- Here is the layout in memory:

a	133268	a[0]
a+1	133272	a[1]
a+2	133276	a[2]
a+3	133280	a[3]
a+4	133284	a[4]
a+5	133288	a[5]

Pointers and Arrays Examples

- The next examples show how to sum up all the elements of a 1D array using pointers:

- Normal way

```
int a[100], i, *p, sum=0;

for(i=0; i<100; ++i)
    sum +=a[i];
```

- Other way

```
int a[100], i, *p, sum=0;

for(i=0; i<100; ++i)
    sum += *(a+i);
```

- Another way

```
int a[100], i, *p, sum=0;

for(p=a; p<&a[100]; ++p)
    sum += *p;
```

Arrays as Function Arguments

- When you are writing functions that work on arrays, it is convenient to **use pointers as arguments**. Once the function has the base address of the array, it can use pointer arithmetic to work with all the array elements. The alternative is to use global array variables or -- more horribly -- pass all the array elements to the function.
- Consider the following function designed to take the sum of elements in a 1D array of doubles:

```
double sum(double *dp, int n) {  
    int i; double res=0.0;  
    for(i=0; i<n; ++i)  
        res += *(dp+i);  
    return res;  
}
```

- Note that all the sum function needed was a **starting address in the array and the number of elements** to be summed together (**n**). A very efficient argument list.

Arrays as Function Arguments Example

- Considering the previous example

```
double sum(double *dp, int n) {  
    int i; double res=0.0;  
    for(i=0; i<n; ++i)  
        res += *(dp+i);  
    return res;  
}
```

- In the main program, the sum function could be used as follows

```
double position[150],length;  
length=sum(position,150); /* sum entire array */  
length=sum(position,75); /* sum first half */  
length=sum(&position[10],10);/* sum from element  
                             10 to element 20 */
```

Pointers and Character Strings

- As strange as this sounds, a string constant -- such as “Happy Thanksgiving” -- is treated by the compiler as an address (Just like we saw with an array name). The value of the string constant address is the **base address** of the character array.
- Thus, we can **use pointers to work with character strings**, in a similar manner that we used pointers to work with “normal” arrays. This is demonstrated in the following code:

```
#include <stdio.h>

main() {
    char *cp;
    cp="Civil War";
    printf("%c\n", *cp);
    printf("%c\n", *(cp+6));
}
```

```
C
W
```

Pointers and Character Strings Example

- Another example illustrates easy string input using pointers:

```
#include <stdio.h>

main() {
    char *name;
    printf("Who are you?\n");
    scanf("%s",name);
    printf("Hi %s welcome to the party, pal\n",name);
}
```

```
Who are you?
Seymour
Hi Seymour welcome to the party, pal
```

Structures

- [Introduction to Structures](#)
- [Structure Variable Declaration](#)
- [Structure Members](#)
- [Initializing Structure Members](#)
- [Structures Example](#)
- [Structures Example Continued](#)
- [More Structures Example Continued](#)
- [Structures within Structures](#)
- [Initializing Structures within Structures](#)
- [Pointers to Structures](#)
- [Pointers to Structures: ->](#)

Introduction to Structures

- A structure is a variable in which **different types of data** can be stored together in **one variable name**. Consider the data a teacher might need for a high school student: Name, Class, GPA, test scores, final score, and final course grade. A structure data type called **student** can hold all this information:

```
struct student {  
    char name[45];  
    char class;  
    float gpa;  
    int test[3];  
    int final;  
    char grade;  
};
```

The diagram shows a C structure declaration with three annotations:

- An arrow points from the word **keyword** to the word **struct**.
- An arrow points from the text **structure data type name** to the word **student**.
- An arrow points from the text **member name & type** to the line `float gpa;`.

- The above is a **declaration of a data type** called **student**. It is not a variable declaration, but a type declaration.

Structure Variable Declaration

- To actually declare a structure variable, the standard syntax is used:

```
struct student  Lisa, Bart, Homer;
```

- You can declare a structure type and variables simultaneously. Consider the following structure representing playing cards.

```
struct playing_card {  
    int pips;  
    char *suit;  
} card1, card2, card3;
```


Structure Members

- The different variable types stored in a structure are called its members. To access a given member the **dot notation** is use. The “dot” is officially called the member access operator. Say we wanted to initialize the structure **card1** to the two of hearts. It would be done this way:

```
card1.pips=2;  
card1.suit="Hearts";
```

- Once you know how to create the name of a member variable, it can be treated the same as any other variable of that type. For example the following code:

```
card2.pips=card1.pips+5;
```

- would make **card2** the seven of some suit.
- Structure variables can also be assigned to each other, just like with other variable types:

```
card3 = card1;
```

- would fill in the card3 **pips** member with 2 and the **suit** member with “Hearts”. In other words, each member of **card3** gets assigned the value of the corresponding member of **card1**.

Initializing Structure Members

- Structure members can be **initialized at declaration**. This is similar to the initialization of arrays; the initial values are simply listed inside a pair of braces, with each value separated by a comma. The structure declaration is preceded by the keyword **static**

```
static struct student Lisa = {  
    "Simpson", 'S', 3.95, 100, 87, 92, 96, 'A' };
```

- The same member names can appear in different structures. There will be no confusion to the compiler because when the member name is used it is prefixed by the name of the structure variable. For example:

```
struct fruit {  
    char *name;  
    int calories; } snack;  
struct vegetable {  
    char *name;  
    int calories; } dinner_course;  
snack.name="banana";  
dinner_course.name="broccoli";
```

Structures Example

- What data type are allowed to structure members? **Anything goes**: basic types, arrays, strings, pointers, even other structures. You can even make an **array of structures**.
- Consider the program on the next few pages which uses an array of structures to make a deck of cards and deal out a poker hand.

```
#include <stdio.h>
struct playing_card {
    int pips;
    char *suit;    } deck[52];
void make_deck(void);
void show_card(int n);
main() {
    make_deck();
    show_card(5);
    show_card(37);
    show_card(26);
    show_card(51);
    show_card(19);
}
```

Structures Example Continued

```
void make_deck(void) {  
    int k;  
    for(k=0; k<52; ++k) {  
        if (k>=0 && k<13) {  
            deck[k].suit="Hearts";  
            deck[k].pips=k%13+2; }  
        if (k>=13 && k<26) {  
            deck[k].suit="Diamonds";  
            deck[k].pips=k%13+2; }  
        if (k>=26 && k<39) {  
            deck[k].suit="Spades";  
            deck[k].pips=k%13+2; }  
        if (k>=39 && k<52) {  
            deck[k].suit="Clubs";  
            deck[k].pips=k%13+2; }  
    }  
}
```

More on Structures Example Continued

```
void show_card(int n) {  
    switch(deck[n].pips) {  
        case 11:  
            printf("%c of %s\n", 'J', deck[n].suit);  
            break;  
        case 12:  
            printf("%c of %s\n", 'Q', deck[n].suit);  
            break;  
        case 13:  
            printf("%c of %s\n", 'K', deck[n].suit);  
            break;  
        case 14:  
            printf("%c of %s\n", 'A', deck[n].suit);  
            break;  
        default:  
            printf("%c of %s\n", deck[n].pips, deck[n].suit);  
            break; }  
}
```

```
7 of Hearts  
K of Spades  
2 of Spades  
A of Clubs  
8 of Diamonds
```

Structures within Structures

- As mentioned earlier, structures can have as members other structures. Say you wanted to make a structure that contained both date and time information. One way to accomplish this would be to combine two separate structures; one for the date and one for the time. For example,

```
struct date {  
    int month;  
    int day;  
    int year; };  
struct time {  
    int hour;  
    int min;  
    int sec; };  
struct date_time {  
    struct date today;  
    struct time now; };
```

- This declares a structure whose elements consist of two other previously declared structures.

Initializing Structures within Structures

- Initialization could be done as follows,

```
static struct date_time veteran = {{11,11,1918},{11,11,11}};
```

- which sets the **today** element of the structure **veteran** to the eleventh of November, 1918. The **now** element of the structure is initialized to eleven hours, eleven minutes, eleven seconds. Each item within the structure can be referenced if desired. For example,

```
++veteran.now.sec;  
if (veteran.today.month == 12)  
    printf("Wrong month! \n");
```

Pointers to Structures

- One can have pointer variable that contain the address of complete structures, just like with the basic data types. Structure pointers are declared and used in the same manner as “simple” pointers:

```
struct playing_card *card_pointer, down_card;  
card_pointer=&down_card;  
(*card_pointer).pips=8;  
(*card_pointer).suit="Clubs";
```

- The above code has **indirectly initialized** the structure `down_card` to the Eight of Clubs through the use of the pointer `card_pointer`.
- The type of the variable `card_pointer` is “pointer to a playing_card structure”.

Pointers to Structures: ->

- In C, there is a special symbol **->** which is used as a shorthand when working with pointers to structures. It is officially called the **structure pointer operator**. Its syntax is as follows:

`*(struct_ptr) .member` is the same as `struct_ptr->member`

- Thus, the last two lines of the previous example could also have been written as:

```
card_pointer->pips=8;  
card_pointer->suit="Clubs";
```

Question: What is the value of `*(card_pointer->suit+2)`?

Answer: `'u'`

- As with arrays, **use structure pointers as arguments to functions** working with structures. This is efficient, since only an address is passed and can also enable “call-by-reference” arguments.

Unions

- [Introduction to Unions](#)
- [Unions and Memory](#)
- [Unions Example](#)

Introduction to Unions

- Unions are C variables whose syntax look similar to structures, but act in a completely different manner. A union is **a variable that can take on different data types** in different situations. The union syntax is:

```
union tag_name {  
    type1 member1;  
    type2 member2;  
    ...  
};
```

- For example, the following code declares a union data type called `intfloat` and a union variable called `proteus`:

```
union intfloat {  
    float f;  
    int i;  
};  
union intfloat proteus;
```

Unions and Memory

- Once a union variable has been declared, the amount of memory reserved is just enough to be able to represent the **largest member**. (Unlike a structure where memory is reserved for **all** members).
- In the previous example, 4 bytes are set aside for the variable **proteus** since a **float** will take up 4 bytes and an **int** only 2 (on some machines).
- Data actually stored in a union's memory can be the data associated with **any** of its members. But **only one** member of a union can contain valid data at a given point in the program.
- It is the user's responsibility to keep track of which type of data has most recently been stored in the union variable.

Unions Example

- The following code illustrates the chameleon-like nature of the union variable **proteus** defined earlier.

```
#include <stdio.h>
main() {
    union intfloat {
        float f;
        int i;
    } proteus;
    proteus.i=4444    /* Statement 1 */
    printf("i:%12d f:%16.10e\n",proteus.i,proteus.f) ;
    proteus.f=4444.0; /* Statement 2 */
    printf("i:%12d f:%16.10e\n",proteus.i,proteus.f) ;
}
```

```
i:          4444 f:6.2273703755e-42
i: 1166792216 f:4.440000000e+03
```

- After **Statement 1**, data stored in **proteus** is an integer the the float member is full of junk.
- After **Statement 2**, the data stored in **proteus** is a float, and the integer value is meaningless.

File Input and Output

- [Introduction to File Input and Output](#)
- [Declaring FILE Variables](#)
- [Opening a Disk File for I/O](#)
- [Reading and Writing to Disk Files](#)
- [Closing a Disk File](#)
- [Additional File I/O Functions](#)
- [Sample File I/O Program](#)
- [Sample File I/O Program: main](#)
- [Sample File I/O Program: processfile](#)
- [Sample File I/O Program: getrecord](#)
- [Sample File I/O Program: printrecord](#)
- [Sample File I/O Program: sample session](#)

Introduction to File Input and Output

- So far, all the output (formatted or not) in this course has been written out to what is called **standard output** (which is usually the monitor). Similarly all input has come from **standard input** (usually associated with the keyboard). The C programmer can also read data directly from files and write directly to files. To work with files, the following steps must be taken:
 - 1 Declare **variables to be of type FILE**.
 - 2 Connect the internal **FILE** variable with an actual data file on your hard disk. This association of a **FILE** variable with a file name is done with the **fopen()** function.
 - 3 Perform I/O with the actual files using **fprint()** and **fscanf()** functions.
 - 4 Break the connection between the internal **FILE** variable and actual disk file. This disassociation is done with the **fclose()** function.

Declaring FILE variables

- Declarations of the file functions highlighted on the previous page must be included into your program. This is done in the standard manner by having

```
#include <stdio.h>
```

- as the first statement in your program.
- The first step in using files in C programs is to declare a file variable. This variable must be of type **FILE** (which is a predefined type in C) and it is a pointer variable. For example, the following statement

```
FILE *in_file;
```

- declares the variable **in_file** to be a “pointer to type **FILE**”.

Opening a Disk File for I/O

- Before using a **FILE** variable, it must be associated with a specific file name. The **fopen()** function performs this association and takes two arguments: 1) the pathname of the disk file, and 2) the access mode which indicates how the file is to be used. The following statement

```
in_file = fopen("myfile.dat", "r");
```

- connects the variable **in_file** to the disk file **myfile.dat** for **read** access. Thus, **myfile.dat** will **only** be read from. Two other access modes can be used:

"w"	indicating write-mode
"a"	indicating append_mode

Reading and Writing to Disk Files

- The functions **fprintf** and **fscanf** are provided by C to perform the analogous operations for the **printf** and **scanf** functions but on a file.
- These functions take an additional (first) argument which is the FILE pointer that identifies the file to which data is to be written to or read from. Thus the statement,

```
fscanf(in_file, "%f %d", &x, &m) ;
```

- will input -- from the file **myfile.dat** -- real and integer values into the variables **x** and **m** respectively.

Closing a Disk File

- The **fclose** function in a sense does the opposite of what the **fopen** does: it tells the system that we no longer need access to the file. This allows the operating system to cleanup any resources or buffers associated with the file.
- The syntax for file closing is simply

```
fclose(in_file) ;
```

Additional File I/O Functions

- Many of the specialized I/O functions for characters and strings that we have described in this course have analogs which can be used for file I/O. Here is a list of these functions

<i>Function</i>	<i>Result</i>
fgets	file string input
fputs	file string output
getc(file_ptr)	file character input
putc(file_ptr)	file character output

- Another useful function for file I/O is **feof()** which tests for the end-of-file condition. **feof** takes one argument -- the FILE pointer -- and returns a nonzero integer value (TRUE) if an attempt has been made to read past the end of a file. It returns zero (FALSE) otherwise. A sample use:

```
if (feof(in_file))  
    printf ("No more data \n");
```

Sample File I/O Program

- The program on the next few pages illustrates the use of file I/O functions. It is an inventory program that reads from the following file

```
lima beans
1.20
10
5
thunder tea
2.76
5
10
Greaters ice-cream
3.47
5
5
boneless chicken
4.58
12
10
```

- which contains stock information for a store. The program will output those items which need to be reordered because their quantity is below a certain limit

Sample File I/O Program: main

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
struct goods {
    char name[20];
    float price;
    int quantity;
    int reorder;
};
FILE *input_file;
void processfile(void);
void getrecord(struct goods *recptr);
void printrecord(struct goods record);

main() {
    char filename[40];
    printf("Example Goods Re-Order File Program\n");
    printf("Enter database file \n");
    scanf("%s",filename);
    input_file = fopen(filename, "r");
    processfile();
}
```

Sample File I/O Program: processfile

```
void processfile(void) {
    struct goods record;
    while (!feof(input_file)) {
        getrecord(&record);
        if (record.quantity <= record.reorder)
            printrecord(record);
    }
}
```

Sample File I/O Program: getrecord

```
void getrecord(struct goods *recptr) {
    int loop=0,number,toolow;
    char buffer[40],ch;
    float cost;
    ch=fgetc(input_file);
    while (ch!='\n') {
        buffer[loop++]=ch;
        ch=fgetc(input_file);
    }
    buffer[loop]=0;
    strcpy(recptr->name,buffer);
    fscanf(input_file,"%f",&cost);
    recptr->price = cost;
    fscanf(input_file,"%d",&number);
    recptr->quantity = number;
    fscanf(input_file,"%d",&toolow);
    recptr->reorder = toolow;
}
```


Sample File I/O Program: printrecord

```
void printrecord (struct goods record) { printf("\nProduct
    name \t%s\n",record.name); printf("Product price
    \t%f\n",record.price); printf("Product quantity
    \t%d\n",record.quantity); printf("Product reorder
    level \t%d\n",record.reorder);
}
```

Sample File I/O Program: sample session

```
Example Goods Re-Order File Program
Enter database file food.dat
```

```
Product name  thunder tea
Product price           2.76
Product quantity        5
Product reorder level           10
```

```
Product name  Greaters ice-cream
Product price           3.47
Product quantity        5
Product reorder level           5
```

Dynamic Memory Allocation

- [Introduction to Dynamic Memory Allocation](#)
- [Dynamic Memory Allocation: sizeof](#)
- [Dynamic Memory Allocation: calloc](#)
- [Dynamic Memory Allocation: free](#)

Introduction to Dynamic Memory Allocation

- A common programming problem is knowing how large to make arrays when they are declared. Consider a grading program used by a professor which keeps track of student information in structures. We want his program to be general-purpose so we need to make arrays large enough to handle the biggest possible class size:

```
struct student class[600];
```

- But when a certain upper-level class has only seven students, this approach can be inelegant and extremely wasteful of memory especially if the `student` structure is quite large itself.
- Thus, it is desirable to **create correct-sized array variables at runtime**. The C programming language allows users to dynamically allocate and deallocate memory when required. The functions that accomplish this are `calloc()` which allocates memory to a variable, `sizeof()`, which determines how much memory a specified variable occupies, and `free()`, which deallocates the memory assigned to a variable back to the system

Dynamic Memory Allocation: sizeof

- The `sizeof()` function **returns the memory size** (in bytes) of the requested variable type. This call should be used in conjunction with the `calloc()` function call, so that only the necessary memory is allocated, rather than a fixed size. Consider the following code fragment:

```
struct time {  
    int hour;  
    int min;  
    int sec;  
};  
int x;  
x=sizeof(struct time);
```

- `x` now contains how many bytes are taken up by a `time` structure (which turns out to be 12 on many machines). `sizeof` can also be used to determine the memory size of basic data type variables as well. For example, it is valid to write `sizeof(double)`.

Dynamic Memory Allocation: calloc

- The `calloc` function is used to **allocate storage to a variable** while the program is running. The function takes two arguments that specify the number of elements to be reserved, and the size of each element in bytes (obtained from `sizeof`). The function returns a pointer to the beginning of the allocated storage area in memory. The storage area is also initialized to zeros.

```
struct time *appt;  
appt = (struct time *) calloc(100,sizeof(struct time));
```

- The code `(struct time *)` is a type cast operator which converts the pointer returned from `calloc` to a pointer to a structure of type `time`. The above function call will allocate just enough memory for one hundred `time` structures, and `appt` will point to the first in the array. Now the array of `time` structures can be used, just like a statically declared array:

```
appt[5].hour=10;  
appt[5].min=30;  
appt[5].sec=0;
```

Dynamic Memory Allocation: free

- When the **variables are no longer required**, the space which was allocated to them by **calloc** should be returned to the system. This is done by,

```
free (appt) ;
```

Command-Line Arguments

- [Introduction to Command-Line Arguments](#)
- [Command-Line Arguments Example](#)
- [Command-Line Arguments Sample Session](#)

Introduction to Command-Line Arguments

- In every program you have seen so far, the **main** function has had no dummy arguments between its parentheses. The **main function is allowed to have dummy arguments** and they match up with command-line arguments used when the program is run.
- The two dummy arguments to the **main** function are called **argc** and **argv**.
 - **argc** contains the number of command-line arguments passed to the main program and
 - **argv[]** is an array of pointers-to-**char**, each element of which points to a passed command-line argument.

Command-Line Arguments Example

- A simple example follows, which checks to see if only a single argument is supplied on the command line when the program is invoked

```
#include <stdio.h>
main(int argc, char *argv[]) {
    if (argc == 2)
        printf("The argument supplied is %s\n", argv[1]);
    else if (argc > 2)
        printf("Too many arguments supplied.\n");
    else
        printf("One argument expected.\n");
}
```

- Note that ***argv[0]** is the program name itself, which means that ***argv[1]** is a pointer to the first “actual” argument supplied, and ***argv[n]** is the last argument. If no arguments are supplied, **argc** will be one. Thus for **n** arguments, **argc** will be equal to **n+1**.

Command-Line Arguments: Sample Session

- A sample session using the previous example follows:

```
#include <stdio.h>
main(int argc, char *argv[]) {
    if (argc == 2)
        printf("The argument supplied is %s\n", argv[1]);
    else if (argc > 2)
        printf("Too many arguments supplied.\n");
    else
        printf("One argument expected.\n");
}
```

```
a.out
One argument expected.
a.out help
The argument supplied is help
a.out help verbose
Too many arguments supplied.
```

Operator Precedence Table

	<i>Description</i>	<i>Represented by</i>
1	Parenthesis	() []
1	Structure Access	. ->
2	Unary	! ++ -- - * &
3	Multiply, Divide, Modulus	* / %
4	Add, Subtract	+ -
5	Shift Right, Left	>> <<
6	Greater, Less Than, etc.	> < ==> <=
7	Equal, Not Equal	== !=
8	Bitwise AND	&
9	Bitwise Exclusive OR	^
10	Bitwise OR	
11	Logical AND	&&
12	Logical OR	
13	Conditional Expression	? :
14	Assignment	= += -= etc
15	Comma	,