Chapter 8

User-Defined Data Types and Variables

Learning Objectives

- Learn about the user-defined data type called structure and its tag, members, and variables
- Access, initialize, and copy structures and their members
- Understand nesting of structures
- Create and initialize arrays of structures
- Use structures as function arguments and return values
- Learn about union data types
- Understand enumeration data types
- Get acquainted with bit fields

Introduction

- C provides facilities to construct user-defined data types from the fundamental data types.
- A user-defined data type may also be called a derived data type.
  - The array type is a derived data type that contains only one kind of fundamental data type defined in C.
- Such non-homogeneous data cannot be grouped to form an array.
- C provides features to pack heterogeneous data in one group, bearing a user-defined data type name, and forming a conglomerate data type called the ‘structure’ that is capable of holding data of existing types.

Key Words

- **Accessing a structure member:** The act of handling any member of a structure for the purpose of assigning a value or using the member in any expression.
- **Arrays of structures:** It refers to the “structure variable” when it is an array of objects, each of which contains the member elements declared within the structure construct.
- **Instance variable:** One of the named pieces of data that make up a structure.

Key Words

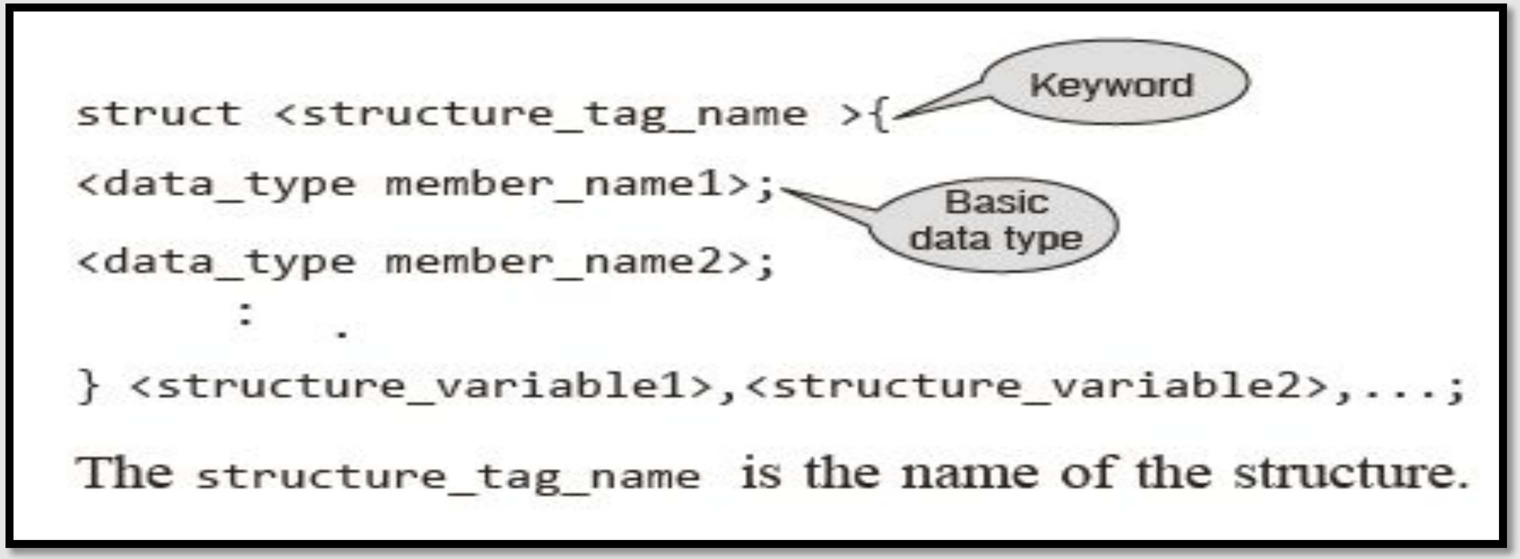
- **Non-homogeneous data**: Data of different types such as integer, float, character, etc.
- **Structure:** A collection of data grouped together and treated as a single object.
- **Type template:** A document or file having a preset format, used as a starting point for a particular application so that the format does not have to be recreated each time it is used.
- **Initialization of structure:** Assigning values to members of an instance variable.

Structure

- A structure is a collection of variables under a single name.
- These variables can be of different types, and each has a name that is used to select it from the structure.
- There can be structures within structures, which is known as nesting of structures.
- Arrays of structures can be formed and initialized as required. Pointers may also be used with structures.
- Structures may be passed as function arguments and they may also be returned by functions.

Declaring Structures and Structure Variables

- A structure is declared by using the keyword struct followed by an optional structure tag followed by the body of the structure.
  - The variables or members of the structure are declared within the body.
  - The general format of declaring a simple structure is given as follows.



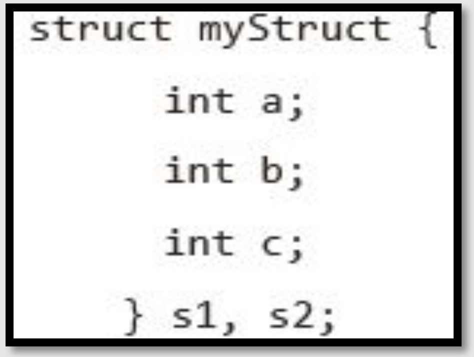
```
struct <structure_tag_name >{
<data_type member_name1>;
<data_type member_name2>;
      :    .
} <structure_variable1>,<structure_variable2>,...;
```

Keyword
Basic data type

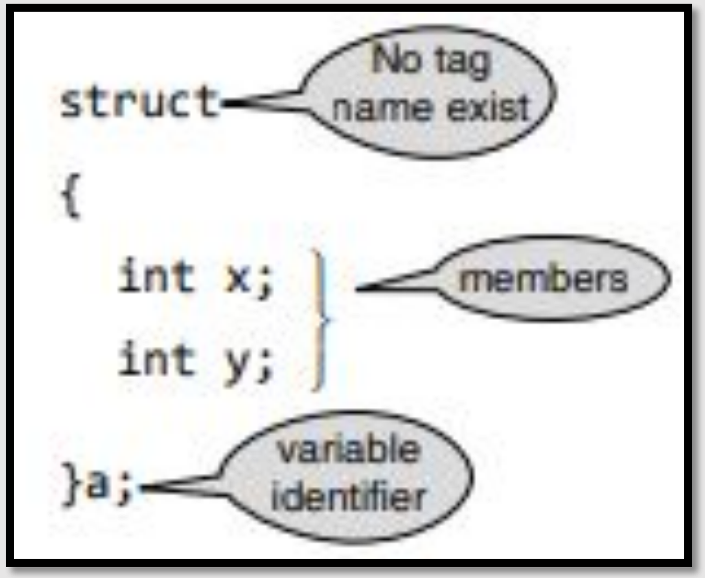
The structure_tag_name is the name of the structure.

Declaring Structures and Structure Variables

- The structure_tag_name is the name of the structure. The structure_variables are the list of variable names separated by commas.
- There are three different ways to declare and/or define a structure. These are
  - Variable structure
  - Tagged structure
  - Type-defined structure
-

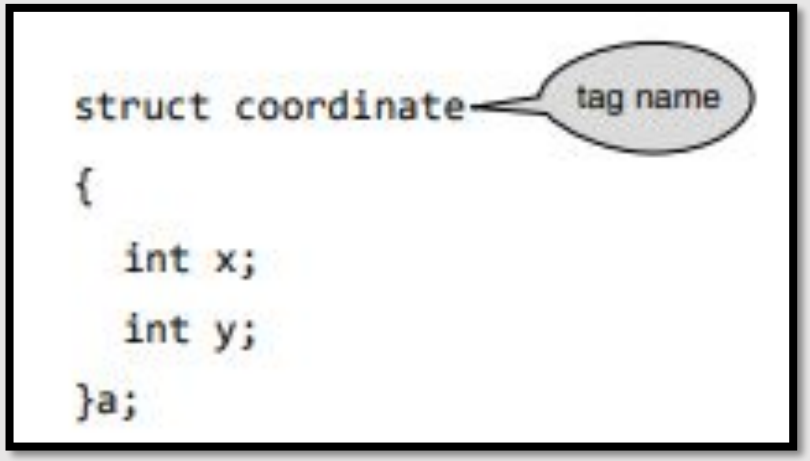


```
struct myStruct {
        int a;
        int b;
        int c;
    } s1, s2;
```



```
struct
{
    int x;
    int y;
}a;
```

 Variable



```
struct coordinate
{
    int x;
    int y;
}a;
```

 Tagged

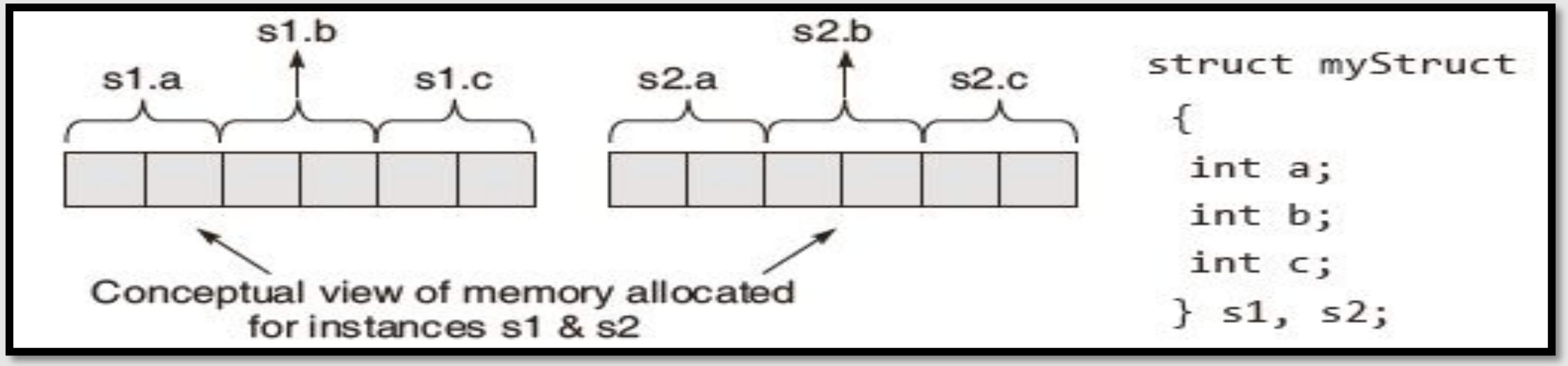
Declaring Structures and Structure Variables

- A structure can be defined as a user-defined data type that is capable of holding heterogeneous data of basic data type.
- The structure is simply a type template with no associate storage.
- The proper place for structure declarations is in the global area of the program before main().
- It is not possible to compare structures for equality using ‘==’, nor is it possible to perform arithmetic on structures.

Accessing the Members of a Structure

- The members of a structure can be accessed in three ways.
  - One of the ways consists of using '.', which is known as the 'dot operator'.
  - The members are accessed by relating them to the structure variable with a dot operator.
- The general form of the statement for accessing a member of a structure is as follows:

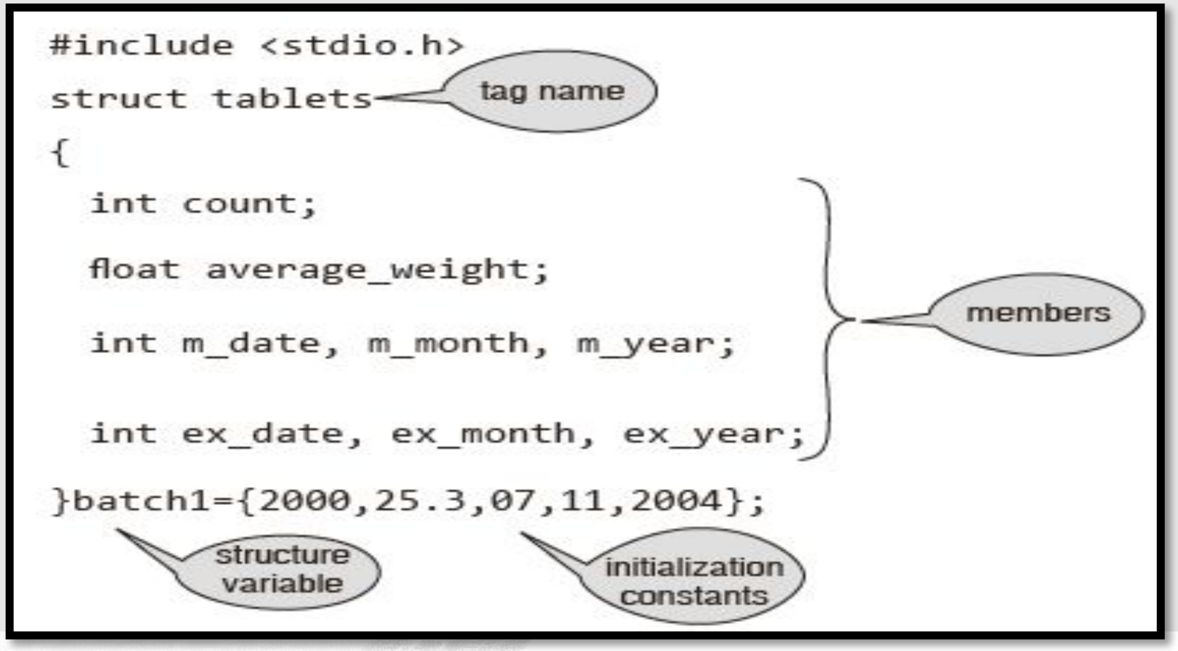
  < structure_variable >**.< member_name > ;**



```
struct myStruct
 {
  int a;
  int b;
  int c;
 } s1, s2;
```

Initialization of Structures

- Structures that are not explicitly initialized by the programmer are, by default, initialized by the system.
  - In most of the C compilers, for integer and float data type members, the default value is zero .
  - For char and string type members the default value is ‘\0’.



```
#include <stdio.h>
struct tablets
{
  int count;
  float average_weight;
  int m_date, m_month, m_year;
  int ex_date, ex_month, ex_year;
}batch1={2000,25.3,07,11,2004};
```

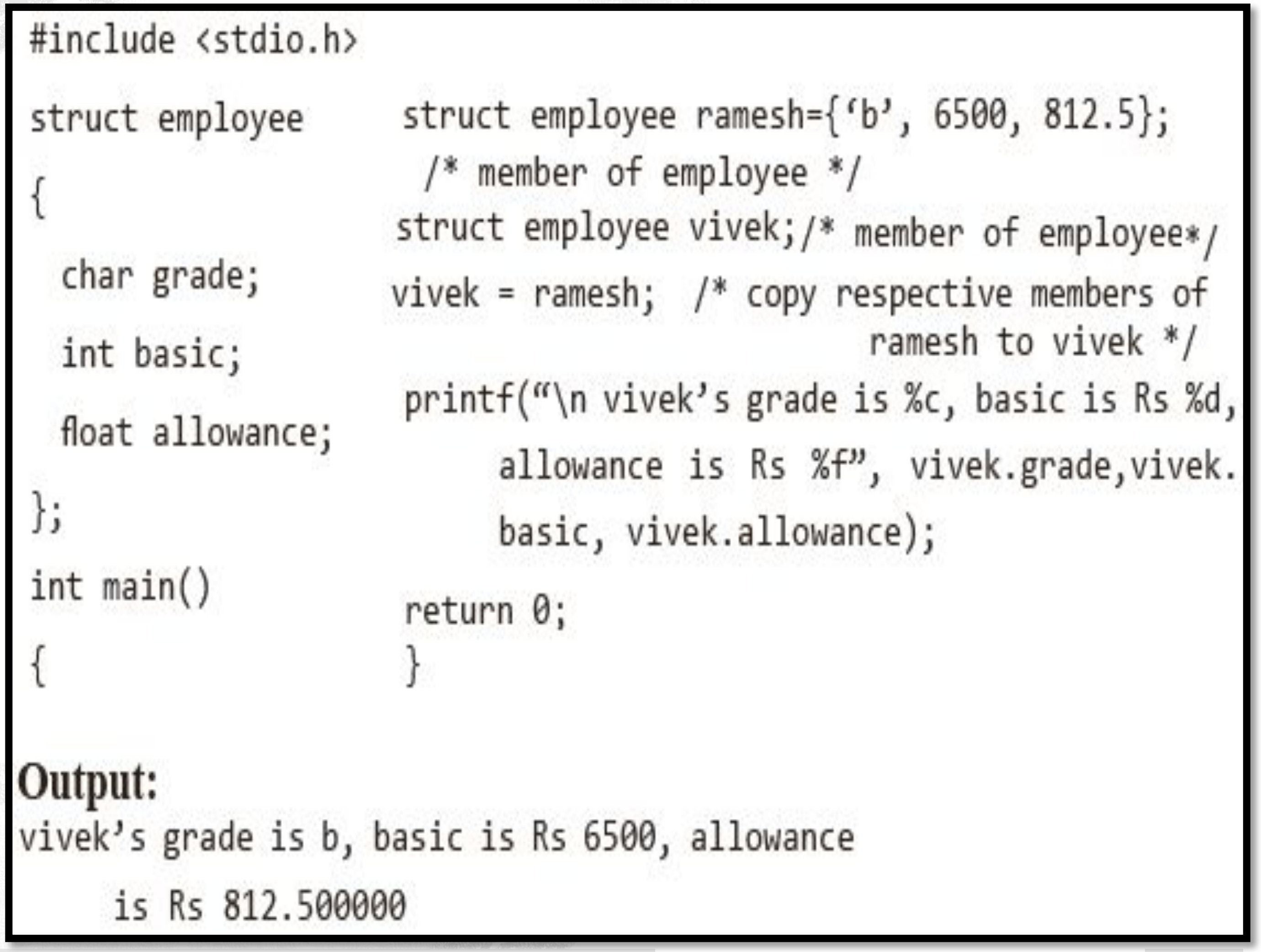
Initialization of Structures

- The general construct for initializing a structure can be any of the two forms given as follows.

```
struct <structure_tag_name>
{
      <data_type member_name1>;
      <data_type member_name2>;
}<structure_variable1> = {constant1,constant2, . .};
or
struct <structure_tag_name> <structure_variable> ={constant1,constant2,..};
```

Copying and Comparing Structures

- A structure can be assigned to another structure of the same type.
  - Here is an example of assigning one structure to another.
  - Comparing one structure variable with another is not allowed in C.
  - However, when comparing two structures, one should compare the individual fields in the structure.

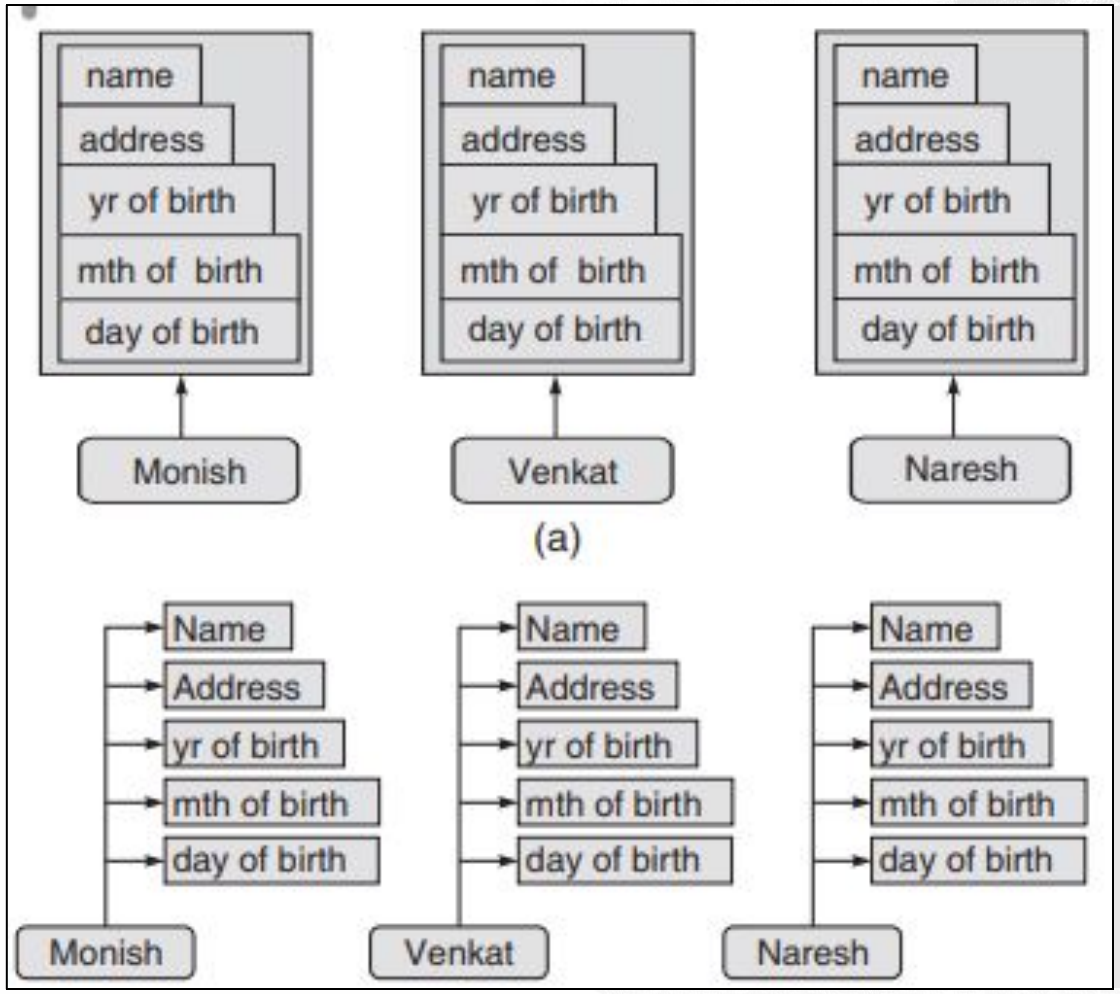


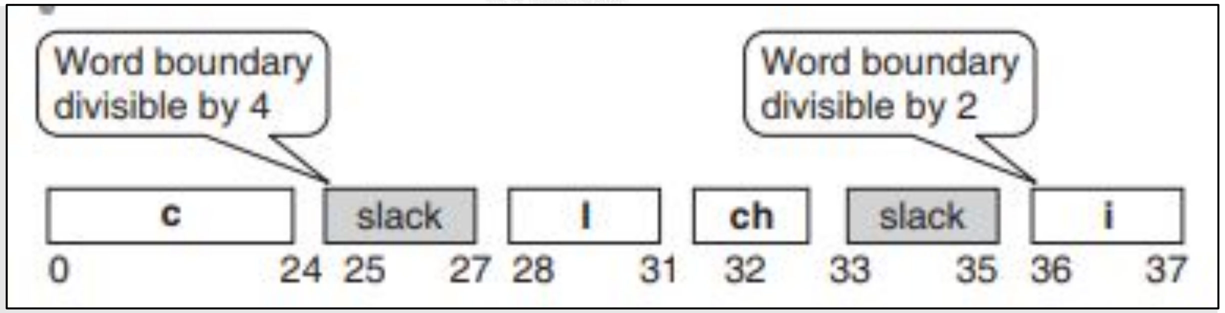
```
#include <stdio.h>

struct employee
{
  char grade;
  int basic;
  float allowance;
};
int main()
{
struct employee ramesh={'b', 6500, 812.5};
 /* member of employee */
struct employee vivek;/* member of employee*/
vivek = ramesh;  /* copy respective members of
                        ramesh to vivek */
printf("\n vivek's grade is %c, basic is Rs %d,
     allowance is Rs %f", vivek.grade,vivek.
     basic, vivek.allowance);
return 0;
}
```

Output:

```
vivek's grade is b, basic is Rs 6500, allowance
    is Rs 812.500000
```


name
address
yr of birth
mth of birth
day of birth
Monish
Venkat
Naresh
(a)
Name
Address
yr of birth
mth of birth
day of birth
Monish
Venkat
Naresh


Word boundary divisible by 4
Word boundary divisible by 2
c
slack
l
ch
slack
i
0
24
25
27
28
31
32
33
35
36
37

Typedef and Its Use in Structure Declarations

- The typedef keyword allows the programmer to create a new data type name for an existing data type.
  - The general form of the declaration statement using the typedef keyword is given as follows.

    typedef <existing data type> <new data type ,….>;
  - The typedef statement does not occupy storage; it simply defines a new type.
  - **typedef** statements can be placed anywhere in a C program as long as they come prior to their first use in the code.
- The following examples show the use of typedef.
  - typedef int id_number;
  - typedef float weight;
  - typedef char lower_case;

Nesting of Structures

- A structure can be placed within another structure.
  - In other words, structures can contain other structures as members.
  - A structure within a structure means nesting of structures.

- In such cases, the dot operator in conjunction with the structure variables are used to access the members of the innermost as well as the outermost structures.
  - It must be noted that an innermost member in a nested structure can be accessed by chaining all the concerned structure variables, from outermost to innermost, with the member using the dot operator.

Arrays of Structures

- The structure variable would be an array of objects, each of which contains the member elements declared within the structure construct.
- The general construct for declaration of an array structure is given as follows:

```
struct <structure_tag_name >
{
      <data_type member_name1>;
      <data_type member_name2>;
      .. .
}<structure_variable>[index];
Or
struct <structure_tag_name> <structure_variable>[index];
```

Example: Arrays of Structures

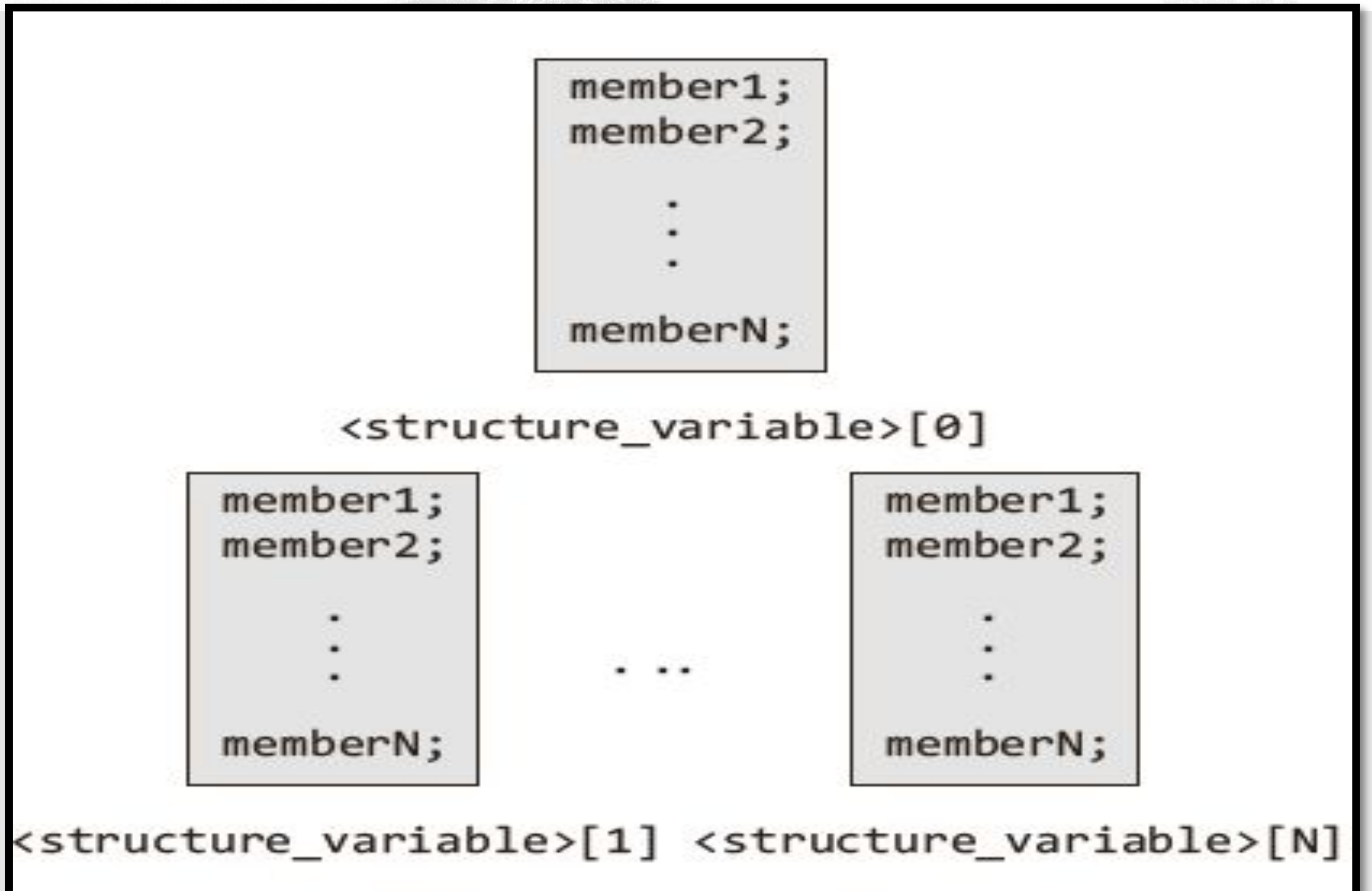
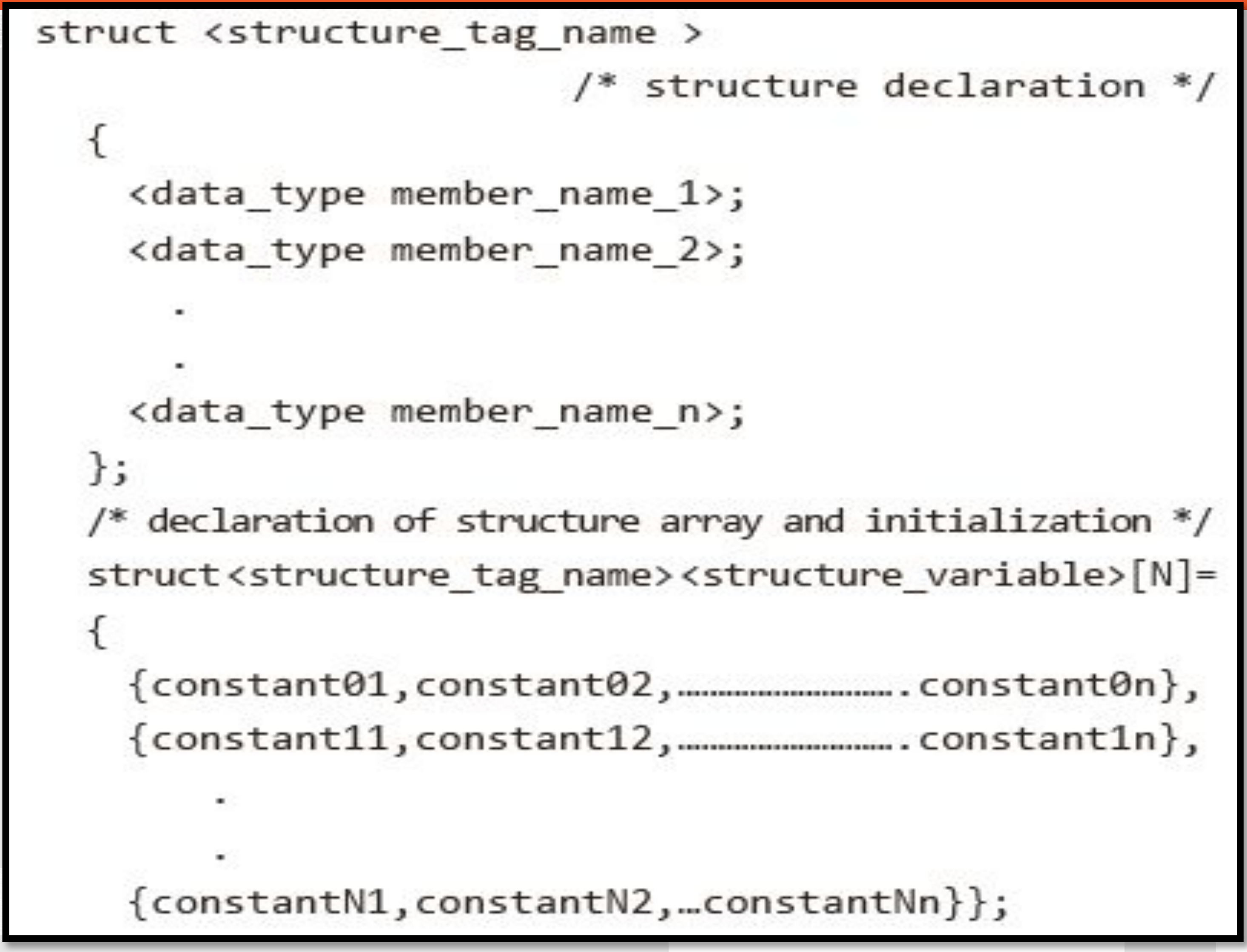


Figure Array of structures

Initializing Arrays of Structures

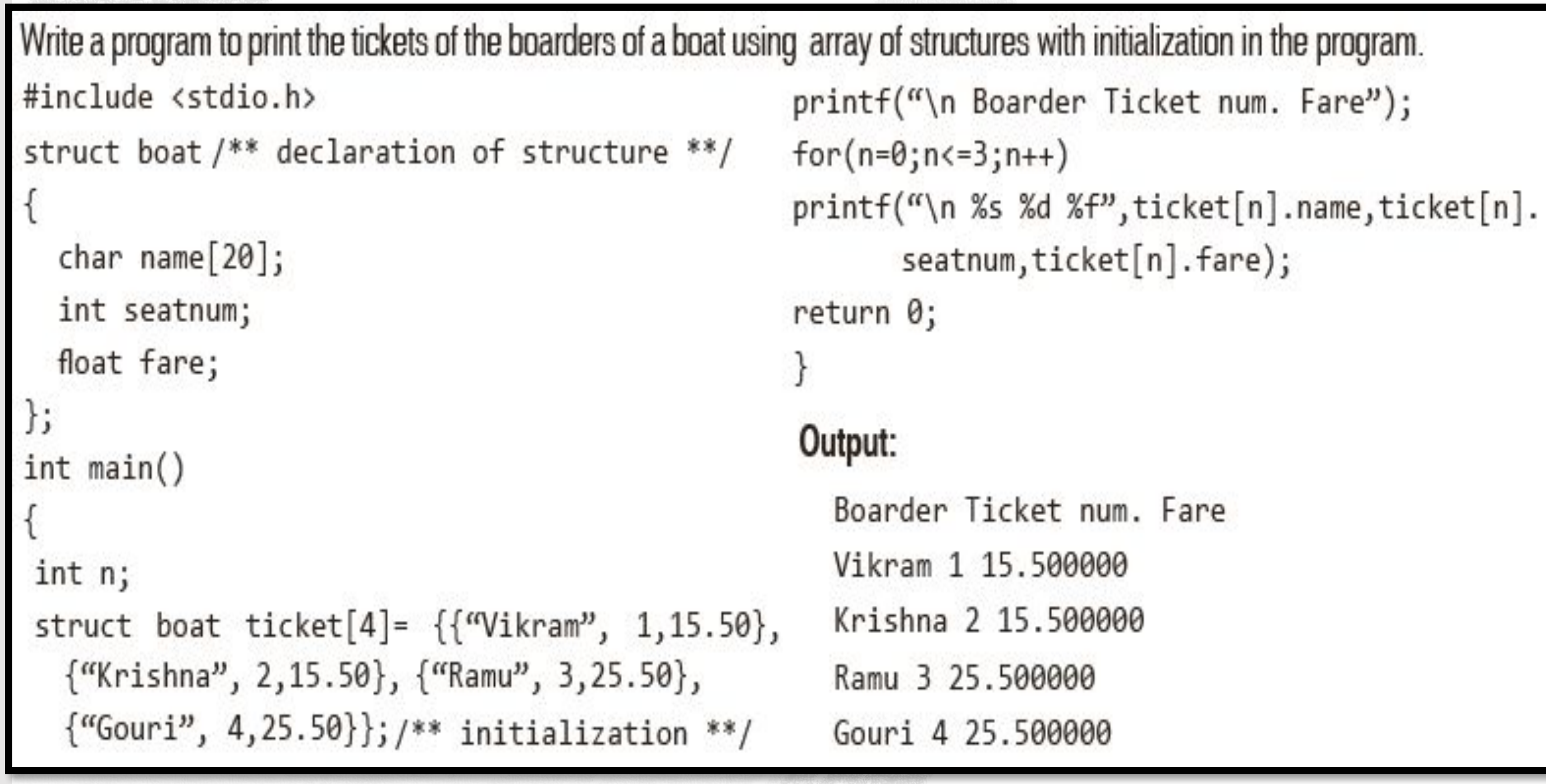
- Initializing arrays of structures is carried out in much the same way as arrays of standard data types.
    - A typical construct for initialization of an array of structures would appear as follows:



```
struct <structure_tag_name >
                        /* structure declaration */
  {
    <data_type member_name_1>;
    <data_type member_name_2>;
      .
      .
    <data_type member_name_n>;
  };
  /* declaration of structure array and initialization */
  struct<structure_tag_name><structure_variable>[N]=
  {
    {constant01,constant02,.........constant0n},
    {constant11,constant12,.........constant1n},
        .
        .
    {constantN1,constantN2,…constantNn}};
```

Arrays within the Structure

- An innermost member in a nested structure can be accessed by chaining all the concerned structure variables, from outermost to innermost, with the member using the dot operator.
   - Example:



Write a program to print the tickets of the boarders of a boat using array of structures with initialization in the program.

```
#include <stdio.h>
struct boat /** declaration of structure **/
{
  char name[20];
  int seatnum;
  float fare;
};
int main()
{
 int n;
 struct boat ticket[4]= {{"Vikram", 1,15.50},
   {"Krishna", 2,15.50}, {"Ramu", 3,25.50},
   {"Gouri", 4,25.50}};/** initialization **/
printf("\n Boarder Ticket num. Fare");
for(n=0;n<=3;n++)
printf("\n %s %d %f",ticket[n].name,ticket[n].
      seatnum,ticket[n].fare);
return 0;
}
```

Output:

```
Boarder Ticket num. Fare
Vikram 1 15.500000
Krishna 2 15.500000
Ramu 3 25.500000
Gouri 4 25.500000
```

Structures and Pointers

- At times it is useful to assign pointers to structures.
  - A pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure.
  - This pointer variable takes four bytes of memory just like any other pointer in a 32-bit machine.
  - Declaring pointers to structures is basically the same as declaring a normal pointer.
  - There are many reasons for using a pointer to a struct. One of them is to make a two-way communication possible within functions.
  - This aspect is explained with examples in the following section.
  - A pointer to a structure is not itself a structure, but merely a variable that holds the address of a structure.

Structures and Pointers

- A typical construct for declaring a pointer to a structure will appear as follows:

```
struct <structure_tag_name
   >
/* structure declaration */
{
<data_type
   member_name_1>;
<data_type
   member_name_2>;
. . .
<data_type
   member_name_n>;
}*ptr;
```

or

```
struct <structure_tag_name>
{
<data_type member_name_1>;
<data_type member_name_2>;
…
<data_type member_name_n>;
};
struct <structure_tag_name>
   *ptr;
```

Structures and Functions

- Passing and working with pointers to large structures may be more efficient while passing structures to a function and working within it.
   - When a structure is passed as an argument, each member of the structure is copied.
   - In fact, each member is passed by value. In case the member is an array, a copy of this array is also passed.
   - This can prove to be inefficient where structures are large or functions are called frequently.
   - The general construct for passing a structure to a function and returning a structure is

      struct structure_tag function_name (struct structure_tag structure_variable);

Union

- A union is a structure all of whose members share the same storage.
  - The amount of storage allocated to a union is sufficient to hold its largest member.
  - At any given time, only one member of the union may actually reside in that storage.
  - A union is identified in C through the use of the keyword union in place of the keyword struct.
  - Virtually all other methods for declaring and accessing unions are identical to those for structures.
- **Declaring a Union and its Members :** The general construct for declaring a union is given as follows:

```
union tag_name
{
    Member1;
    . . .
    memberN;
}variable1,variable2,variable3,…,variableX;
```

Union

- The general construct of declaring the individual union variables is
  - union tag_name variable1,variable2,…,variableX;
- As an example, consider the following declarations for a union that has a tag named mixed.

```
union mixed
{
        char letter;
        float radian;
        int number;
};
union mixed all;
```

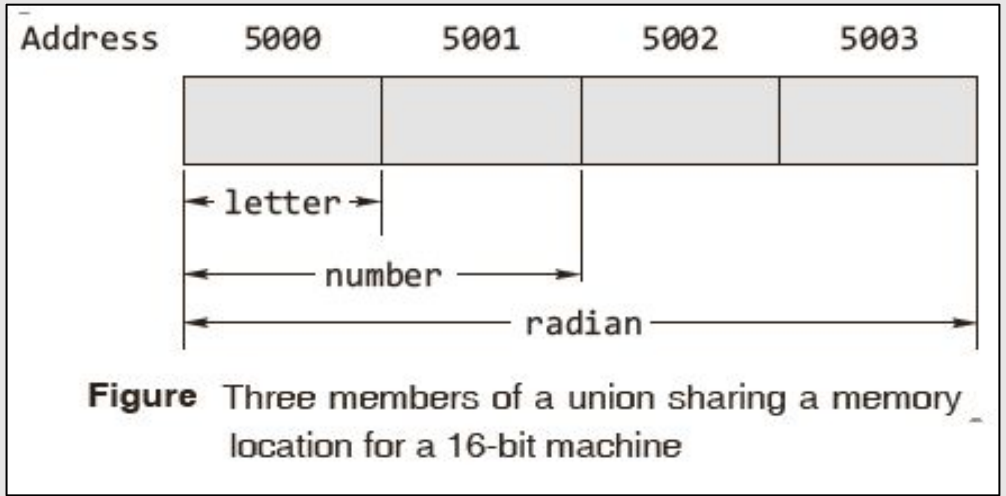


Figure Three members of a union sharing a memory location for a 16-bit machine

- The first declaration consists of a union of type *mixed,* which consists of a char, float, or int variable as a member.

Union

- The union data type was created to prevent the computer from breaking its memory up into several inefficiently sized pieces, which is called *memory fragmentation.*
- **Accessing and Initializing the Members of a Union:**
  - Consider, the general declaration construct of a union.

```
union tag_name
{
member1;
member2;
.. .
memberN;
}variable1,variable2,variable3,…,variableX;
```

Accessing and Initializing the Members of a Union

- For accessing members of, say, variable1 to N of the union tag_name, the following constructs are used.

```
variable1.member1
variable2.member2
.. .
 variableX.memberN
```

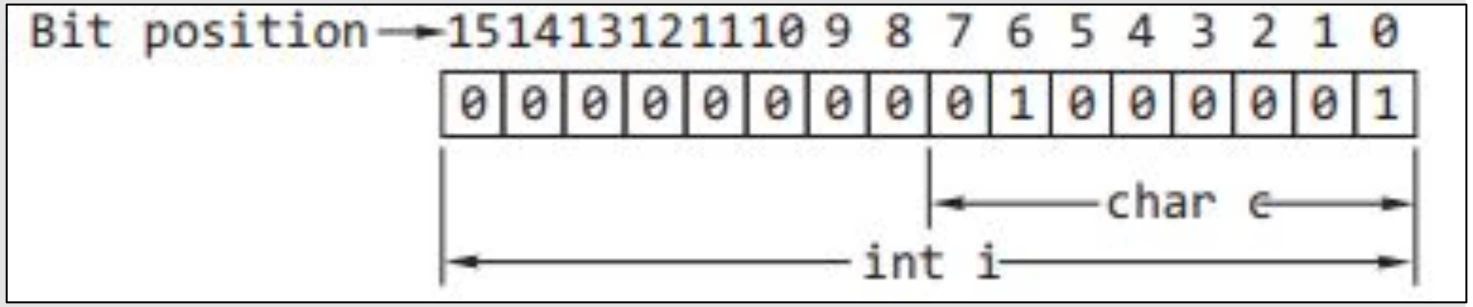
- Only a member that exists at the particular instance in storage should be accessed.
- The general construct for individual initialization of a union member is

```
variableX.memberN = constant;
```

where X is any value 1 to X and N is any value 1 to N.

```
struct conditions
  {
    float temp;
    union feels_like {
      float wind_chill;
      float heat_index;
      }
  } today;
```

```
union test                 /* declaration of union */
{
int i;                           /* integer member */
char c;                        /* character member */
}var;                                 /* variable */
```



Structure versus Union

- At any given time, only one member of the union may actually reside in the storage.
- In a union, the amount of memory required is same as that of the largest member.
- It is important to remember which union member is being used. If the user fills in a member of one type and then tries to use a different type, the results can be unpredictable.
- Performing arithmetical or logical operations on union variables is not allowed.

Structure versus Union

- The following operations on union variables are valid:
  - A union variable can be assigned to another union variable.
  - A union variable can be passed to a function as a parameter.
  - The address of a union variable can be extracted by using & operator.
  - A function can accept and return a union or pointer to a union.
- No attempt should be made to initialize more than one union member.

Do’s and don’ts for Unions

- It is important to remember which union member is being used.
  - If the user fills in a member of one type and then tries to use a different type, the results can be unpredictable.

- The following operations on union variables are valid.
  - A union variable can be assigned to another union variable.
  - A union variable can be passed to a function as a parameter.
  - The address of a union variable can be extracted by using & operator.
  - A function can accept and return a union or a pointer to a union.
  - Don’t try to initialize more than the first union member.
  - Don’t forget that the size of a union is equal to its largest member.
  - Don’t perform arithmetical or logical operations on union variables.

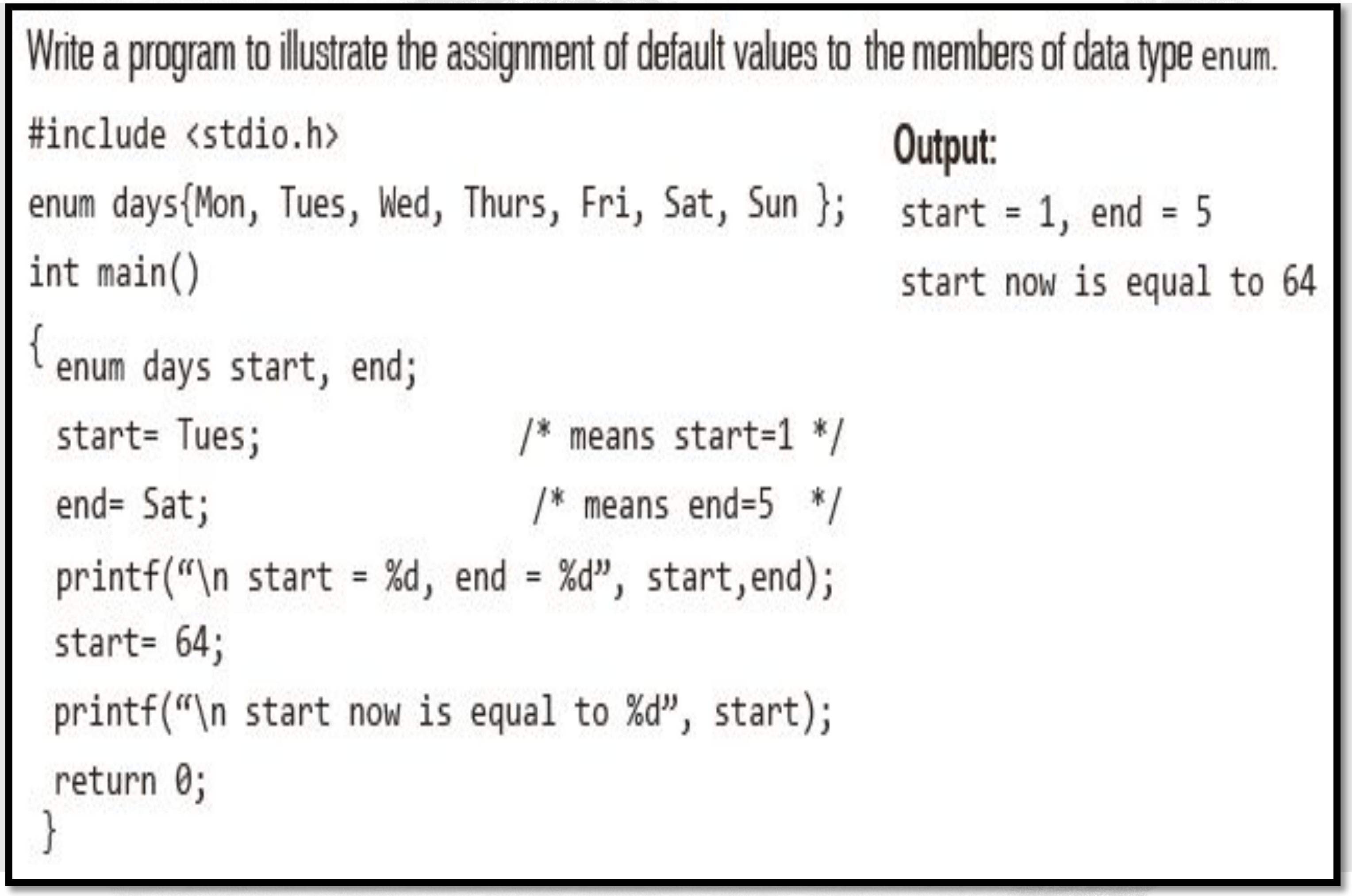
Enumeration Types

- Enumeration data types are data items whose values may be any member of a symbolically declared set of values.
   - The symbolically declared members are integer constants.
- The keyword enum is used to declare an enumeration type. The general construct used to declare an enumeration type is enum:
   - **tag_name{member1, member2,…, memberN}**
- variable1,...,variableX; In this declaration, either tag_name or variable may be omitted or both may be present.
- But at least one of them must exist in this declaration construct.

Enumeration Types

- The enum tag_name specifies the user-defined type.
- The members are integer constants. By default, the first member, that is, member1, is given the value 0.
- The second member, member2, is given the value 1.
- Members within the braces may be initialized, in which case, the next member is given a value one more than the preceding member. So, each member is given the value of the previous member plus 1.

Example



Write a program to illustrate the assignment of default values to the members of data type enum.

```
#include <stdio.h>
enum days{Mon, Tues, Wed, Thurs, Fri, Sat, Sun };
int main()
{ enum days start, end;
  start= Tues;                    /* means start=1 */
  end= Sat;                        /* means end=5  */
  printf("\n start = %d, end = %d", start,end);
  start= 64;
  printf("\n start now is equal to %d", start);
  return 0;
 }
```

Output:

```
start = 1, end = 5
start now is equal to 64
```

Bit Fields

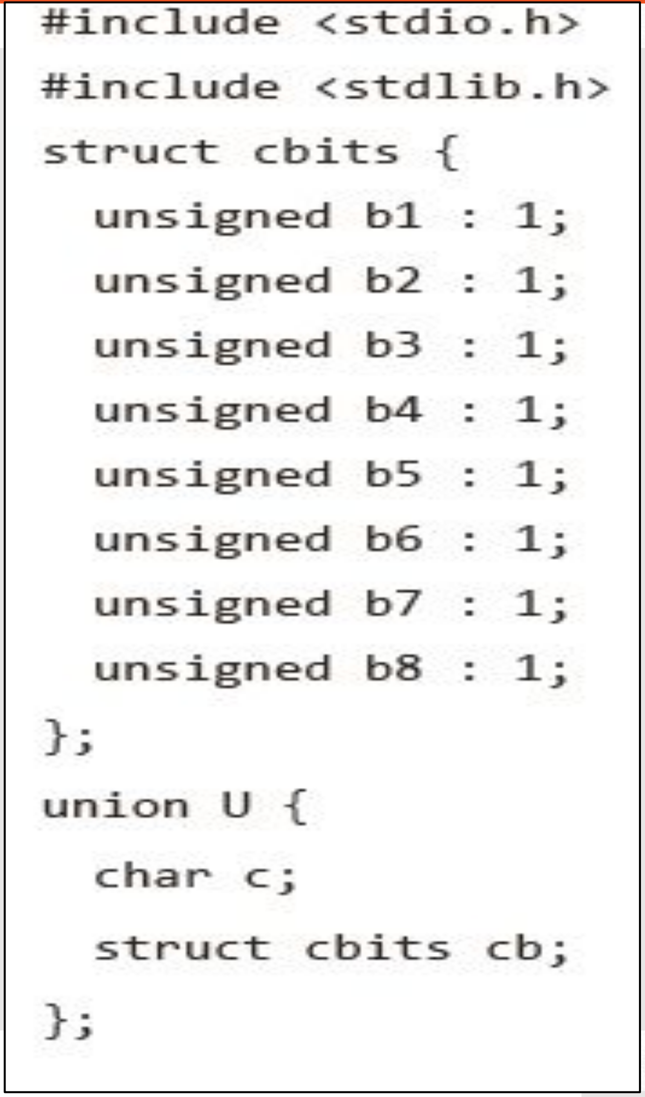
- There are two ways to manipulate bits in C.
  - One of the ways consists of using bitwise operators.
  - The other way consists of using bit fields in which the definition and the access method are based on structure.
- The general format for declaring a bit field using a structure is given as follows:

```
struct bitfield_tag
{
      unsigned int member1: bit_width1;
      unsigned int member2: bit_width2;
      .. .
      unsigned int memberN: bit_widthN;
};
```

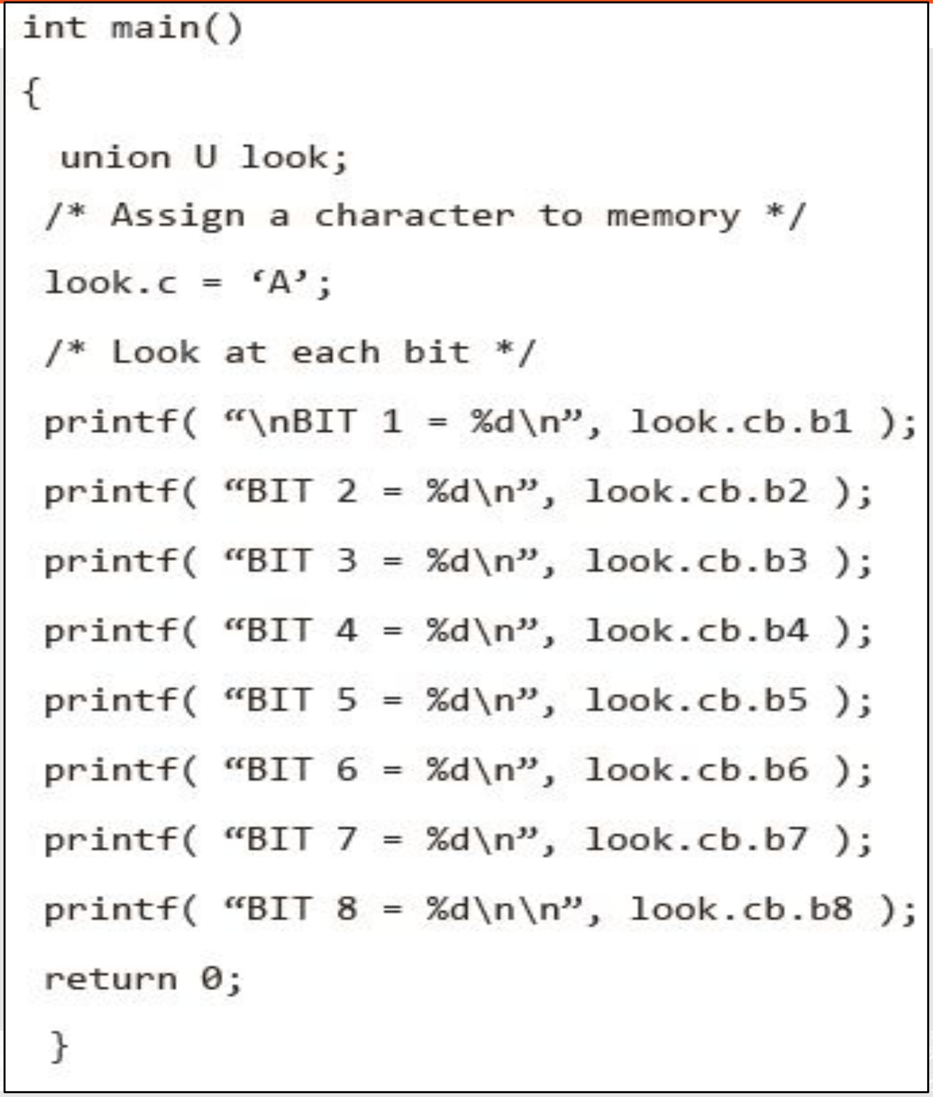
Bit Fields

- With reference to bitfields, it should be noted that a field in a word has no address.
- In this construct, the declaration of variable name is optional. The construct for individually declaring the variables to this structure is given by
    - □ struct bitfield_tag variable_name;
- Each bit field, for example, ‘unsigned int member1: bit_ width1’, is an integer that has a specified bit width.

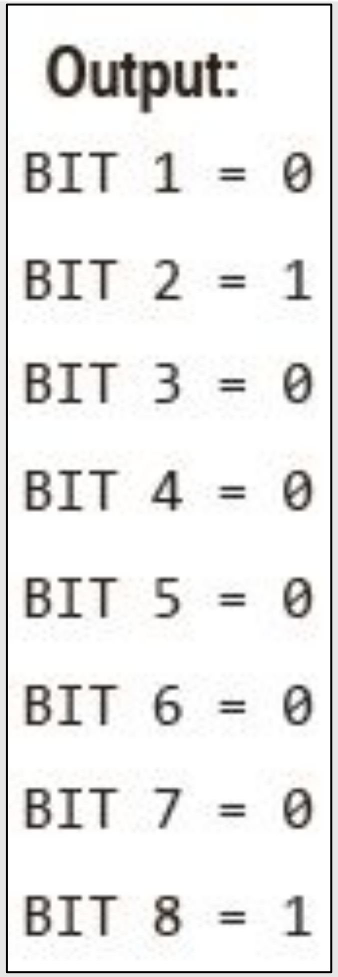
Example



```
#include <stdio.h>
#include <stdlib.h>
struct cbits {
  unsigned b1 : 1;
  unsigned b2 : 1;
  unsigned b3 : 1;
  unsigned b4 : 1;
  unsigned b5 : 1;
  unsigned b6 : 1;
  unsigned b7 : 1;
  unsigned b8 : 1;
};
union U {
  char c;
  struct cbits cb;
};
```



```
int main()
{
 union U look;
 /* Assign a character to memory */
 look.c = ‘A’;
 /* Look at each bit */
 printf( “\nBIT 1 = %d\n”, look.cb.b1 );
 printf( “BIT 2 = %d\n”, look.cb.b2 );
 printf( “BIT 3 = %d\n”, look.cb.b3 );
 printf( “BIT 4 = %d\n”, look.cb.b4 );
 printf( “BIT 5 = %d\n”, look.cb.b5 );
 printf( “BIT 6 = %d\n”, look.cb.b6 );
 printf( “BIT 7 = %d\n”, look.cb.b7 );
 printf( “BIT 8 = %d\n\n”, look.cb.b8 );
 return 0;
 }
```



Output:

```
BIT 1 = 0
BIT 2 = 1
BIT 3 = 0
BIT 4 = 0
BIT 5 = 0
BIT 6 = 0
BIT 7 = 0
BIT 8 = 1
```

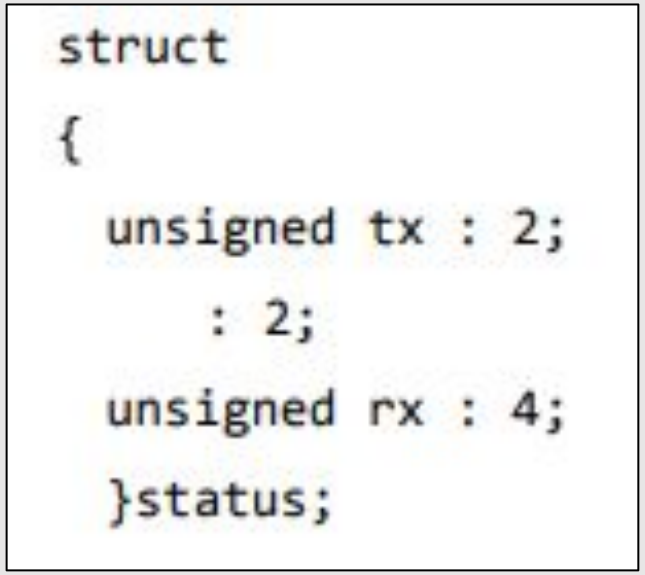
Bit Fields

- The previous output makes sense because

```
01000001(binary)=65(decimal)=101(octal)=41(hexadecimal)=A(ASCII)
```

- If one wants to do this with an integer, the size using the function
- sizeof(int) has to be first determined, then a structure is created with eight bit-fields for each byte counted by sizeof(int).

- Bitfields are extremely implementation dependent.
- For example, C does not specify whether fields must be stored left to right within a word, or vice-versa.
- Some compilers may not allow fields to cross a
- word boundary.
- Unnamed fields may be used as fillers.



```
struct
{
  unsigned tx : 2;
           : 2;
  unsigned rx : 4;
 }status;
```

Command line Arguments - int main(int argc, char *argv[]) { /* … */ }

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
   printf("Program name %s\n", argv[0]);
    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");
   }
}
```

COMPLEX NUMBERS

- A complex number is a number with a real part and an imaginary part.
- It is of the form a + bi where i is the square root of minus one, and a and b are real numbers.
- a is the real part, and bi is the imaginary part of the complex number.
- A complex number can also be regarded as an ordered pair of real numbers (a, b).

According to C99, three complex types are supported:

```
float complex
double complex
long double complex
```

C99 implementations support three imaginary types also:

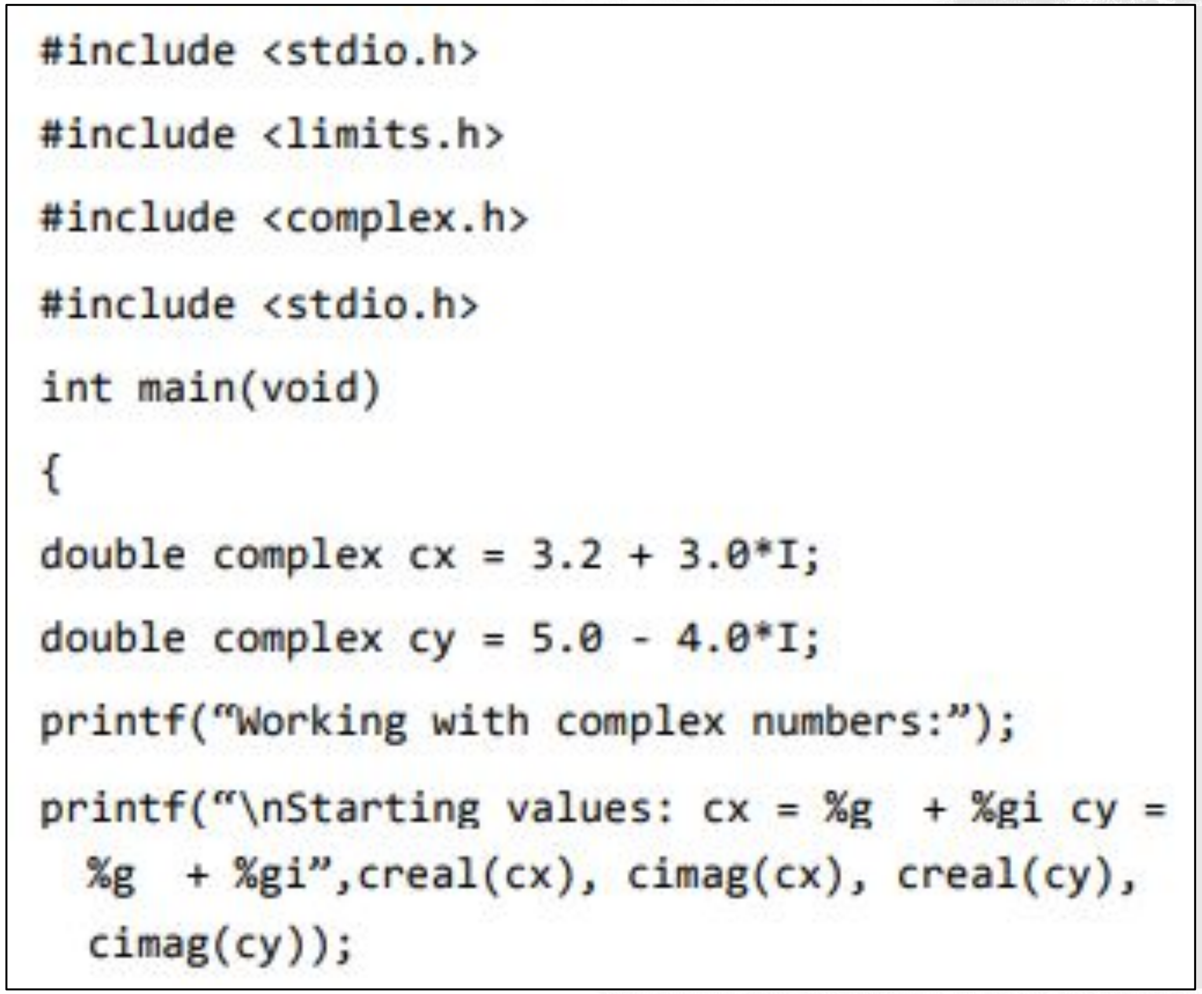
```
float imaginary
double imaginary
long double imaginary
```

COMPLEX NUMBERS

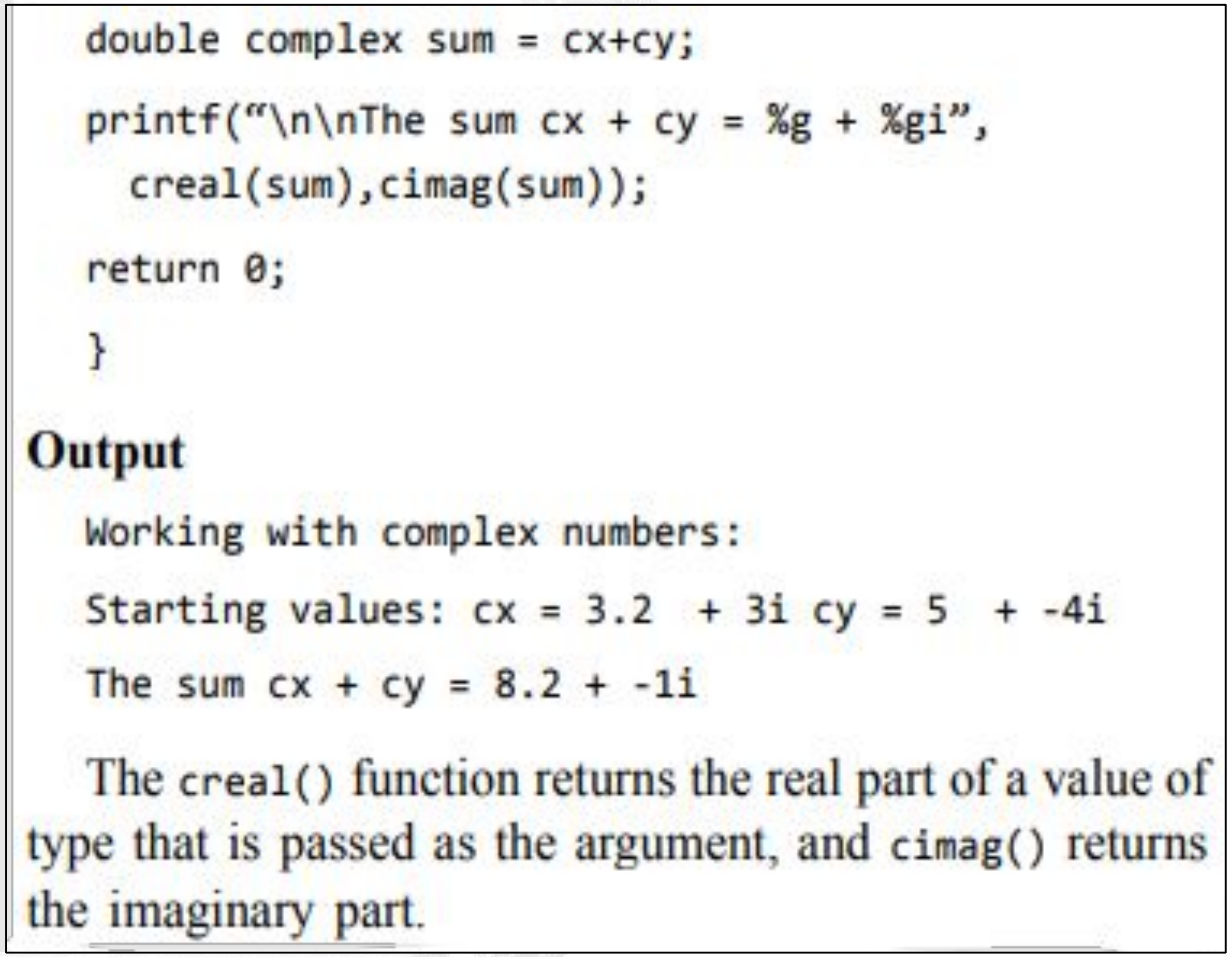
To use the complex types, the complex.h header file must be included.

double complex c1 = 3.2 + 2.0 * I;

float imaginary c2= -5.0 * I;



```c
#include <stdio.h>
#include <limits.h>
#include <complex.h>
#include <stdio.h>
int main(void)
{
double complex cx = 3.2 + 3.0*I;
double complex cy = 5.0 - 4.0*I;
printf(“Working with complex numbers:”);
printf(“\nStarting values: cx = %g  + %gi cy =
  %g  + %gi”,creal(cx), cimag(cx), creal(cy),
  cimag(cy));
```



```c
double complex sum = cx+cy;
printf(“\n\nThe sum cx + cy = %g + %gi”,
  creal(sum),cimag(sum));
return 0;
}
```

Output

```
Working with complex numbers:
Starting values: cx = 3.2  + 3i cy = 5  + -4i
The sum cx + cy = 8.2 + -1i
```

The `creal()` function returns the real part of a value of type that is passed as the argument, and `cimag()` returns the imaginary part.

Constant Parameter in Function

Macro Functions

- Macros are generally used to define constant values that are being used repeatedly in program.
- Macros can even accept arguments and such macros are known as function-like macros.
- It can be useful if tokens are concatenated into code to simplify some complex declarations.
- Macros provide text replacement functionality at pre-processing time.

```
#define MAX_SIZE 10
#define SWAP(a,b)({a ^= b; b ^= a; a ^= b;})
#define SQUARE(x) (x*x)
```

- Macros are handled by the pre-compiler, and are thus guaranteed to be inlined.
- Macros are used for short operations and it avoids function call overhead.
- It can be used if any short operation is being done in program repeatedly.
- Function-like macros are very beneficial when the same block of code needs to be executed multiple times.

```
#define AREA(l, b) (l * b)
#define min(a, b) (((a) < (b)) ? (a) : (b))
#define  AREA(r)  (PI*(r)*(r))
#define CUBE(b) b*b*b
```

There are 2 types of macros present in C such as:-

- Object-like Macros.
- Function-like Macros.

macro to define logging function

It allows variable arguments list and displays arguments on standard output as per format specified.

```
#include <stdio.h>
#define TRACE_LOG(fmt, args...) fprintf(stdout, fmt, ##args);
int main() {
int i=1;
TRACE_LOG("%s", "Sample macro\n");
TRACE_LOG("%d %s", i, "Sample macro\n");
return 0;
}
```

```
// Multi-line Macro definition
#define ELE 1, \
            2, \
            3

int main()
{
     // Array arr[] with elements
    // defined in macros
    int arr[] = { ELE };
```

Conditional Macros

```
#include <stdio.h>
int main() {

#if 0
printf("commented code 1");
printf("commented code 2");
#endif

#define TEST1 1
#ifdef TEST1
printf("MACRO TEST1 is defined\n");
#endif
```

```
#ifdef TEST3
printf("MACRO TEST3 is defined\n");
#else
printf("MACRO TEST3 is NOT
defined\n");
#endif
return 0;
}
```

Predefined Macros in C

__DATE__	Current date as MMM DD YYYY format.
__TIME__	Current time as HH:MM:SS format.
__FILE__	Contains current filename.
__LINE__	Contains current line number.
__STDC__	Defined as 1 when the compiler compiles.

Predefined Macros in C

```
#include <stdio.h>
int main() {
	char filename[] = __FILE__;
	char date[] = __DATE__;
	char time[] = __TIME__;
	int line = __LINE__;
	int ansi = __STDC__;
	printf("File name is: %s\n", filename);
	printf("Date is: %s\n", date);
	printf("Now time is: %s\n", time);
	printf("Current line number: %d\n", line);
	printf("Compilation Success: %d\n", ansi);
}
```

```
File name is: main.c
Date is: Dec 23 2022
Now time is: 09:47:33
Current line number: 15
Compilation Success: 1
```

Why to use Macros in C?

- It becomes handy when you use it for anything magic number or string related.
- You can use macros to create automatic loop unrolling.
- With macros, you can do some things which you cannot do with functions like Token Pasting.
- You can use a macro for writing debugging messages also.

Macro vs Functions

Macro	Function
Macros are Preprocessed	Functions are Compiled
No Type Checking is done in Macro	Type Checking is Done in Function
Using Macro increases the code length	Using Function keeps the code length unaffected
Use of macro can lead to side effect at later stages	Functions do not lead to any side effect in any case
Speed of Execution using Macro is Faster	Speed of Execution using Function is Slower
Before Compilation, macro name is replaced by macro value	During function call, transfer of control takes place
Macros are useful when small code is repeated many times	Functions are useful when large code is to be written
Macro does not check any Compile-Time Errors	Function checks Compile-Time Errors

Tower of Hanoi

```
#include <stdio.h>

void towerOfHanoi(int n, char from_tower, char to_tower, char aux_tower)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from tower%c to tower%c", from_tower, to_tower);
        return;
    }
    towerOfHanoi(n-1, from_tower, aux_tower, to_tower);
    printf("\n Move disk %d from tower%c to tower%c", n, from_tower, to_tower);
    towerOfHanoi(n-1, aux_tower, to_tower, from_tower);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of tower
    return 0;
}
```

Thank You!