Today

- **Structures**
  - Structure Definitions
  - Initializing Structures
  - Accessing Members of Structures
  - typedef
  - Using Structures With Functions
  - Structures and Pointers
  - Assignments
  - Arrays of Structures
- **Linked Lists**
- **Unions**
  - Union definitions
  - Union operations
- **Enumeration Constants**

**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
- Combined with pointers, can create linked lists, stacks, queues, and trees
Structure Definitions

Example 1:

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

- `struct` introduces the definition for structure `card`
- `card` is the structure name and is used to declare variables of the structure type
- `card` contains two members of type `char *`
  - These members are `face` and `suit`

Example 2:

```c
struct point {
    int x;
    int y;
};

struct point pt; /* defines a variable pt which is a structure of type struct point */

pt.x = 15;
pt.y = 30;
printf("%d, %d", pt.x, pt.y);
```

Structure Definitions

- `struct` information
  - Can contain a member that is a pointer to the same structure type
  - A structure definition does not reserve space in memory
    - Instead creates a new data type used to define structure variables

- Definitions
  - Defined like other variables:
    ```c
    struct card oneCard, deck[ 52 ], *cPtr;
    ```
  - Can use a comma separated list:
    ```c
    struct card {
        char *face;
        char *suit;
    } oneCard, deck[ 52 ], *cPtr;
    ```

/* Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners. */

```c
struct rect {
    struct point pt1;
    struct point pt2;
};

struct rect screen;

/* Print the pt1 field of screen */
printf("%d, %d", screen.pt1.x, screen.pt1.y);

/* Print the pt2 field of screen */
printf("%d, %d", screen.pt2.x, screen.pt2.y);
```
Structure 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

Initializing Structures

- Initializer lists
  - Example:
    ```c
    struct card oneCard = { "Three", "Hearts" }; 
    ```
- Assignment statements
  - Example:
    ```c
    struct card threeHearts = oneCard;
    ```
  - Could also define and initialize threeHearts as follows:
    ```c
    struct card threeHearts;
    threeHearts.face = "Three";
    threeHearts.suit = "Hearts";
    ```

Accessing Members of Structures

- Dot operator (.) used with structure variables
  ```c
  struct card myCard;
  printf( "%s", myCard.suit );
  ```
- Arrow operator (->) used with pointers to structure variables
  ```c
  struct card *myCardPtr = &myCard;
  printf( "%s", myCardPtr->suit );
  ```
- `myCardPtr->suit` is equivalent to
  ```c
  ( *myCardPtr ).suit
  ```

```c
#include <stdio.h>
/* card structure definition */
struct card {
    char *face; /* define pointer face */
    char *suit; /* define pointer suit */
}; /* end structure card */

int main() {
    struct card a; /* define struct a */
    struct card *aPtr; /* define a pointer to card */
    /* place strings into card structures */
    a.face = "Ace";
    a.suit = "Spades";
    aPtr = &a; /* assign address of a to aPtr */
    printf( "%s\n%s\n%s\n", a.face, " of ", a.suit,
            aPtr->face, " of ", aPtr->suit, ( *aPtr ).face, " of ",
            ( *aPtr ).suit );
    return 0; /* indicates successful termination */
} /* end main */
```
typedef
- Creates synonyms (aliases) for previously defined data types
- Use typedef to create shorter type names

Example:
```c
typedef struct point pixel;
``` - Defines a new type name `pixel` as a synonym for type `struct point`

```c
typedef struct Card *CardPtr;
``` - Defines a new type name `CardPtr` as a synonym for type `struct Card *`

- `typedef` does not create a new data type
  - Only creates an alias

---

Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
  - Or, pass individual members
  - Both pass call by value

- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it

- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

---

Using Structures with Functions 1

```c
#include<stdio.h> /* Demonstrates passing a structure to a function */

struct data{
   int amount;
   char fname[30];
   char lname[30];
}rec;

void printRecord(struct data x){
   printf("Donor%s %s gave $%d
", x.fname, x.lname, x.amount);
}

int main(void){
   printf("Enter the donor's first and last names\n");
   scanf("%s %s",rec.fname, rec.lname);
   printf("Enter the donation amount: ");
   scanf("%d",&rec.amount);
   printRecord(rec);
   return 0;
}
```

Using Structures with Functions 2

```c
/* Make a point from x and y components. */

struct point makepoint (int x, int y)
{
   struct point temp;
   temp.x = x;
   temp.y = y;
   return (temp);
}

/* makepoint can now be used to initialize a structure */

struct rect screen;
struct point middle;

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(50,100);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```
/* add two points */
struct point addpoint (struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}

Both arguments and the return value are structures in the function addpoint.

Structures and Pointers

- Pointers to structures are so frequently used that an alternative is provided as a shorthand.
- If \( p \) is a pointer to a structure, then
  \( p \rightarrow \text{field_of_structure} \) refers to a particular field.
- We could write
  \[
  \text{printf(“Origin is (%d %d)\n”, } p\rightarrow x, p\rightarrow y);
  \]

Structures and Pointers

- Both . and -> associate from left to right
- Consider
  \[
  \text{struct rect } r, *rp = &r;
  \]
- The following 4 expressions are equivalent.
  \[
  r.pt1.x \\
  rp \rightarrow pt1.x \\
  (r.pt1).x \\
  (rp\rightarrow pt1).x
  \]
  \[
  \text{struct rect }
  \{
  \text{struct point pt1;}
  \text{struct point pt2;}
  \};
  \]

- Parenthesis are necessary in \( (*p).x \) because the precedence of the structure member operator (dot) is higher than *.
- The expression \( *p.x \equiv *(p.x) \) which is illegal because \( x \) is not a pointer.
Assignments

```c
struct student {
    char *last_name;
    int student_id;
    char grade;
};
struct student temp, *p = &temp;
```

```c
temp.grade = 'A';
temp.last_name = "Casanova";
temp.student_id = 590017;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equiv. Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp.grade</td>
<td>p -&gt; grade</td>
<td>A</td>
</tr>
<tr>
<td>temp.last_name</td>
<td>p -&gt; last_name</td>
<td>Casanova</td>
</tr>
<tr>
<td>temp.student_id</td>
<td>p -&gt; student_id</td>
<td>590017</td>
</tr>
<tr>
<td>(*p).student_id</td>
<td>p -&gt; student_id</td>
<td>590017</td>
</tr>
</tbody>
</table>

Arrays of Structures

- Usually a program needs to work with more than one instance of data.
- For example, to maintain a list of phone #s in a program, you can define a structure to hold each person’s name and number.

```c
struct entry {
    char fname[10];
    char lname[12];
    char phone[8];
};
```

- A phone list has to hold many entries, so a single instance of the entry structure isn’t of much use. What we need is an array of structures of type entry.

- After the structure has been defined, you can define the array as follows:

```c
struct entry list[1000];
```
To assign data in one element to another array element, you write
\[
\text{list[1] = list[5];}
\]

To move data between individual structure fields, you write
\[
\text{strcpy(list[1].phone, list[5].phone);}
\]

To move data between individual elements of structure field arrays, you write
\[
\text{list[5].phone[1] = list[2].phone[3];}
\]

Arrays of structures can be very powerful programming tools, as can pointers to structures.

```c
struct part {
    int number;
    char name[10];
};

struct part data[100];
struct part *p_part;

p_part = data;
printf("%d %s", p_part->number, p_part->name);
```

The above diagram shows an array named `x` that consists of 3 elements. The pointer `ptr` was initialized to point at `x[0]`. Each time `ptr` is incremented, it points at the next array element.
/* Array of structures */
#include <stdio.h>
#define MAX 4

struct part {
    int number;
    char name[10];

int main (void) {
    struct part *p_part;
    int count;
    p_part = data;
    for (count = 0; count < MAX; count++) {
        printf("\n%d %s", p_part -> number, p_part -> name);
        p_part++;
    }
    return 0;
}

**Today**
- Structures
  - Structure Definitions
  - Initializing Structures
  - Accessing Members of Structures
  - typedef
  - Using Structures With Functions
  - Structures and Pointers
  - Assignments
  - Arrays of Structures
- Linked Lists
- Unions
  - Union definitions
  - Union operations
- Enumeration Constants

**Introduction**
- Dynamic data structures
  - Data structures that grow and shrink during execution
- Linked lists
  - Allow insertions and removals anywhere
- Stacks
  - Allow insertions and removals only at top of stack
- Queues
  - Allow insertions at the back and removals from the front
- Binary trees
  - High-speed searching and sorting of data and efficient elimination of duplicate data items

**Self-Referential Structures**
- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a NULL pointer (0)
    ```c
    struct node {
        int data;
        struct node *nextPtr;
    }
    ```
- nextPtr
  - Points to an object of type node
  - Referred to as a link
    - Ties one node to another node
Dynamic Memory Allocation

Two self-referential structures linked together

\[
\begin{array}{cc}
15 & \bullet \\
10 &
\end{array}
\]

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
  - `malloc`
    - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
    - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example
    ```c
    newPtr = malloc( sizeof( struct node ) );
    ```
  - `free`
    - Deallocates memory allocated by `malloc`
    - Takes a pointer as an argument
    ```c
    free( newPtr );
    ```

Linked Lists

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to `NULL` to mark the list’s end
- Use a linked list instead of an array when
  - You have an unpredictable number of data elements
  - Your list needs to be sorted quickly

Linked Lists

![Graphical representation of a linked list](image-url)

*Fig. 12.2* A graphical representation of a linked list.
#include <stdio.h>
#include <stdlib.h>

/* self-referential structure */
struct listNode {
    char data;   /* define data as char */
    struct listNode *nextPtr; /* listNode pointer */
}; /* end structure listNode */

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

/* prototypes */
void insert( ListNodePtr *sPtr, char value);
char delete( ListNodePtr *sPtr, char value);
int isEmpty( ListNodePtr sPtr);
void printList( ListNodePtr currentPtr);
void instructions( void);

int main(){
    ListNodePtr startPtr = NULL; /* initialize startPtr */
    int choice;   /* user's choice */
    char item;    /* char entered by user */

    instructions(); /* display the menu */
    printf( "? " );
    scanf( "%d", &choice );
    /* loop while user does not choose 3 */
    while ( choice != 3 ) {
        switch ( choice ) {
            case 1:
                printf( "Enter a character: " );
                scanf( "%c", &item );
                insert( &startPtr, item );
                printList( startPtr );
                break;
            case 2:
                /* default: */
                printf( "Invalid choice.\n\n" );
                break;
            default: /* end switch */
        }
    }
    /* end if */
}

/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, char value ) {
    ListNodePtr newPtr;   /* pointer to new node */
    ListNodePtr previousPtr; /* pointer to previous node in list */
    ListNodePtr currentPtr; /* pointer to current node in list */

    newPtr = malloc( sizeof( ListNode ) );
    if ( newPtr != NULL ) { /* is space available */
        newPtr->data = value;
        newPtr->nextPtr = NULL;
        previousPtr = NULL;
        currentPtr = *sPtr;

        /* loop to find the correct location in the list */
        while ( currentPtr != NULL && value > currentPtr->data ) {
            previousPtr = currentPtr;
            /* walk to ... */
            currentPtr = currentPtr->nextPtr; /* ... next node */
        } /* end while */
        /* insert newPtr at beginning of list */
        if ( previousPtr == NULL ) {
            newPtr->nextPtr = *sPtr;
            *sPtr = newPtr;
        } /* end if */
        else { /* insert newPtr between previousPtr and currentPtr */
            previousPtr->nextPtr = newPtr;
            newPtr->nextPtr = currentPtr;
        } /* end else */
    } /* end if */
} /* end function insert */
Enter your choice:
1 to insert an element into the list.
2 to delete an element from the list.
3 to end.

? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A
The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

Linked Lists

Unions
- `union`
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a union share space
  - Conserves storage
  - Only the last data member defined can be accessed

- `union` definitions
  - Same as `struct`
    ```
    union Number {
      int x;
      float y;
    };
    union Number value;
    ```

Today
- Structures
  - Structure Definitions
  - Initializing Structures
  - Accessing Members of Structures
  - `typedef`
  - Using Structures With Functions
  - Structures and Pointers
  - Assignments
  - Arrays of Structures
- Linked Lists
- Unions
  - `union` definitions
  - `union` operations
- Enumeration Constants
Unions

Valid union operations
- Assignment to union of same type: =
- Taking address: &
- Accessing union members: .
- Accessing members using pointers: ->

```c
#include <stdio.h>

/* number union definition */
union number {
    int x;    /* define int x */
    double y; /* define double y */
}; /* end union number */

/* end main */
```

```c
int main(){
    union number value; /* define union value */
    value.x = 100; /* put an integer into the union */
    printf( "%s\n%s\n%s%d\n%s%f\n\n", "Put a value in the integer member", "and print both members.", "int: ", value.x, "double: \n", value.y);
    value.y = 100.0; /* put a double into the same union */
    printf( "%s\n%s\n%s%d\n%s%f\n\n", "Put a value in the floating member", "and print both members.", "int: ", value.x, "double: \n", value.y);
    return 0; /* indicates successful termination */
} /* end main */
```

Put a value in the integer member and print both members.
int: 100
double: -9223372036854775808

Put a value in the floating member and print both members.
int: 0
double: 100.000000

Today

- Structures
  - Structure Definitions
  - Initializing Structures
  - Accessing Members of Structures
  - typedef
  - Using Structures With Functions
- Union Operations
  - Assignments
  - Arrays of Structures
- Linked Lists
- Unions
  - Union definitions
  - Union operations
- Enumeration Constants
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

---

```c
#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 };

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

    for ( month = JAN; month <= DEC; month++ )
        printf( "%2d%11s\n", month, monthName[ month ] );

    return 0; /* indicates successful termination */
} /* end main */
```

---

**Summary**

- Structures
- Linked Lists
- Unions
- Enumeration Constants
Next week

- File Input and Output
- Strings