# Chapter 10

# C Structures, Typedef, and Enumerations

# Chapter 10 - C Structures, Typedef, and Enumerations

# Structures

# Outline of Structure Topics

- Basic struct
- Pointer to struct
- Array of struct
- Nested structs
- Array of nested structs

# 10.1 Introduction

- Structures
  - Collections of related variables (aggregates) under one name
    - Can contain variables of different data types
  - Commonly used to define records to be stored in files
  - When combined with pointers, can create data structures such as linked lists, stacks, queues, and trees

# 10.2 Structure Definitions

- Example

```
struct student
{
    int num;
    char name[20];
};
```

  - `struct` introduces the definition for structure `student`
  - `student` is the structure name and will be used to declare variables of that structure type

    ```
    struct student Ogr;
    ```

  - `student` contains two member variables
    - These members are `num` and `name`

# 10.2 Structure Definitions

- `struct` information
  - A `struct` cannot contain an instance of itself
  - Can contain a member that is a pointer to the same structure type (Example: Linked List)
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables
  - A structure can contain any other type of structures

# 10.2 Structure Definitions

- Definitions
  - Defined like other variables:

```
struct student Ogr1, OgrListe[ 60 ], *Ptr;
```

  - Alternative Method: We can use a comma separated list.

```
struct student {
   int num;
   char name[20];
} Ogr1, OgrListe[ 60 ], *Ptr;
```

# 10.2 Structure Definitions

- Valid Operations
  - Assigning a structure to a structure of the same type
  - Taking the address (&) of a structure
  - Accessing the members of a structure
  - Using the `sizeof` operator to determine the size of a structure

# 10.3 Initializing Structures

- Initializer lists
  - Example:

```
struct student Ogr1 = {40010478,“Mehmet Uslu" };
```

- Assignment statements
  - Could also define and initialize Ogr1 as follows:

```
struct student  Ogr1;
Ogr1.num = 40010478;
strcpy(Ogr1.name, "Mehmet Uslu“);
```

  - Copying (i.e. assignment) example:

```
struct student Ogr2 = Ogr1;  // Copies entire struct
```

# Struct Membership Operators

| OPERATOR | NOTATION | WHEN USED |
|---|---|---|
| • | Dot Operator | Used to access member item of a normal struct variable. |
| -> | Arrow Operator | Used to access member item of a pointed struct variable. |

# 10.4 Accessing Members of Structures

– Dot operator ( . ) used with structure variables

```
struct student Ogr1;


printf("Enter student number and name :");
scanf("%d  %s", &Ogr1.num, Ogr1.name );


printf("%d  %s \n", Ogr1.num, Ogr1.name );
```

# 10.4 Accessing Members of Structures

– Arrow operator (**->**) used with pointers to structure variables

```
struct student *Ptr;
Ptr = &Ogr1;  // Get address of Ogr1 struct variable


printf("Enter student number and name :");
scanf("%d  %s", &(Ptr->num), Ptr->name );


printf("%d  %s \n", Ptr->num, Ptr->name );
```

– `Ptr->num` is equivalent to following notation:
`( *Ptr ).num`

# Example : Using Struct and Pointer to struct

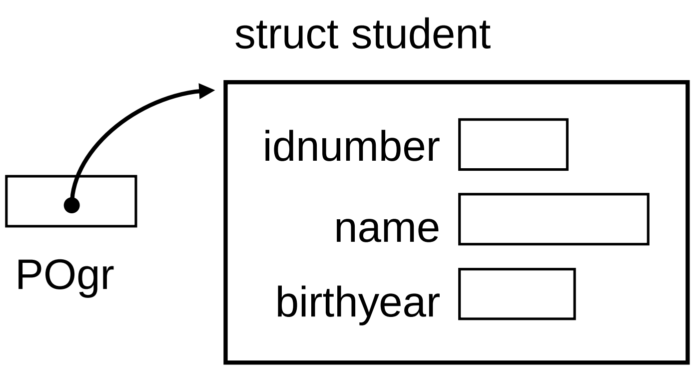
```
/*  Using the structure member and structure pointer operators */
#include <stdio.h>
#include <string.h>

// student structure definition
struct student {
     int num; // define student number
     char name[20]; // define student name
}; // end structure student

int main() {
    struct student a;     // define struct a
    struct student *aPtr; // define a pointer to student struct

    // place data into student structure
    a.num = 40010478;
    strcpy(a.name , "Mehmet Uslu");

    aPtr = &a; // assign address of a to aPtr
    printf( "%d %s \n", a.num, a.name);
    printf( "%d %s \n", aPtr->num, aPtr->name);
    printf( "%d %s \n", (*aPtr).num, (*aPtr).name );
} // end main
```

Program Output

```
40010478 – Mehmet Uslu
40010478 – Mehmet Uslu
40010478 – Mehmet Uslu
```

# Example: Pointer to dynamically allocated Struct

```
struct student {
  int   idnumber;
  char  name[20];
  int   birthyear;
};

struct student  * POgr;

// Dynamic memory allocation:
POgr = malloc(sizeof(struct student));

printf("Enter ID, name, birthyear :");
scanf("%d %s %d", &(POgr->idnumber),
                  POgr->name,
                  &(POgr->birthyear) );

printf("%d %s %d \n", POgr->idnumber,
                      POgr->name,
                      POgr->birthyear);
```



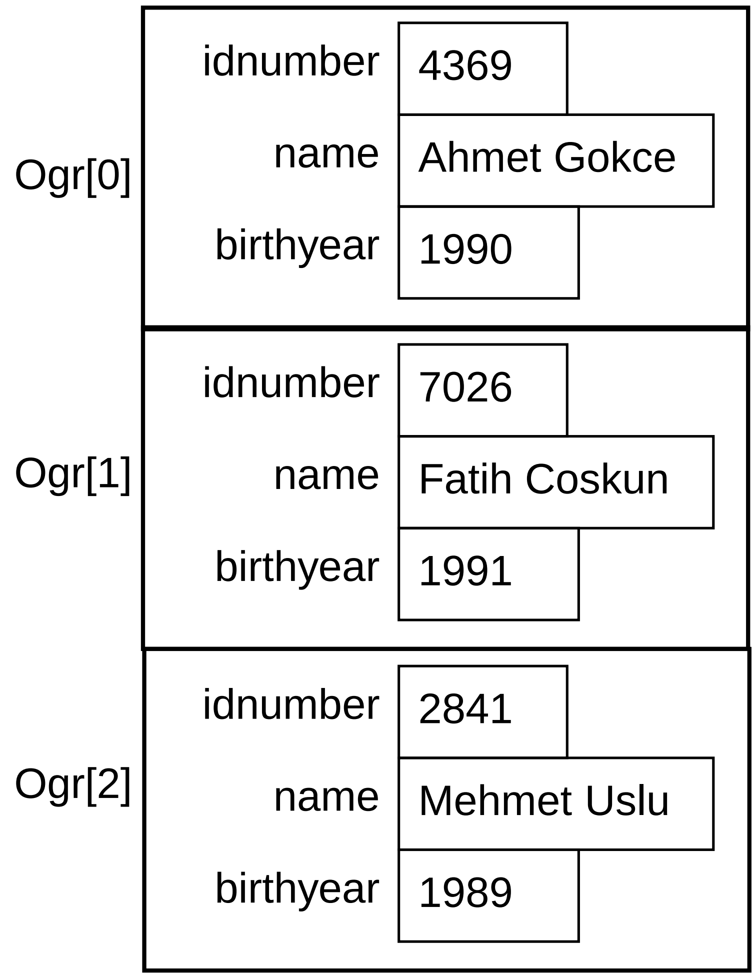
# Example: Copying a Struct variable

```
struct student  Ogr1 = {40394869, "Mehmet Uslu", 1990};
struct student  Ogr2, Ogr3;


// Copy members of Ogr1 to Ogr2 one by one:
Ogr2.idnumber  = Ogr1.idnumber;
Ogr2.birthyear = Ogr1.birthyear;
strcpy(Ogr2.name , Ogr1.name);


// Copy entire Ogr1 to Ogr3
Ogr3 = Ogr1; // Easy method for structure copying


printf("%d %s %d \n", Ogr1.idnumber, Ogr1.name, Ogr1.birthyear);
printf("%d %s %d \n", Ogr2.idnumber, Ogr2.name, Ogr2.birthyear);
printf("%d %s %d \n", Ogr3.idnumber, Ogr3.name, Ogr3.birthyear);
```

# Example: Initializing an Array of Struct

```c
#define N 3 // Number of persons

struct  student  Ogr[N] = {
   {443369, "Ahmet Gokce", 1990},
   {704326, "Fatih Coskun",1991},
   {221841, "Mehmet Uslu", 1989} };


 for (i=0; i < N; i++)
   printf("%d %s %d \n", Ogr[i].idnumber,
                         Ogr[i].name,
                         Ogr[i].birthyear);
```

Struct student



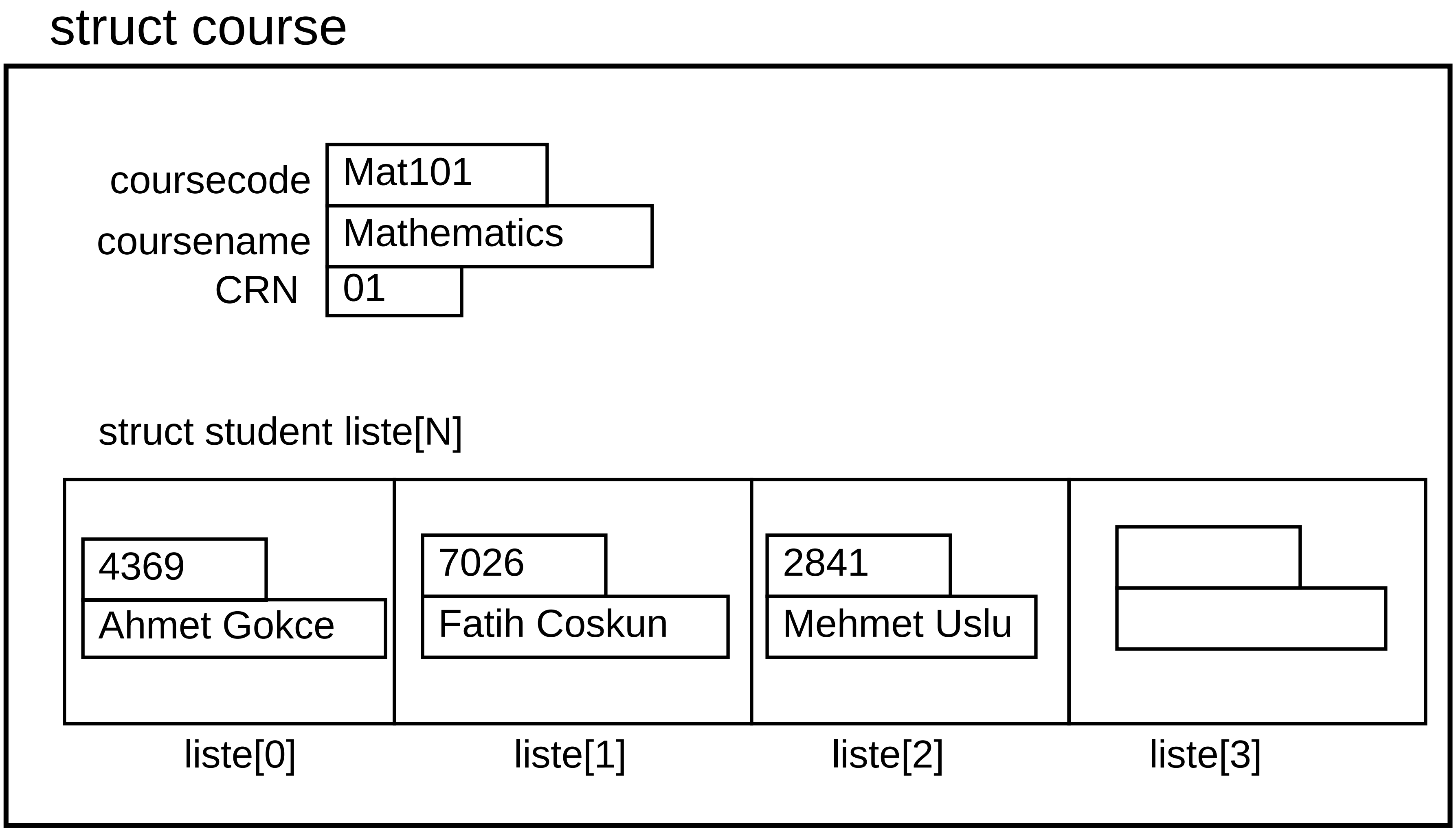
# Example: Inputting an Array of Struct

```
#define N 3 // Number of persons

struct  student  Ogr[N];

 for (i=0; i < N; i++)
 {
   printf("Enter ID,name,birthyear of %d.person :“,i+1);

   scanf("%d %s %d", &Ogr[i].idnumber,
                     Ogr[i].name,
                     &Ogr[i].birthyear);
 }
```

# Example: Copying an Array of Struct

```
#define N 3 // Number of persons

struct  student  liste1[N] ={ {4369, "Ahmet Gokce", 1990},
                              {7026, "Fatih Coskun",1991},
                              {2841, "Mehmet Uslu", 1989} };

 struct  student  liste2[N];


 for (i=0; i < N; i++)
 {
   liste2[i] = liste1[i]; // Easy copying of element i

   printf("%d %s %d \n", liste2[i].idnumber,
                         liste2[i].name,
                         liste2[i].birthyear);
 }
```

# Example : Nested Structs



# Example: Nested Structs

```
#define N 100 // Maximum number of students in course section


struct student {
  int   stunumber;
  char  stuname[20];
};


struct course {
  char  coursecode[10];
  char  coursename[30];
  int   CRN;  // Section number
  struct student liste[N];  // Registered students
};
```

# Example: Initializing Nested Structs

```
struct course Sube1 = {
   "Mat101", "Mathematics", 01,
   { 4369,"Ahmet Gokce",
     7026,"Fatih Coskun",
     2841,"Mehmet Uslu" }
};


struct course Sube2 = {
   "Mat101", "Mathematics", 02,
   { 6283,"Kemal Yılmaz",
     1194,"Bulent Aktas" }
};
```

# Example: Printing Nested Structs

```
int Count, i;

printf("COURSE CODE : %s \n", Sube1.coursecode);
printf("COURSE CRN  : %d \n", Sube1.CRN);
printf("COURSE NAME : %s \n", Sube1.coursename);
printf(“LIST OF STUDENTS: \n");

// Calculate number of students in Sube1.
Count = sizeof(Sube1.liste) / sizeof(struct student);


for (i=0; i < Count; i++)
{
   printf("%d %s \n", Sube1.liste[i].stunumber,
                      Sube1.liste[i].stuname);
}
```

# Example: Array of Nested Structs

```
#define M 3 // Maximum number of course sections
struct course Sube[M]; // Array of sections
int Count, i, j;

for (i=0; i < M; i++)
{
  printf("COURSE CODE : %s \n", Sube[i].coursecode);
  printf("COURSE CRN  : %d \n", Sube[i].CRN);
  printf("COURSE NAME : %s \n", Sube[i].coursename);
  printf("LIST OF STUDENTS: \n");

  count = sizeof(Sube[i].liste)  /  sizeof(struct student);

  for (j=0; j < Count; j++)
  {
     printf("%d %s \n", Sube[i].liste[j].stunumber,
                        Sube[i].liste[j].stuname);
  } // end inner loop

} // end outer loop
```

# Typedef

# Example: Simple typedef

```
#include <stdio.h>

typedef int Tamsayi;          //Defines a synonym
typedef float Kesirlisayi;    //Defines a synonym

int main()
{
  Tamsayi  a=5;
  Kesirlisayi  b = 7.4;

  printf("a = %d   b = %f \n", a ,b);
}
```

# 10.6 typedef

- `typedef`
  - Creates synonyms (aliases) for previously defined data types
  - Use `typedef` to create shorter type names
  - `typedef` does not create a new data type
    - Only creates an alias
  - Example: Define a new type name TStudent as a synonym for type `struct student`.

```
typedef struct student TStudent;
TStudent  Ogr; // Define a variable

Ogr.idnumber = 1234;
strcpy(Ogr.name , "Mehmet Uslu");
```

# 10.6 typedef

- Example:

  Define a new type name **TStuPtr** as a synonym for type
  struct student *

```
typedef struct student *TStuPtr;
TStuPtr  POgr; // Define a pointer variable


POgr = malloc(sizeof(struct student));

POgr->idnumber = 1234;
strcpy(Pogr->name , "Mehmet Uslu");
```

# Enumeration

# 10.10 Enumeration Constants

- Enumeration
  - Set of integer constants represented by identifiers
  - Enumeration constants are like symbolic constants whose values are automatically set
    - Values start at `0` and are incremented by `1`
    - Values can be set explicitly with `=`
    - Need unique constant names
  - Example:

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC};
```

    - Creates a new type `enum` Months in which the identifiers are set to the integers `1` to `12`
  - Enumeration variables can only assume their enumeration constant values (not the integer representations)

# Example : Enumeration

```c
// Fig. 10.18: fig10_18.c
// Using an enumeration
#include <stdio.h>

// enumeration constants represent months of the year
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // end enum months

int main()
{
   enum months month; // can contain any of the 12 months

   // initialize array of pointers
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   // loop through months
   for ( month = JAN; month <= DEC; ++month ) {
      printf( "%2d%11s\n", month, monthName[ month ] );
   } // end for
} // end main
```

Program Output

```
 1     January
 2    February
 3       March
 4       April
 5         May
 6        June
 7        July
 8      August
 9   September
10     October
11    November
12    December
```

# Alternative method to Enumaration

- The following method also defines constant symbols.
- Enumaration method is more effective.

```
#define JAN 1
#define FEB 2
#define MAR 3
#define APR 4
#define MAY 5
#define JUN 6
```

```
#define JUL 7
#define AUG 8
#define SEP 9
#define OCT 10
#define NOV 11
#define DEC 12
```

# Bit Manipulations

# 10.8 Bit Manipulations

- All data are represented internally as sequences of bits
  - Each bit can be either 0 or 1
  - Sequence of 8 bits forms a byte

- ***APPLICATIONS:***

***Bitwise operators are mostly used in Operating Systems programming for low level operations.***

# 10.8 Bitwise Operators

| Operator | | Description |
|---|---|---|
| & | bitwise AND | The bits in the result are set to 1 if the corresponding bits in the two operands are both 1. |
| \| | bitwise inclusive OR | The bits in the result are set to 1 if at least one of the corresponding bits in the two operands is 1. |
| ^ | bitwise exclusive OR | The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1. |
| << | left shift | Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits. |
| >> | right shift | Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine dependent. |
| ~ | one's complement | All 0 bits are set to 1 and all 1 bits are set to 0. |

Fig. 10.6 The bitwise operators.

# Bitwise AND Operator ( & )

| Bit 1 | Bit 2 | Bit 1 & Bit 2 |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Results of combining two bits with the bitwise AND operator &.

# Bitwise OR Operator
# ( | )

| Bit 1 | Bit 2 | Bit 1 \| Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Results of combining two bits with the bitwise inclusive OR operator |.

# Bitwise XOR Operator ( ^ )

| Bit 1 | Bit 2 | Bit 1 ^ Bit 2 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Results of combining two bits with the bitwise exclusive OR operator ^.

# Bitwise Complement Operator ( ~ )

| Bit | ~ Bit |
|---|---|
| 0 | 1 |
| 1 | 0 |

Result of complement operator **~** on a bit.

# 10.8 Bitwise ASSIGNMENT Operators

| Operators | Description |
| --- | --- |
| &= | Bitwise AND assignment operator. |
| \|= | Bitwise inclusive OR assignment operator. |
| ^= | Bitwise exclusive OR assignment operator. |
| <<= | Left-shift assignment operator. |
| >>= | Right-shift assignment operator. |

Fig. 10.14 The bitwise assignment operators.

# 10.8 Operator Precedences

| Operator | Associativity | Type |
|---|---|---|
| () [] . -> | left to right | Highest |
| + - ++ -- ! & * ~ sizeof (type) | right to left | Unary |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| << >> | left to right | shifting |
| < <= > >= | left to right | relational |
| == != | left to right | equality |
| & | left to right | bitwise AND |
| ^ | left to right | bitwise OR |
| \| | left to right | bitwise OR |
| && | left to right | logical AND |
| \|\| | left to right | logical OR |
| ?: | right to left | conditional |
| = += -= *= /= &= \|= ^= <<= >>= %= | right to left | assignment |
| , | left to right | comma |

Fig. 10.15 Operator precedence and associativity.

# Example : Printing bits

Part 1 of 2

```
// Fig. 10.7: fig10_07.c
// Displaying an unsigned int in bits
#include <stdio.h>

void displayBits( unsigned int value ); // prototype

int main()
{
   unsigned int x; // variable to hold user input

   printf( "%s", "Enter a nonnegative int: " );
   scanf( "%u", &x );

   displayBits( x );
} // end main
```

Part 2 of 2

```c
// display bits of an unsigned int value
void displayBits( unsigned int value )
{
   unsigned int c; // counter

   // define displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;

   printf( "%10u = ", value );

   // loop through bits
   for ( c = 1; c <= 32; ++c ) {
      putchar( value & displayMask ? '1' : '0' );
      value <<= 1; // shift value left by 1

      if ( c % 8 == 0 ) { // output space after 8 bits
         putchar( ' ' );
      } // end if
   } // end for

   putchar( '\n' );
} // end function displayBits
```

displayMask = $2^{31}$ = 2147483648 = 10000000 00000000 00000000 00000000

Program Output

```
Enter an unsigned integer: 65000

  65000 = 00000000  00000000  11111101  11101000
```

- The following definitions have the same meaning:

```
unsigned  displayMask;
unsigned long int  displayMask;
```

- The following statements have the same results:

```
displayMask = 1 << 31; // Bitwise shift
displayMask = 2147483648;  // Decimal
displayMask = 0x80000000;  // Hexadecimal
```

- The following statements have the same results:

```
value  <<=  1;
value  =  value  <<  1;
```

# Example: Bitwise Shift

```
#include <stdio.h>

int main()
{
 int Sayi=10, i;

 printf("Sayi = %d \n", Sayi);

 for (i=1; i <= 50; i++)
 {
   Sayi = Sayi << 1;
   printf("%d.shift sonunda Sayi=%d \n", i, Sayi);
 }

}
```

Program Output

```
Sayi = 10
1.shift sonunda Sayi=20
2.shift sonunda Sayi=40
3.shift sonunda Sayi=80
4.shift sonunda Sayi=160
5.shift sonunda Sayi=320
6.shift sonunda Sayi=640
7.shift sonunda Sayi=1280
8.shift sonunda Sayi=2560
9.shift sonunda Sayi=5120
10.shift sonunda Sayi=10240
11.shift sonunda Sayi=20480
12.shift sonunda Sayi=40960
13.shift sonunda Sayi=81920
14.shift sonunda Sayi=163840
15.shift sonunda Sayi=327680
16.shift sonunda Sayi=655360
17.shift sonunda Sayi=1310720
18.shift sonunda Sayi=2621440
19.shift sonunda Sayi=5242880
20.shift sonunda Sayi=10485760
21.shift sonunda Sayi=20971520
22.shift sonunda Sayi=41943040
23.shift sonunda Sayi=83886080
24.shift sonunda Sayi=167772160
25.shift sonunda Sayi=335544320
```

```
26.shift sonunda Sayi=671088640
27.shift sonunda Sayi=1342177280
28.shift sonunda Sayi=-1610612736
29.shift sonunda Sayi=1073741824
30.shift sonunda Sayi=-2147483648
31.shift sonunda Sayi=0
32.shift sonunda Sayi=0
33.shift sonunda Sayi=0
34.shift sonunda Sayi=0
35.shift sonunda Sayi=0
36.shift sonunda Sayi=0
37.shift sonunda Sayi=0
38.shift sonunda Sayi=0
39.shift sonunda Sayi=0
40.shift sonunda Sayi=0
41.shift sonunda Sayi=0
42.shift sonunda Sayi=0
43.shift sonunda Sayi=0
44.shift sonunda Sayi=0
45.shift sonunda Sayi=0
46.shift sonunda Sayi=0
47.shift sonunda Sayi=0
48.shift sonunda Sayi=0
49.shift sonunda Sayi=0
50.shift sonunda Sayi=0
```

# Example : Bitwise And, Or, Xor, Complement Operators



Part 1 of 2

```c
// Fig. 10.9: fig10_09.c
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR and bitwise complement operators
#include <stdio.h>

void displayBits( unsigned int value ); // prototype

int main()
{
   unsigned int number1;
   unsigned int number2;
   unsigned int mask;
   unsigned int setBits;

   // demonstrate bitwise AND (&)
   number1 = 65535;
   mask = 1;
   puts( "The result of combining the following" );
   displayBits( number1 );
   displayBits( mask );
   puts( "using the bitwise AND operator & is" );
   displayBits( number1 & mask );
```

Part 2 of 2

```
 // demonstrate bitwise inclusive OR (|)
   number1 = 15;
   setBits = 241;
   puts( "\nThe result of combining the following" );
   displayBits( number1 );
   displayBits( setBits );
   puts( "using the bitwise inclusive OR operator | is" );
   displayBits( number1 | setBits );

   // demonstrate bitwise exclusive OR (^)
   number1 = 139;
   number2 = 199;
   puts( "\nThe result of combining the following" );
   displayBits( number1 );
   displayBits( number2 );
   puts( "using the bitwise exclusive OR operator ^ is" );
   displayBits( number1 ^ number2 );

   // demonstrate bitwise complement (~)
   number1 = 21845;
   puts( "\nThe one's complement of" );
   displayBits( number1 );
   puts( "is" );
   displayBits( ~number1 );
} // end main
```

## Program Output

```
The result of combining the following
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

# Example : Bitwise Left and Right Shift Operators

```c
// Fig. 10.13: fig10_13.c
// Using the bitwise shift operators
#include <stdio.h>

void displayBits( unsigned int value ); // prototype

int main( void )
{
   unsigned int number1 = 960; // initialize number1

   // demonstrate bitwise left shift
   puts( "\nThe result of left shifting" );
   displayBits( number1 );
   puts( "8 bit positions using the left shift operator << is" );
   displayBits( number1 << 8 );

   // demonstrate bitwise right shift
   puts( "\nThe result of right shifting" );
   displayBits( number1 );
   puts( "8 bit positions using the right shift operator >> is" );
   displayBits( number1 >> 8 );
} // end main
```

Program Output

```
The result of left shifting
4294967295 = 11111111 11111111 11111111 11111111

8 bit positions using the left shift operator << is
4294967040 = 11111111 11111111 11111111 00000000


The result of right shifting
4294967295 = 11111111 11111111 11111111 11111111

8 bit positions using the right shift operator >> is
16777215 = 00000000 11111111 11111111 11111111
```