BLG 102E Introduction to Scientific Computing and Engineering

SPRING 2025

WEEK 11

ISTANBUL TECHNICAL UNIVERSITY

Structs

Introduction (1 of 2)

- **Structures**—sometimes referred to as **aggregates**—are collections of related variables under one name.
- Structures may contain variables of many different data types—in contrast to arrays, which contain **only** elements of the same data type.
- Structures are commonly used to define **records** to be stored in files (see Chapter 11, C File Processing).
- Pointers and structures facilitate the formation of more complex data structures such as linked lists, queues, stacks and trees (see Chapter 12, C Data Structures).

Structure Definitions (1 of 8)

- Structures are **derived data types**—they’re constructed using objects of other types.
- Consider the following structure definition:

```
struct card {
   char *face;
   char *suit;
};
```

- Keyword **`struct`** introduces the structure definition.
- The identifier `card` is the **structure tag**, which names the structure definition and is used with `struct` to declare variables of the **structure type**—e.g., `struct card`.

Structure Definitions (2 of 8)

- Variables declared within the braces of the structure definition are the structure's **members**.
- Members of the same structure type must have unique names, but two different structure types may contain members of the same name without conflict (we'll soon see why).
- Each structure definition **must** end with a semicolon.

Common Programming Error

Forgetting the semicolon that terminates a structure definition is a syntax error.

Structure Definitions (3 of 8)

- The definition of `struct` card contains members `face` and `suit`, each of type `char *`.
- Structure members can be variables of the primitive data types (e.g., `int, float`, etc.), or aggregates, such as arrays and other structures.
- Structure members can be of many types.

Structure Definitions (4 of 8)

- For example, the following `struct` contains character array members for an employee's first and last names, an `unsigned int` member for the employee's age, a `char` member that would contain '`M`' or '`F`' for the employee's gender and a `double` member for the employee's hourly salary:,

```
struct employee {
   char firstName[20];
   char lastName[20];
   unsigned int age;
   char gender;
   double hourlySalary;
};
```

Self-Referential Structures (1 of 2)

- **A structure cannot contain an instance of itself.**
- For example, a variable of type `struct employee` cannot be declared in the definition for `struct employee`.
- A pointer to `struct employee`, however, may be included.

Self-Referential Structures (2 of 2)

- For example,

```
struct employee2 {
   char firstName[20];
   char lastName[20];
   unsigned int age;
   char gender;
   double hourlySalary;
   struct employee2 person; // ERROR
   struct employee2 *ePtr; // pointer
};
```

- `struct employee2` contains an instance of itself (`person`), which is an error.

Structure Definitions (5 of 8)

- Because `ePtr` is a pointer (to type `struct employee2`), it's permitted in the definition.
- A structure containing a member that's a pointer to the **same** structure type is referred to as a **self-referential structure**.
- Self-referential structures are used in Chapter 12 to build linked data structures.

Defining Variables of Structure Types

- Structure definitions do **not** reserve any space in memory; rather, each definition creates a new data type that's used to define variables.
- Structure variables are defined like variables of other types.
- The definition

```
struct card aCard, deck[52], *cardPtr;
```

declares `aCard` to be a variable of type `struct card`, declares `deck` to be an array with 52 elements of type `struct card` and declares `cardPtr` to be a pointer to `struct` card.

Structure Definitions (6 of 8)

- Variables of a given structure type may also be declared by placing a comma-separated list of the variable names between the closing brace of the structure definition and the semicolon that ends the structure definition.
- For example, the preceding definition could have been incorporated into the `struct card` definition as follows:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[52], *cardPtr;
```

Structure Tag Names

- The structure tag name is optional.
- If a structure definition does not contain a structure tag name, variables of the structure type may be declared **only** in the structure definition—**not** in a separate declaration.

Common Programming Error

A structure cannot contain an instance of itself.

Good Programming Practice

Always provide a structure tag name when creating a structure type. The structure tag name is required for declaring new variables of the structure type later in the program.

Operations That Can be Performed on Structures (1 of 2)

The only valid operations that may be performed on structures are:

1. assigning structure variables to structure variables of the same type,
2. taking the address (&) of a structure variable,
3. accessing the members of a structure variable (see Section) and
4. using the size of operator to determine the size of a structure variable.

Common Programming Error

Assigning a structure of one type to a structure of a different type is a compilation error.

Operations That Can be Performed on Structures (2 of 2)

- Structures may **not** be compared using operators **==** and **!=** because structure members are not necessarily stored in consecutive bytes of memory.
- Sometimes there are “holes” in a structure, because computers may store specific data types only on certain memory boundaries such as half-word, word or double-word boundaries.
- A word is a standard memory unit used to store data in a computer—usually 2 bytes or 4 bytes.

Structure Definitions (7 of 8)

- Consider the following structure definition, in which `sample1` and `sample2` of type `struct example` are declared:

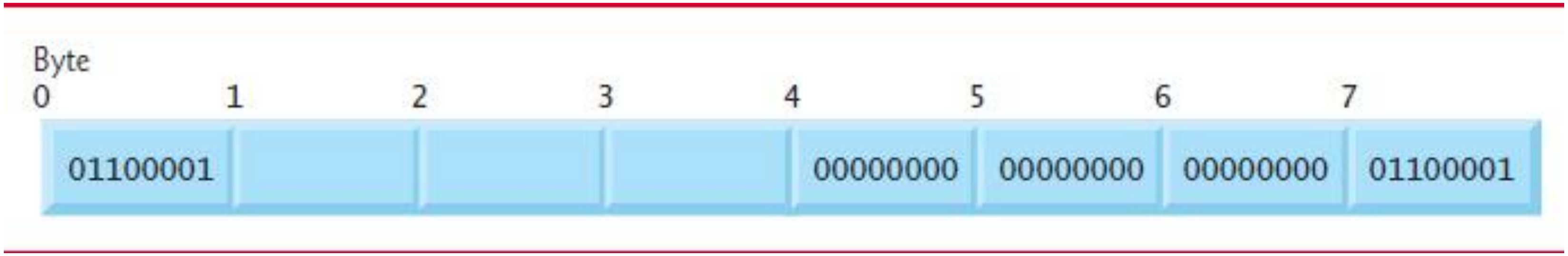
```
struct example {
   char c;
   int i;
} sample1, sample2;
```

- A computer with 4-byte words may require that each member of `struct example` be aligned on a word boundary, i.e., at the beginning of a word (this is machine dependent).

Structure Definitions (8 of 8)

- Figure shows a sample storage alignment for a variable of type struct example that has been assigned the character 'a' and the integer 97 (the bit representations of the values are shown).
- If the members are stored beginning at word boundaries, there's a 3-bytes hole (byte 1 in the figure) in the storage for variables of type struct example.
- The value in the 3-bytes hole is undefined.
- Even if the member values of sample1 and sample2 are in fact equal, the structures are not necessarily equal, because the undefined 3-bytes holes are not likely to contain identical values.

Possible Storage Alignment for a Variable of Type `struct` Example Showing an Undefined Area in Memory



Portability Tip

Because the size of data items of a particular type is machine dependent and because storage alignment considerations are machine dependent, so too is the representation of a structure.

Initializing Structures (1 of 2)

- Structures can be initialized using initializer lists as with arrays.
- To initialize a structure, follow the variable name in the definition with an equals sign and a brace-enclosed, comma-separated list of initializers.
- For example, the declaration

```
struct card aCard = {"Three", "Hearts"};
```

- creates variable aCard to be of type struct card (as defined in Section) and initializes member face to "Three" and member suit to "Hearts".

Initializing Structures (2 of 2)

- If there are fewer initializers in the list than members in the structure, the remaining members are automatically initialized to `0` (or `NULL` if the member is a pointer).
- Structure variables defined outside a function definition (i.e., externally) are initialized to `0` or `NULL` if they're not explicitly initialized in the external definition.
- Structure variables may also be initialized in assignment statements by assigning a structure variable of the **same** type, or by assigning values to the **individual** members of the structure.

Accessing Structure Members (1 of 4)

- Two operators are used to access members of structures: the structure member operator (.)—also called the dot operator—and the structure pointer operator (->) - also called the arrow operator.
- The structure member operator accesses a structure member via the structure variable name.
- For example, to print member suit of structure variable a Card defined in Section , use the statement

```
printf("%s", aCard.suit); // displays Hearts
```

Accessing Structure Members (2 of 4)

- The structure pointer operator—consisting of a minus (−) sign and a greater than (>) sign with no intervening spaces—accesses a structure member via a **pointer to the structure**.
- Assume that the pointer `cardPtr` has been declared to point to `struct` card and that the address of structure a `Card` has been assigned to `cardPtr`.
- To print member `suit` of structure `aCard` with pointer `cardPtr`, use the statement
  - `printf("%s", cardPtr->suit); // displays Hearts`

Accessing Structure Members (3 of 4)

- The expression `cardPtr->suit` is equivalent to `(*cardPtr).suit`, which dereferences the pointer and accesses the member `suit` using the structure member operator.
- The parentheses are needed here because the structure member operator (`.`) has a higher precedence than the pointer dereferencing operator (`*`).
- The structure pointer operator and structure member operator, along with parentheses (for calling functions) and brackets (`[]`) used for array subscripting, have the highest operator precedence and associate from left to right.

Good Programming Practice

Do not put spaces around the `->` and `.` operators. Omitting spaces helps emphasize that the expressions the operators are contained in are essentially single variable names.

Common Programming Error

Inserting space between the - and > components of the structure pointer operator is a syntax error.

Common Programming Error

Attempting to refer to a structure member by using only the member’s name is a syntax error.

Common Programming Error

Not using parentheses when referring to a structure member that uses a pointer and the structure member operator (e.g., `*cardPtr.suit`) is a syntax error. To prevent this problem use the arrow (->) operator instead.

Accessing Structure Members (4 of 4)

- The program of Figure. demonstrates the use of the structure member and structure pointer operators.
- Using the structure member operator, the members of structure a Card are assigned the values “Ace” and “Spades”, respectively
- Pointer card P t r is assigned the address of structure a Card
- Function print f prints the members of structure variable a Card using the structure member operator with variable name a Card, the structure pointer operator with pointer card P t r and the structure member operator with dereferenced pointer card P t r

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.

Example : Pointer to struct

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

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

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

    // assign data into student structure variable
    a.num = 111;
    strcpy(a.name , "ABCD");

    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

```
111 – ABCD
111 – ABCD
111 – ABCD
```

Example: Pointer to dynamically allocated Struct

- Program defines a struct pointer and allocates memory for the struct.
- The malloc (memory allocate) built-in C function is called to dynamically allocate memory. (Defined in <stdlib.h> header file.)

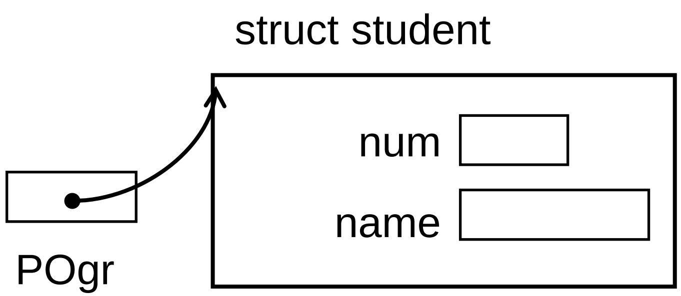
```c
struct student {
  int   num;
  char  name[20];
};

struct student  *POgr; // Pointer

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

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

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



Example: Copying a Struct variable

- Program copies a struct variable into another.

```
struct student  Ogr1 = {111, "ABCD"};
struct student  Ogr2, Ogr3;


// Method 1: Copy member fields of Ogr1 to Ogr2 one by one:
Ogr2.num = Ogr1.num;
strcpy(Ogr2.name , Ogr1.name);


// Method 2: Copy entire Ogr1 to Ogr3
Ogr3 = Ogr1; // Easy method for structure copying


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

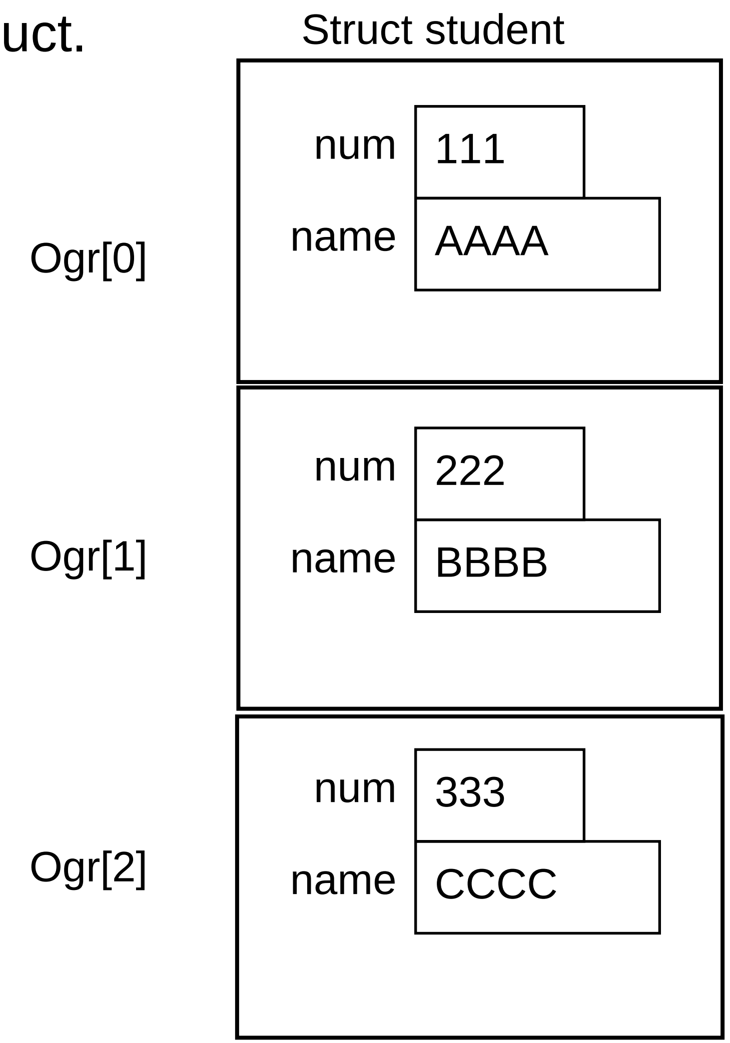
Example: Initializing an Array of Struct

- Program defines an array variable of struct.

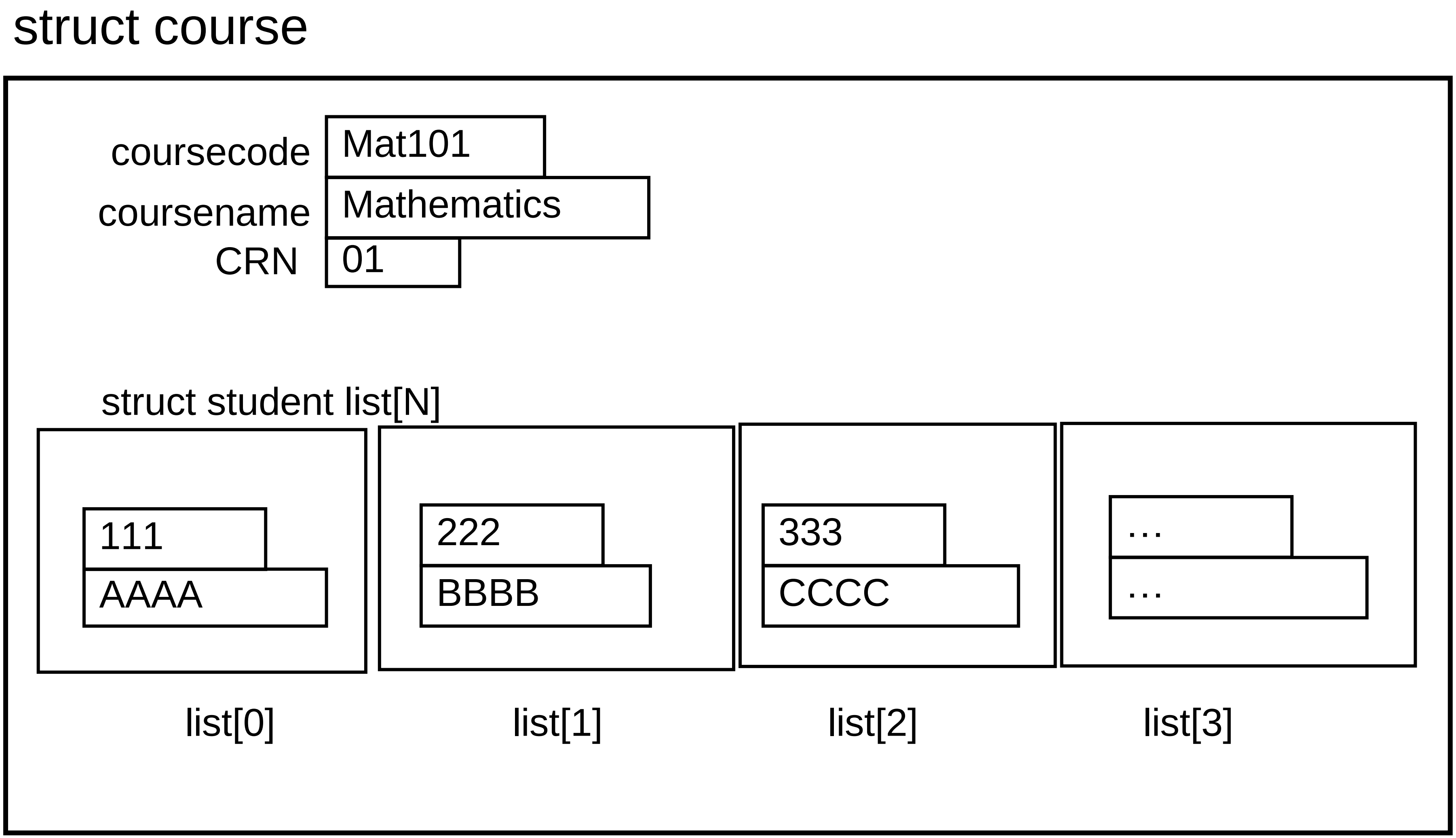
```c
#define N 3 // Number of persons

struct  student  Ogr[N] = {
   {111, "AAAA"},
   {222, "BBBB},
   {333, "CCCC"}
};


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



Example: Nested Structs



Example: Nested Structs

- Program defines nested strucs (student struct is in course struct).

```c
#define N 50
// Maximum number of students in course

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


struct course {
  char  coursecode[10];
  char  coursename[30];
  struct student list[N]; // Array of registered students
};
```

Example: Nested Structs (continued)

```
int main()
{
  struct course Course;
  int i;

  // Initialization of struct variables with data is done.
  // ...

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

  for (i = 0; i < N; i++) // loop for students
       printf("%d %s \n", Course.list[i].num, Course.list[i].name);

} // end of main
```

Figure Structure Member Operator and Structure Pointer Operator (1 of 2)

```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
};

int main(void)
{
   struct card aCard; // define one struct card variable

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

```

Figure Structure Member Operator and Structure Pointer Operator (2 of 2)

```
    printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
       cardPtr->face, " of ", cardPtr->suit,
       (*cardPtr).face, " of ", (*cardPtr).suit);
}
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

Using Structures with Functions (1 of 2)

- Structures may be passed to functions by passing individual structure members, by passing an entire structure or by passing a pointer to a structure.
- When structures or individual structure members are passed to a function, they’re passed by value.
- Therefore, the members of a caller’s structure cannot be modified by the called function.
- To pass a structure by reference, pass the address of the structure variable.

Using Structures with Functions (2 of 2)

- Arrays of structures—like all other arrays—are automatically passed by reference.
- To pass an array by value, create a structure with the array as a member.
- Structures are passed by value, so the array is passed by value.

Common Programming Error

Assuming that structures, like arrays, are automatically passed by reference and trying to modify the caller's structure values in the called function is a logic error.

Performance Tip

Passing structures by reference is more efficient than passing structures by value (which requires the entire structure to be copied).

typedef (1 of 4)

- The keyword **`typedef`** provides a mechanism for creating synonyms (or aliases) for previously defined data types.
- Names for structure types are often defined with `typedef` to create shorter type names.
- For example, the statement

```
typedef struct card Card;
```

  defines the new type name `Card` as a synonym for type `struct card`.
- C programmers often use `typedef` to define a structure type, so a structure tag is not required.

typedef (2 of 4)

- For example, the following definition

```
typedef struct {
    char *face;
    char *suit;
} Card;
```

creates the structure type `Card` without the need for a separate `typedef` statement.

Good Programming Practice

Capitalize the first letter of `typedef` names to emphasize that they’re synonyms for other type names.

typedef (3 of 4)

- `Card` can now be used to declare variables of type `struct card`.
- The declaration

  Card deck[**52**];

  declares an array of 52 `Card` structures (i.e., variables of `type struct` card).
- Creating a new name with `typedef` does **not** create a new type; `typedef` simply creates a new type name, which may be used as an alias for an existing type name.

typedef (4 of 4)

- A meaningful name helps make the program self-documenting.
- For example, when we read the previous declaration, we know "`deck` is an array of 52 `Cards`."
- Often, `typedef` is used to create synonyms for the basic data types.
- For example, a program requiring four-byte integers may use type `int` on one system and type `long` on another.
- Programs designed for portability often use `typedef` to create an alias for four-byte integers, such as `Integer`.
- The alias `Integer` can be changed once in the program to make the program work on both systems.

Portability Tip

Use `typedef` to help make a program more portable.

Good Programming Practice

Using `typedef`s can help make a program more readable and maintainable.

Example: Card Shuffling and Dealing Simulation (1 of 3)

- The program in Figure is based on the card shuffling and dealing simulation discussed in Chapter 7.
- The program represents the deck of cards as an array of structures and uses high-performance shuffling and dealing algorithms.
- The program output is shown in Figure see slide 58.

Figure Card Shuffling and Dealing Program Using Structures (1 of 4)

```c
// Fig. 10.3: fig10_03.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

// prototypes
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[]);
void shuffle(Card * const wDeck);
void deal(const Card * const wDeck);

```

Figure Card Shuffling and Dealing Program Using Structures (2 of 4)

```
int main(void)
{
    Card deck[CARDS]; // define array of Cards

    // initialize array of pointers
    const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
        "Six", "Seven", "Eight", "Nine", "Ten",
        "Jack", "Queen", "King"};

    // initialize array of pointers
    const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};

    srand(time(NULL)); // randomize

    fillDeck(deck, face, suit); // load the deck with Cards
    shuffle(deck); // put Cards in random order
    deal(deck); // deal all 52 Cards
}

```

Figure Card Shuffling and Dealing Program Using Structures (3 of 4)

```
// place strings into Card structures
void fillDeck(Card * const wDeck, const char * wFace[],
    const char * wSuit[])
{
    // loop through wDeck
    for (size_t i = 0; i < CARDS; ++i) {
        wDeck[i].face = wFace[i % FACES];
        wDeck[i].suit = wSuit[i / FACES];
    }
}

// shuffle cards
void shuffle(Card * const wDeck)
{
    // loop through wDeck randomly swapping Cards
    for (size_t i = 0; i < CARDS; ++i) {
        size_t j = rand() % CARDS;
        Card temp = wDeck[i];
        wDeck[i] = wDeck[j];
        wDeck[j] = temp;
    }
}

```

Figure Card Shuffling and Dealing Program Using Structures (4 of 4)

```
// deal cards
void deal(const Card * const wDeck)
{
   // loop through wDeck
   for (size_t i = 0; i < CARDS; ++i) {
      printf("%5s of %-8s%s", wDeck[i].face , wDeck[i].suit ,
         (i + 1) % 4 ? "  " : "\n");
   }
}
```

Figure Output for the Card Shuffling and Dealing Simulation

```
Three of Hearts      Jack of Clubs      Three of Spades      Six of Diamonds
 Five of Hearts     Eight of Spades     Three of Clubs     Deuce of Spades
 Jack of Spades      Four of Hearts     Deuce of Hearts      Six of Clubs
Queen of Clubs      Three of Diamonds   Eight of Diamonds   King of Clubs
 King of Hearts     Eight of Hearts     Queen of Hearts     Seven of Clubs
Seven of Diamonds    Nine of Spades      Five of Clubs      Eight of Clubs
  Six of Hearts     Deuce of Diamonds    Five of Spades      Four of Clubs
Deuce of Clubs       Nine of Hearts     Seven of Hearts      Four of Spades
  Ten of Spades      King of Diamonds     Ten of Hearts      Jack of Diamonds
 Four of Diamonds     Six of Spades      Five of Diamonds     Ace of Diamonds
  Ace of Clubs       Jack of Hearts       Ten of Clubs      Queen of Diamonds
  Ace of Hearts       Ten of Diamonds    Nine of Clubs       King of Spades
  Ace of Spades      Nine of Diamonds   Seven of Spades     Queen of Spades
```

Example: High-Performance Card Shuffling and Dealing Simulation (2 of 3)

- In the program, function `fillDeck` initializes the `Card` array in order with “Ace” through “King” of each suit.
- The `Card` array is passed to function `shuffle`, where a shuffling algorithm is implemented.
- Function `shuffle` takes an array of 52 `Cards` as an argument.
- The function loops through the 52 `Cards`

Example: High-Performance Card Shuffling and Dealing Simulation (3 of 3)

- For each card, a number between 0 and 51 is picked randomly.
- Next, the current `Card` and the randomly selected `Card` are swapped in the array
- A total of 52 swaps are made in a single pass of the entire array, and the array of `Cards` is shuffled!
- Because the `Cards` were swapped in place in the array, the high-performance dealing algorithm implemented in function `deal` requires only **one** pass of the array to deal the shuffled `Cards`.

Common Programming Error

Forgetting to include the array index when referring to individual structures in an array of structures is a syntax error.

C Preprocessor Directives

Introduction

- The **C preprocessor** executes **before** a program is compiled.
- Actions:
  - inclusion of other files in the file being compiled,
  - definition of **symbolic constants** and **macros**,
  - **conditional compilation** of program code
  - **conditional execution of preprocessor directives.**
- Preprocessor directives begin with # and only whitespace characters and comments may appear before a preprocessor directive on a line.

Preprocessor Directives

Directive	Description
#include	Include a header file.
#define	Define a constant symbol.
#undef	Undefine a constant symbol.
#ifdef	If defined (test whether a symbol was defined.)
#ifndef	If not defined (test whether a symbol was not defined.)
#if	If (test whether a precompile condition is true.)
#else	Optinal else directive.
#elif	Optinal else if directive.
#endif	End of if directive.

The #include Preprocessor Directive

- `#include`
  - Copy of a specified file is included in place of the directive.
  - Must be used at the beginning of a program.
  - `#include <filename.h>`
    - Searches in standard library directory for header file
    - Use for C standard library files
  - `#include "filename.h"`
    - Searches in current directory, then in standard library directory
    - Use for user-defined files
  - include is used for:
    - Programs with multiple source files to be compiled together
    - Header file – has common declarations and definitions (structures, function prototypes)

Figure Some of the Standard Library Headers (1 of 2)

Header	Explanation
<assert.h>	Contains information for adding diagnostics that aid program debugging.
<ctype.h>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<errno.h>	Defines macros that are useful for reporting error conditions.
<float.h>	Contains the floating-point size limits of the system.
<limits.h>	Contains the integral size limits of the system.
<locale.h>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it’s running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
<math.h>	Contains function prototypes for math library functions.

Figure Some of the Standard Library Headers (2 of 2)

Header	Explanation
<signal.h>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<stdarg.h>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<stddef.h>	Contains common type definitions used by C for performing calculations.
<stdio.h>	Contains function prototypes for the standard input/output library functions, and information used by them.
<stdlib.h>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.
<string.h>	Contains function prototypes for string-processing functions.
<time.h>	Contains function prototypes and types for manipulating the time and date.

The #define Preprocessor Directive: Symbolic Constants

- Only the English letters, numbers and underscore are allowed for symbolic constant naming.
- Example:

```
#define  PI  3.14
```

- Example: Following defines a hexadecimal constant (0x is the prefix for hexadecimals)

```
#define  NUM  0x5F2B
```

- Example: Following defines a binary constant (0b is the prefix for binaries)

```
#define  NUM  0b0101111100101011
```

The #define Preprocessor Directive: Symbolic Constants

- After defining a symbolic constant, it can not be changed or redefined.

```
#define A  5
....
A = 8;            // Compiler error: Assignment is not allowed.
....
#define A  8      // Compiler error: Redefinition is not allowed.
```

Suffixes for Integer and Floating-Point Constants

- C provides suffixes (postfix) for constants
  - unsigned integer (u , U)
  - long integer (l , L)
  - unsigned long integer (ul, UL , lu, LU)
  - float (f , F)
  - double (lf , LF)

  - Examples:

```
#define CONSTANT1  17u
#define CONSTANT2  475L
#define CONSTANT3  3862ul
#define PI         3.14f
```

  - If a floating point constant is not suffixed, it is considered as `double`

    `#define PI  3.14` → PI is double by default

The #define Preprocessor Directive: Macros

- Macro
  - Operation defined in `#define`
  - A macro without arguments is treated like a symbolic constant.
  - **A macro with arguments is treated like a function.**
  - The arguments are substituted, when the macro is expanded.
  - Performs a text substitution – no data type checking
  - Recommended only for very short functions
  - The macro

```
#define CIRCLE_AREA(R) ( PI * (R) * (R) )
```

would cause

```
a = CIRCLE_AREA(4); // Original call statement
```

to become

```
a = ( 3.14 * (4) * (4) ); //Statement after preprocessing
```

The #define Preprocessor Directive: Macros

- Using the parentheses is very important.
  - Without them the macro

```
#define CIRCLE_AREA(R)  PI *  R  *  R
```

would cause

```
a = CIRCLE_AREA(c + 2);
```

to become

```
a = 3.14 * c + 2 * c + 2;
```

Wrong result

- Multiple arguments

```
#define RECTANGLE_AREA(x, y)  ( (x) * (y) )
```

would cause

```
r = RECTANGLE_AREA( a+4, b+7 );
```

to become

```
r= ( (a+4) * (b+7) );
```

The #undef Preprocessor Directive

- `#undef`
  - Undefines a symbolic constant or macro.
    (i.e. cancels a previosly defined constant or macro)
  - If a symbolic constant or macro has been undefined it can later be redefined again.

```
#include <stdio.h>
int main()
{
  #define  A   5
  printf("%d", A);
  #undef  A

  printf("%d", A);
}
```

Scope of the constant symbol A is limited between #define and #undef

Compiler error

Conditional Compilation (1 of 7)

- **Conditional compilation** enables you to control the execution of preprocessor directives and the compilation of program code.
- Each conditional preprocessor directive evaluates a constant integer expression.
- Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives.
- The conditional preprocessor construct is much like the `if` selection statement.

Conditional Compilation (2 of 7)

- Consider the following preprocessor code:

```
#if !defined(MY_CONSTANT)
   #define MY_CONSTANT 0
#endif
```

  Which determines whether `MY_CONSTANT` is `defined`—that is, whether `MY_CONSTANT` has already appeared in an earlier `#define` directive.

- The expression `defined(MY_CONSTANT)` evaluates to `1` if `MY_CONSTANT` is defined and `0` otherwise.
- If the result is `0`, `!defined(MY_CONSTANT)` evaluates to `1` and `MY_CONSTANT` is defined.
- Otherwise, the `#define` directive is skipped.

Conditional Compilation (3 of 7)

- Every `#if` construct ends with `#endif`.
- Directives `#ifdef` and `#ifndef` are shorthand for `#if defined`**(name)** and `#if !defined`**(name).**
- A multiple-part conditional preprocessor construct may be tested by using the `#elif` (the equivalent of `else if` in an `if` statement) and the `#else` (the equivalent of `else` in an `if` statement) directives.
- These directives are frequently used to **prevent header files from being included multiple times in the same source file**.

Conditional Compilation (4 of 7)

- During program development, it's often helpful to "comment out" portions of code to prevent them from being compiled.
- If the code contains multiline comments, `/*` and `*/` cannot be used to accomplish this task, because such comments cannot be nested.
- Instead, you can use the following preprocessor construct:

```
#if 0
    code prevented from compiling
#endif
```

- To enable the code to be compiled, replace the `0` in the preceding construct with `1`.

Conditional Compilation (5 of 7)

- Conditional compilation is commonly used as a **debugging** aid.
- Many C implementations include **debuggers**, which provide much more powerful features than conditional compilation.
- If a debugger is not available, `printf` statements are often used to print variable values and to confirm the flow of control.
- These `printf` statements can be enclosed in conditional preprocessor directives so the statements are compiled only while the debugging process is not completed.

Conditional Compilation (6 of 7)

- For example,

```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

causes a `printf` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined (`#define DEBUG`) before directive `#ifdef DEBUG`.

Conditional Compilation (7 of 7)

- When debugging is completed, the `#define` directive is removed from the source file (or commented out) and the `printf` statements inserted for debugging purposes are ignored during compilation.
- In larger programs, it may be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.
- Many compilers allow you to define and undefine symbolic constants with a compiler flag so that you do not need to change the code.

Examples: #if and #ifdef

PROGRAM1

```
#include <stdio.h>

#define VERSION 5

int main()
{
#ifdef VERSION
  printf("Version : %d ", VERSION);
#else
  printf("Version unknown");
#endif

  printf("\n");
}
```

Program Output

```
Version : 5
```

PROGRAM2

```
#include <stdio.h>

#define VERSION 2

int main()
{
#if VERSION == 1
  printf("First version");
#elif VERSION == 2
  printf("Second version");
#else
  printf("Version unknown");
#endif

  printf("\n");
}
```

Program Output

```
Second version
```

Secure C Programming (2 of 3)

- For example, if we call `CIRCLE_AREA` as follows:

```
result = CIRCLE_AREA(++radius);
```

the call to the macro `CIRCLE_AREA` is expanded to:

```
result = ((3.14159) * (++radius) * (++radius));
```

which increments radius twice in the statement.

- In addition, the result of the preceding statement is undefined because C allows a variable to be modified **only once** in a statement.

Secure C Programming (3 of 3)

- In a function call, the argument is evaluated **only once before** it’s passed to the function.
- So, functions are always preferred to unsafe macros.

Using Command-Line Arguments (1 of 4)

- On many systems, it's possible to pass arguments to main from a command line by including parameters int argc and char *argv[] in the parameter list of main.
- Parameter argc receives the number of command-line arguments that the user has entered.
- Parameter argv is an array of strings in which the actual command-line arguments are stored.
- Common uses of command-line arguments include passing options to a program and passing filenames to a program.

Example: multiply.c

```
#include <stdio.h>
#include <stdlib.h>  // for strtol function

int main (int argc, char * argv [ ] )
{
    int num1, num2; // Local variables

    if ( argc != 3)
    {
       printf(“Inappropriate arguments! \n");
       printf(“Example usage : multiply number1  number2 \n");
       return 0; // stop
    }

    char **endptr = NULL;
    num1 = (int) strtol( argv[1], endptr, 0);    // convert ascii string to int
    num2 = (int) strtol( argv[2], endptr, 0);    // convert ascii string to int

    printf("%d * %d = %d \n", num1, num2, num1 * num2);

} // end main
```

Running program from command-line

- The following command-line instruction should be used to run the program.
- The "C:>" symbol is the automatic command-line prompt in Windows operating system.

Program screen output

```
C:> multiply  50  7

50 * 7 = 350
```

- The arguments in main program are automatically assigned as the followings:

Variable	Value
argc	3
argv[0]	"multiply"
argv[1]	"50"
argv[2]	"7"

Using Command-Line Arguments (2 of 4)

- We assume that the executable file for the program is called `mycopy`.
- A typical command line for the `mycopy` program on a Linux/UNIX system is

```
$ mycopy input output
```

- This command line indicates that file `input` is to be copied to file `output`.
- When the program is executed, `if argc` is not `3` (`mycopy` counts as one of the arguments), the program prints an error message and terminates.
- Otherwise, array `argv` contains the strings "`mycopy`", "`input`" and "`output`".

Using Command-Line Arguments (3 of 4)

- The second and third arguments on the command line are used as file names by the program.
- The files are opened using function `fopen`.
- If both files are opened successfully, characters are read from file `input` and written to file `output` until the end-of-file indicator for file `input` is set.
- Then the program terminates.
- The result is an exact copy of file `input` (if no errors occur during processing.
- See your system documentation for more information on command-line arguments.

Example: Using Command-Line Arguments (1 of 2)

```
// Fig. 14.3: fig14_03.c
// Using command-line arguments
#include <stdio.h>

int main(int argc, char *argv[])
{
   // check number of command-line arguments
   if (argc != 3) {
      puts("Usage: mycopy infile outfile");
   }
   else {
      FILE *inFilePtr; // input file pointer

      // try to open the input file
      if ((inFilePtr = fopen(argv[1], "r")) != NULL) {
         FILE *outFilePtr; // output file pointer

         // try to open the output file
         if ((outFilePtr = fopen(argv[2], "w")) != NULL) {
            int c; // holds characters read from source file

```

Example: Using Command-Line Arguments (2 of 2)

```
            // read and output characters
            while ((c = fgetc(inFilePtr)) != EOF) {
               fputc(c, outFilePtr);
            }

            fclose(outFilePtr); // close the output file
         }
         else { // output file could not be opened
            printf("File \"%s\" could not be opened\n", argv[2]);
         }

         fclose(inFilePtr); // close the input file
      }
      else { // input file could not be opened
         printf("File \"%s\" could not be opened\n", argv[1]);
      }
   }
}
```

Compiling Multiple-Source-File Programs

- Program can contain multiple source files
  - Main program and other function definitions can be in a separate files
  - Global variables accessible to functions in same file
    - Global variables must be defined in every file in which they are used
  - Example:
    - If integer `num` is defined in one file
    - To use it in another file you must write the statement

```
extern int num;
```

  - `extern`
    - States that the variable is defined in another source file (external)

Example: Multiple-Source-Files and using extern specifier

part1.c

```c
#include <stdio.h>

int main()
{
  extern int a;

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

part2.c

```c
// Global variable:

int a = 50;

...
...
```

Compiling two C source files and generating an executable file from the command-line window:

```
gcc  -std=c99  -Wall  -Werror  part1.c  part2.c  -o  myprog
```

C Project Files

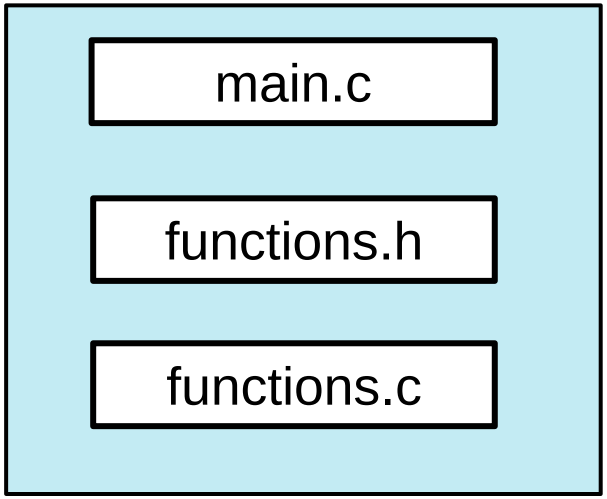
- A project can contain multiple C source files (*.c) and header files (*.h) .

- Only one file should contain the main() function.
- Other files can contain the C functions written by the programmer.
- This modular method is useful when the source code is too long.

Example: Project file components

- Each source file in the project can be considered as a program module.

Project files



main.c

```c
#include <stdio.h>
#include "functions.h"

int main() {
  int choice;
  while(1) { // Infinite loop
     printf(“Tasks \n");
     printf("1. Enter Grades \n");
     printf("2. Compute Average \n");
     printf("3. Quit \n");
     printf("   Your choice: ");
     scanf("%d", &choice);

     switch (choice) {
       case 1: read_grade();      break;
       case 2: compute_avg();     break;
       case 3: return 0;        //Stop
       default: printf(" Invalid choice! \n");
     } // end switch
    } // end while
    printf(" Program finished. \n");
} // end main
```

functions.h

```c
//functions.h

// Function prototypes:
void read_grade();
void compute_avg();
```

functions.c

```c
#include <stdio.h>
#include "functions.h"

//Global variables:
int grades[50]; // Array
int counter;    // Loop counter as array index


void read_grade()
{
   printf("Enter grades (-1 to finish) : ");
   counter=0;
   do
   {
     scanf("%d", &grades[counter]);
     counter++;
   } while (grades[counter-1] != -1);

   counter--;
   //The last data was -1 (sentinel)

} // end function
```

```c
void compute_avg()
{
 int i, tot=0;

 for (i=0; i < counter; i++)
     tot += grades[i];

 printf("Average = %f \n",
      (float) tot / counter);
}
```

Bit Manipulations

- Bitwise operators are used for low level bit operations.

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

- Bitwise operators can be applied on integer values and variables.

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.

Truth Tables for Bitwise Operators

Bit 1	Bit 2	Bit 1 & Bit 2	Bit 1 | Bit 2	Bit 1 ^ Bit 2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bit	~ Bit
0	1
1	0

Example: Bitwise Logical Operators

```c
#include <stdio.h>
int main() {
  unsigned char num1, num2;
  num1 = 3;
  num2 = 5;
  printf("%d \n", num1 & num2);                  //bitwise and
  printf("%d \n", num1 | num2);                  //bitwise or
  printf("%d \n", num1 ^ num2);                  //bitwise xor
  printf("%d \n", (unsigned char) ~num1);   //bitwise not
  printf("%d \n", (unsigned char) ~num2);   //bitwise not
}
```

Program Output

```
1
7
6
252
250
```

Variable/Expression	Binary value (8 bit)	Decimal value
num1	00000011	3
num2	00000101	5
num1 & num2	00000001	1
num1 | num2	00000111	7
num1 ^ num2	00000110	6
~ num1	11111100	252
~ num2	11111010	250

Example: Bitwise Shifting

```c
#include <stdio.h>

int main()
{
  unsigned short int num = 1;
 //num has 16 bits length (2 bytes).
 //Maximum value of Num can be 65535.


 int i; //Loop counter

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


 for (i=1; i <= 16; i++)
 {
   num = num << 1;
   printf("%d.shift  num=%d \n",
          i, num);
 }

}
```

Program Output

```
Num = 1

1.shift  Num=2
2.shift  Num=4
3.shift  Num=8
4.shift  Num=16
5.shift  Num=32
6.shift  Num=64
7.shift  Num=128
8.shift  Num=256
9.shift  Num=512
10.shift  Num=1024
11.shift  Num=2048
12.shift  Num=4096
13.shift  Num=8192
14.shift  Num=16384
15.shift  Num=32768
16.shift  Num=0
```

Overflow

One left shift means multiplying by 2.

Operator Precedences

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

Fig. 10.15 Operator precedence and associativity.

Variable-Length Argument Lists (1 of 5)

- It's possible to create functions that receive an unspecified number of arguments.
- Most programs in the text have used the standard library function `printf`, which, as you know, takes a variable number of arguments.
- As a minimum, `printf` must receive a string as its first argument, but `printf` can receive any number of additional arguments.
- The function prototype for `printf` is
  - `int printf(const char *format, ...);`
- The **ellipsis (…)** in the prototype indicates that the function receives a **variable number of arguments of any type.**

Variable-Length Argument Lists (2 of 5)

- The ellipsis must always be placed at the end of the parameter list.
- The macros and definitions of the variable arguments headers <stdarg.h> provide the capabilities necessary to build functions with variable-length argument lists.
- Figure demonstrates function average that receives a variable number of arguments.
- The first argument of average is always the number of values to be averaged.

Figure stdarg.h Variable-Length Argument-List Type and Macros

Identifier	Explanation
`va_list`	A **type** suitable for holding information needed by macros `va_start`, `va_arg` and `va_end`. To access the arguments in a variable-length argument list, an object of type `va_list` must be defined.
`va_start`	A **macro** that's invoked before the arguments of a variable-length argument list can be accessed. The macro initializes the object declared with `va_list` for use by the `va_arg` and `va_end` macros.
`va_arg`	A **macro** that expands to the value of the next argument in the variablelength argument list—the value has the type specified as the macro's second argument. Each invocation of `va_arg` modifies the object declared with `va_list` so that it points to the next argument in the list.
`va_end`	A **macro** that facilitates a normal return from a function whose variablelength argument list was referred to by the `va_start` macro.

Figure Using Variable-Length Argument Lists (1 of 3)

```c
// Fig. 14.2: fig14_02.c
// Using variable-length argument lists
#include <stdio.h>
#include <stdarg.h>

double average(int i, ...); // prototype

int main(void)
{
   double w = 37.5;
   double x = 22.5;
   double y = 1.7;
   double z = 10.2;

   printf("%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
      "w = ", w, "x = ", x, "y = ", y, "z = ", z);
   printf("%s%.3f\n%s%.3f\n%s%.3f\n",
      "The average of w and x is ", average(2, w, x),
      "The average of w, x, and y is ", average(3, w, x, y),
      "The average of w, x, y, and z is ",
      average(4, w, x, y, z));
}

```

Figure Using Variable-Length Argument Lists (2 of 3)

```
// calculate average
double average(int i, ...)
{
   double total = 0; // initialize total
   va_list ap; // stores information needed by va_start and va_end

   va_start(ap, i); // initializes the va_list object

   // process variable-length argument list
   for (int j = 1; j <= i; ++j) {
      total += va_arg(ap, double);
   }

   va_end(ap); // clean up variable-length argument list
   return total / i; // calculate average
}
```

Figure Using Variable-Length Argument Lists (3 of 3)

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Variable-Length Argument Lists (3 of 5)

- Function `average` uses all the definitions and macros of header `<stdarg.h>`.
- Object ap, of type **`va_list`**, is used by macros **`va_start`, `va_arg`** and **`va_end`** to process the variable-length argument list of function `average`.
- The function begins by invoking macro `va_start` to initialize object `ap` for use in `va_arg` and `va_end`.
- The macro receives two arguments—object `ap` and the identifier of the rightmost argument in the argument list **before** the ellipsis—`i` in this case (`va_start` uses `i` here to determine where the variable-length argument list begins).

Variable-Length Argument Lists (4 of 5)

- Next, function `average` repeatedly adds the arguments in the variable-length argument list to `total`.
- The value to be added to `total` is retrieved from the argument list by invoking macro `va_arg`.
- Macro `va_arg` receives two arguments—object `ap` and the type of the value expected in the argument list—`double` in this case.
- The macro returns the value of the argument.
- Function `average` invokes macro `va_end` with object `ap` as an argument to facilitate a normal return to `main` from `average`.
- Finally, the average is calculated and returned to `main`.

Common Programming Error

Placing an ellipsis in the middle of a function parameter list is a syntax error—an ellipsis may be placed only at the end of the parameter list.

Variable-Length Argument Lists (5 of 5)

- The reader may question how function `printf` and function `scanf` know what type to use in each `va_arg` macro.
- The answer is that they scan the format conversion specifiers in the format control string to determine the type of the next argument to be processed.

Assertions (1 of 4)

- The **`assert`** macro—defined in the **`<assert.h>`** header—tests the value of an expression at execution time.
- If the value of the expression is false `(0)`, `assert` prints an error message and calls function **`abort`** (of the general utilities library—`<stdlib.h>`) to terminate program execution.
- This is a useful **debugging** tool for testing whether a variable has a correct value.
- For example, suppose variable x should never be larger than `10` in a program.

Assertions (2 of 4)

- An assertion may be used to test the value of x and print an error message if the value of x is incorrect.
- The statement would be

```
assert(x <= 10);
```

- If x is greater than `10` when the preceding statement is encountered in a program, an error message containing the line number and filename is printed and the program **terminates**.

Assertions (3 of 4)

- You may then concentrate on this area of the code to find the error.
- If the symbolic constant `NDEBUG` is defined, subsequent assertions will be **ignored**.
- Thus, when assertions are no longer needed, the line

```
#define NDEBUG
```

  is inserted in the program file rather than each assertion being deleted manually.

Secure C Programming (1 of 3)

- The CIRCLE_AREA macro defined in Section

```c
#define CIRCLE_AREA(x) ((PI) * (x) * (x))
```

  is considered to be an unsafe macro because it evaluates its argument x more than once.
- This can cause subtle errors.
- If the macro argument contains side effects—such as incrementing a variable or calling a function that modifies a variable’s value—those side effects would be performed multiple times.

and ## Operators (1 of 3)

- The # and ## preprocessor operators are available in Standard C.
- The # operator causes a replacement text token to be converted to a string surrounded by quotes.
- Consider the following macro definition:
  - `#define HELLO(x) printf("Hello, " #x "\n");`
- When `HELLO(John)` appears in a program file, it's expanded to
  - `printf("Hello, " "John" "\n");`
- The string `"John"` replaces `#x` in the replacement text.

and ## Operators (2 of 3)

- Strings separated by white space are concatenated during preprocessing, so the preceding statement is equivalent to
  - `printf("Hello, John\n");`
- The # operator must be used in a macro with arguments because the operand of # refers to an argument of the macro.
- The ## operator concatenates two tokens.

and ## Operators (2 of 3)

```
#include <stdio.h>
// Macro definition using stringification (#)
// convert the argument x into a string
#define PRINT_STRING(x) printf(#x)
int main() {
   // Using the macro
   PRINT_STRING(Hello Geeks);
return 0;  }
```

and ## Operators (3 of 3)

- Consider the following macro definition:
  - `#define TOKENCONCAT(x, y)  x ## y`
- When `TOKENCONCAT` appears in the program, its arguments are concatenated and used to replace the macro.
- For example, `TOKENCONCAT(O, K)` is replaced by `OK` in the program.
- The ## operator must have two operands

and ## Operators (3 of 3)

```
#include <stdio.h>

// Macro definition using the Token-pasting operator
#define concat(a, b) a##b
int main(void) {
    int xy = 30;

    // Printing the concatenated value of x and y
    printf("%d", concat(x, y));
    return 0;
}
```

Line Numbers (1 of 2)

- The **`#line` preprocessor directive** causes the subsequent source code lines to be renumbered starting with the specified constant integer value.
- The directive

```
#line 100
```

  starts line numbering from `100` beginning with the next source code line.
- A filename can be included in the `#line` directive.

Line Numbers (2 of 2)

- The directive

```
#line 100 "file1.c"
```

  indicates that lines are numbered from `100` beginning with the next source code line and that the name of the file for the purpose of any compiler messages is `"file1.c"`.

- The directive normally is used to help make the messages produced by syntax errors and compiler warnings more meaningful.
- The line numbers do not appear in the source file.

Predefined Symbolic Constants

- Standard C provides predefined symbolic constants, several of which are shown in Figure ——the rest are in Section .8 of the C standard document.
- The identifiers for each of the predefined symbolic constants begin and end with two underscores.
- These identifiers and the defined identifier (used in Section) cannot be used in #define or #undef directives.

Figure Some Predefined Symbolic Constants

Symbolic constant	Explanation
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The name of the source file (a string).
`__DATE__`	The date the source file was compiled (a string of the form "`Mmm dd yyyy`" such as "`Jan 19 2002`").
`__TIME__`	The time the source file was compiled (a string literal of the form "`hh:mm:ss`").
`__STDC__`	The value `1` if the compiler supports Standard C; `0` otherwise. Requires the compiler flag `/Za` in Visual C++.