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.

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.

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 lets 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;
```

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;
```

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
```

Format Specifiers Table

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

Specifier	*Type*
`%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

Basic Output Examples

`printf("ABC");`	**ABC** (cursor after the C)
`printf("%d\n",5);`	**5** (cursor at start of next line)
`printf("%c %c %c",'A','B','C');`	**A B C**

`printf("From sea ");`
`printf("to shining ");`
`printf ("C");`
	From sea to shining C

`printf("From sea \n");`
`printf("to shining \n");`
`printf ("C");`
	From sea
	to shining
	C

`leg1=200.3; leg2=357.4;` `printf("It was %f miles",leg1+leg2);`	**It was 557.700012 miles**
`num1=10; num2=33;` `printf("%d\t%d\n",num1,num2);`	**10 33**
`big=11e+23;` `printf("%e \n",big);`	**1.100000e+24**
`printf("%c \n",'?');`	**?**
`printf("%d \n",'?');`	**63**
`printf("\007 That was a beep\n");`	try it yourself

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

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

`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

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");
}
```

`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-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;
```

`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"); }
```

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.

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**).

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 ***n*** characters of string
`strncmp`	Compares ***n*** characters of two strings
`strncpy`	Copies ***n*** characters of one string to another
`strnset`	Sets ***n*** 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

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

Function	*Operation*
`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**

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`**.

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
```

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.

char and int Formatted Output Example

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

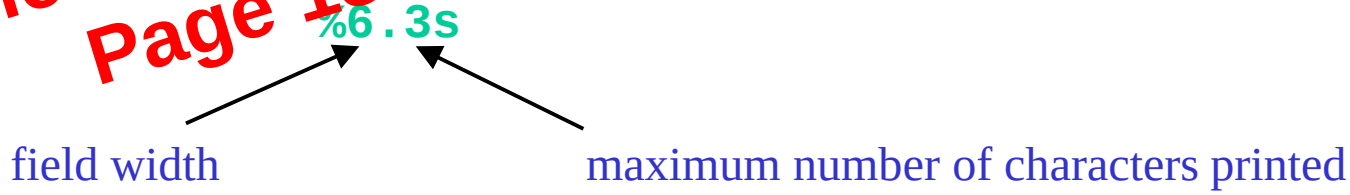

```
#include <stdio.h>
main() {
    char lett='w';
    int i=1,j=29;
    printf ("%c\n",lett);
    printf ("%4c\n",lett);
    printf ("%-3c\n\n",lett);
    printf ("%d\n",i);
    printf ("%d\n",j);
    printf ("%10d\n",j);
    printf ("%010d\n",j);
    printf ("%-010d\n",j);
    printf ("%2o\n",j);
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

```
w
   w
w

1
29
        29
0000000029
29
35
1d
```

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
```

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;
```

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, ad 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;
};
```

keyword

structure data type name

member name & type

- 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;
```

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(ptr);
```