

User Defined Data Types in C

C supports the idea of programmers creating their own data types.

Ordinal Data Types

Ordinal data types have the following characteristics:

1. There exists a smallest value in the set of values.
2. There exists a largest value in the set of values.
3. There exists an order of values from the smallest to the largest.

Example:

The “char” data type is an ordinal data type. Each character value has an integer value associated with it. There are 256 characters.

1. The smallest character value has the ASCII value of 0.
2. The largest character value has the ASCII value of 255.
3. The sequence of ASCII values is 0, 1, 2, ..., 253, 254, 255.

‘A’ == 65 ‘B’ == 66 ‘C’ == 67 etc...

Enumerated Data Types

Enumerated data types are a user defined ordinal data type. The main purpose of the enumerated data type is to allow numbers to be replaced by words. This is intended to improve the readability of programs.

Basic Format:

```
enum data_type_name { word1, word2, ..., word(n-1), word(n) };
```

OR

```
enum data_type_name { word1 = integer1, word2 = integer2, etc... };
```

Examples:

```
enum Boolean { FALSE, TRUE };
```

In the example above a user defined data type called Boolean has been defined. The new data type has two values: FALSE and TRUE. The word FALSE has the integer value of 0. The word TRUE has the integer value of 1.

If no integer values are specified then the left most word has integer value 0 and each one after that is incremented by one from that point. (0, 1, 2, 3, etc...) This also means that the left-most word is generally the smallest and the right-most word is generally the largest.

```
enum Boolean { FALSE = 0, TRUE = 1 };
```

In the example above a user defined data type called Boolean has been defined. The new data type has two values: FALSE and TRUE. The word FALSE has the integer value of 0. The word TRUE has the integer value of 1.

```
enum Weekdays { Monday = 1, Tuesday, Wednesday, Thursday, Friday };
```

In the example above a user defined data type called Weekdays has been defined. The new data type has five values: Monday, Tuesday, Wednesday, Thursday, and Friday. Monday is 1, Tuesday is 2, Wednesday is 3, Thursday is 4, and Friday is 5. Notice that the starting value is 1 in this example instead of 0.

```
enum Boolean Positive; /* uninitialized variable declaration */  
enum Weekdays Day = Wednesday; /* initialized variable declaration */
```

```
if ( Number > 0 )  
    Positive = TRUE;  
else  
    Positive = FALSE;
```

```
for ( Day = Monday; Day <= Friday; ++ Day )  
{  
    /* execute the body of loop */  
}
```

The disadvantage of enumerated data types is that normal input and output operations are not supported.

```
scanf ( “%d%c”, &Day, &Enter_Key );          /* integer based */
```

The defined values for the variable Day are: Monday, Tuesday, Wednesday, Thursday, and Friday. None of these values can be type in from the keyboard. Any integer value can be typed in even though the only defined values are 1 through 5.

```
printf ( “%d\n”, Monday );                   /* integer based */
```

The word Monday will not be displayed on the screen for output. Instead the integer value 1 will be displayed.

Type Definition Statement

The type definition statement is used to allow user defined data types to be defined using other already available data types.

Basic Format:

```
typedef existing_data_type new_user_define_data_type;
```

Examples:

```
typedef int Integer;
```

The example above simply creates an alternative data type name or “alias” called Integer for the built in data type called “int”. This is generally not a recommended use of the typedef statement.

```
typedef char Characters [ WORDSIZE ];        /* #define WORDSIZE 20 */
```

The example above creates a user defined data type called Characters. Characters is a data type that supports a list of “char” values. The fact that this user defined data type is an array of WORDSIZE is now built into the data type.

```
Characters One_String;                       /* [ ] and WORDSIZE are not specified */  
                                              /* this also applies to argument list use */
```

The previous example is a commonly used technique for creating a special type of arrays of characters called “Strings” or “Character Strings”.

A string value is different from an array of characters in that it can be processed as a list of individual characters like an array of characters or it can be processed as a string of characters or a single word unlike an array of characters. In order to be processed as a string value the list of characters must be terminated by placing a special character in the array just after the last character in the string value. This character is called the NULL character and is represented by ‘\0’ in C.

```
typedef char String [ WORDSIZE + 1 ];
```

The user defined data type called String has a maximum size of WORDSIZE + 1. The extra position is so that there is room to store the NULL character in addition to the other characters in the string value.

```
String Word_1;           /* uninitialized array of characters – no ‘\0’ */
String Word_2 = { '\0' }; /* initialized empty string – ‘\0’ position 0 */
String Word_3 = { "" };  /* initialized empty string – ‘\0’ position 0 */
String Word_4 = { 'H', 'I', '\0' }; /* initialized ‘H’ ‘I’ ‘\0’ – positions 0, 1, 2 */
String Word_5 = { "HI" }; /* initialized ‘H’ ‘I’ ‘\0’ – positions 0, 1, 2 */
```

Using the “...” double quotation marks tells the compiler that you are using a string value. When assigning string values to string variables using the double quotation marks the NULL character is automatically inserted at the end of the string value.

```
Word_1 = "Hello"; /* ‘H’ ‘e’ ‘l’ ‘l’ ‘o’ ‘\0’ – positions 0, 1, 2, 3, 4, 5 */
/* string assignment not supported – strcpy required */
scanf( "%s", Word_3 ); /* Stores character sequence typed at the keyboard */
/* NULL character is added to the end */
fscanf( InFile, "%s", Word_4 ); /* Value read from external file */
/* Terminates with white space or end-of-line */

printf ( "Hello" );

printf ( "%s", Word_4 ); /* Works the same way as outputting a string value */
```

In order to correctly process string values in C you must include “**string.h**”. This contains a special set of functions/operations that work with arrays of characters that contain string values that are terminated by the NULL character.

```
strcpy ( string_value_variable, string_value );    /* String Copy */  
  
strcpy ( Word_1, “Happy” );                      /* Word_1 = “Happy”; */  
strcpy ( Word_1, Word_5 );                        /* Word_1 = Word_5; */
```

The strcpy function is required because the assignment of any kind of array is not supported with C.

```
int strcmp ( string_value1, string_value2 );    /* String Compare */
```

The strcmp function returns a value based upon the result of the comparison of the two string values.

Negative – if the first string value is less than the second string value.

```
Result = strcmp ( “Apple”, “Baker” );          /* ‘A’ (65) < ‘B’ (66) */
```

Zero – if the first string value is equal to the second string value.

```
Result = strcmp ( “Apple”, “Apple” );         /* all characters match */
```

Positive – if the first string value is greater than the second string value.

```
Result = strcmp ( “Baker”, “Apple” );        /* ‘B’ (66) > ‘A’ (65) */
```

The strcmp function is required because relational operations are not supported with strings.

```
int strlen ( string_value );                  /* String Length */
```

The strlen function returns the length of the string value. The strlen function returns the number of individual characters contained in the string value. The length is indicated by the position that the NULL character is stored in.

Arrays of Strings

C supports the use of character string values with arrays. Each string that is stored in the array can be processed as either an individual string value or as part of a list of string values.

```
#define WORDSIZE 20 /* macros must be used */
#define LISTSIZE 35 /* Maximum string size */
/* Maximum list size */

typedef char Char20 [ WORDSIZE+1 ]; /* Global string data type */

Char20 Strings [ LISTSIZE ]; /* Array of string values */
/* Array of string pointers */

Like with any other arrays use the name of the array variable and the position value
to access the individual string values.

strcpy ( Strings [ 0 ], "SDSU" ); /* Strings [ 0 ] = "SDSU"; */

scanf ( "%s", Strings [ 5 ] ); /* address operator not used */
/* Strings [ 5 ] is a pointer */

printf ( "%s", Strings [ 10 ] );

if ( strcmp ( Strings [ Comp - 1 ], Strings [ Comp ] ) > 0 ) {
    Swap_Char20 ( Strings [ Comp - 1 ], Strings [ Comp ] ); /* function call */
}

void Swap_Char20 ( Char20 First, Char20 Second ) { /* definition */
    Char20 Temp;
    strcpy ( Temp, First );
    strcpy ( First, Second );
    strcpy ( Second, Temp );
    return;
}

void Swap_Char20 ( Char20, Char20 ); /* declaration */

Get_Strings ( Strings, &Size ); /* function call */

void Get_Strings ( Char20 [], int * ); /* declaration */
```