Computer programming

"An expert is a man who has made all the mistakes which can be made, in a narrow field."

Niels Bohr

Outline

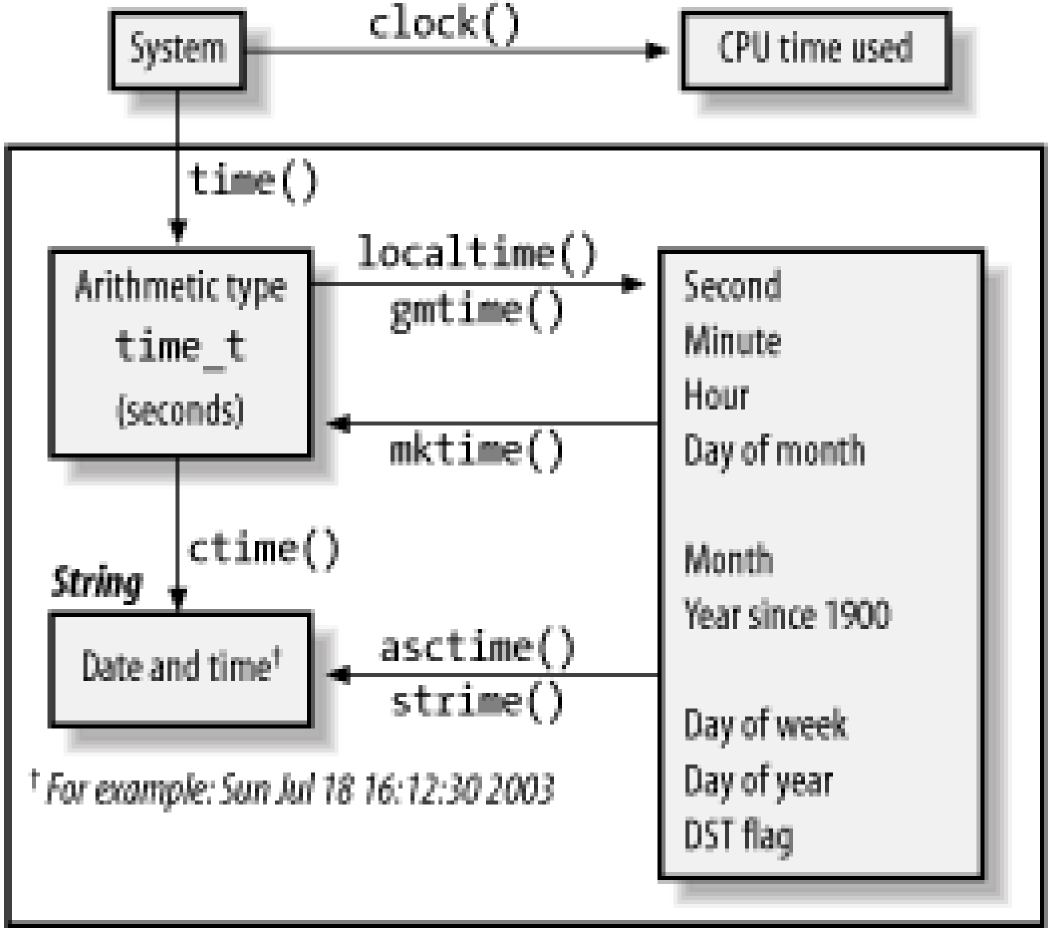
- Working with time
- I/O redirection
- Variable length argument lists
- Command line arguments
- Self referential structures
  - Lists
  - Stacks
  - Queues

Date and Time: time.h

- The `time.h` header file defines two macros.
  - NULL, representing the null pointer.
  - CLOCKS_PER_SEC: clocks()/CLOCKS_PER_SEC=time in seconds.
- Types Defined in time.h

Type	Description
size_t	The integer type returned by the sizeof operator
clock_t	An arithmetic type suitable to represent time
time_t	An arithmetic type suitable to represent time
struct tm	A structure type for holding components of calendar time

Usage of time and date functions



Date and Time: time.h. Broken down time: struct_tm

Member	Description
`int tm_sec`	Seconds after the minute (0–61)
`int tm_min`	Minutes after the hour (0–59)
`int tm_hour`	Hours after midnight (0–23)
`int tm_mday`	Day of the month (0–31)
`int tm_mon`	Months since January (0–11)
`int tm_year`	Years since 1900
`int tm_wday`	Days since Sunday (0–6)
`int tm_yday`	Days since January 1 (0–365)
`int tm_isdst`	Daylight Savings Time flag (greater than zero value means DST is in effect; zero means not in effect; negative means information not available)

Date and Time: time.h. Time functions:

Prototype	Description
clock_t clock(void);	Returns best approximation of the processor time elapsed since the program was invoked. Returns (clock_t)(-1) if the time is not available or representable.
double difftime(time_t t1, time_t t0);	Calculates the difference (t1 - t0) between two calendar times; expresses the result in seconds and returns the result.
time_t mktime(struct tm *tmptr);	Converts the broken-down time in the structure pointed to by tmptr into a calendar time; out-of-range values are adjusted (for example, 2 minutes, 100 seconds becomes 3 minutes, 40 seconds) and tm_wday and tm_yday are set to the values implied by the other members. Returns (time_t)(-1) if the calendar time cannot be represented; otherwise, returns the calendar time in time_t format.

Date and Time: time.h. Time functions:

Prototype	Description
time_t time(time_t *ptm)	Returns the current calendar time and also places it in the location pointed to by ptm, provided ptm is not NULL. Returns (time_t)(-1) if the calendar time is not available.
char *asctime(const struct tm *tmpt);	Converts the broken-down time in the structure pointed to by tmpt into a string of the form Thu Feb 26 13:14:33 1998\n\0 and returns a pointer to that string.
char *ctime(const time_t *ptm);	Converts the calendar time pointed to by ptm into a string in the form Wed Aug 11 10:48:24 1999\n\0 and returns a pointer to that string.

Date and Time: time.h. Time functions:

Prototype	Description
struct tm *gmtime(const time_t *ptm);	Converts the calendar time pointed to by ptm into a broken-down time, expressed as Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time (GMT), and returns a pointer to a structure holding that information. Returns NULL if UTC is not available.
struct tm *localtime(const time_t *ptm);	Converts the calendar time pointed to by ptm into a broken-down time, expressed as local time. Stores a tm structure and returns a pointer to that structure.

Date and Time: time.h. Time functions:

Prototype	Description
size_t strftime(char * restrict s, size_t max const char * restrict fmt, const struct tm * restrict tmpt);	Copies string fmt to string,s, replacing format specifiers in fmt with appropriate data derived from the contents of the broken-down time structure pointed to by tmpt; no more than max characters are placed into s. The function returns the number of characters placed (excluding the null character); if the resulting string (including null character) is larger than max characters, the function returns 0 and the contents of s are indeterminate.

Redirecting Input/Output on UNIX and Windows Systems

- Standard I/O : stdin – stdout
  - Redirect input and output
- Redirect symbol (<)
  - Operating system feature, NOT C++ feature
  - UNIX and Windows
  - `$` or `%` represents command line

  Example:

  ```
  $ myProgram < input
  ```

  - Rather than inputting values by hand, read them from a file
- Pipe command (|)
  - Output of one program becomes input of another

  ```
  $ firstProgram | secondProgram
  ```

  - Output of **`firstProgram`** goes to **`secondProgram`**

Redirecting Input/Output on UNIX and Windows Systems (II)

- Redirect output (>)
  - Determines where output of a program goes
  - `$ myProgram > myFile`
    - Output goes into **`myFile`** (erases previous contents)

- Append output (>>)
  - Add output to end of file (preserve previous contents)
  - `$ myOtherProgram >> myFile`
    - Output goes to the end of **`myFile`**

Variable-Length Argument Lists

- Functions with unspecified number of arguments
  - Load **`<stdarg.h>`**
  - Use ellipsis (`...`) at end of parameter list
  - Need at least one defined parameter

```
double myfunction (int i, ...);
```

  - Prototype with variable length argument list
  - Example: prototype of **`printf`**

```
int printf( const char*format, ... );
```

Variable-Length Argument Lists (II)

- Macros and declarations in function definition

  `va_list`
  - Type specifier, required (**`va_list arguments`**`;`)

  `va_start(arguments,` *other variables*)
  - Intializes parameters, required before use

  `va_arg(arguments,` *type*)
  - Returns a parameter each time **`va_arg`** is called
  - Automatically points to next parameter

  `va_end(arguments)`
  - Helps function have a normal return

Variable-Length Argument Lists. Example

Outline

1. Load <stdarg.h> header
1.1 Function prototype (variable length argument list)
1.2 Initialize variables
2. Function calls
3. Function definition
3.1 Create ap (va_list object)
3.2 Initialize ap (va_start(ap, i))
3.3 Access arguments
va_arg(ap, double)
3.4 End function
va_end(ap);
return total/1;

Using Command-Line Arguments

- Pass arguments to **`main`** in Windows and UNIX

```
int main( int argc, char *argv[] )
```

`int argc` – number of arguments passed

`char *argv[]` – array of strings, has names of arguments in order (**`argv[ 0 ]`** is first argument)

Example: **`$ copy input output`**

```
argc: 3
argv[ 0 ]: "copy"
argv[ 1 ]: "input"
argv[ 2 ]: "output"
```

Outline

```c
/* Using command-line arguments */

#include <stdio.h>

int main( int argc, char *argv[] )
{
   FILE *inFilePtr, *outFilePtr;
   int c;

   if ( argc != 3 )
      printf( "Usage: copy infile outfile\n" )
   else
      if ( ( inFilePtr = fopen( argv[ 1 ], "

         if ( ( outFilePtr = fopen( argv[ 2 ], "w" ) ) !=
NULL )
            while ( ( c = fgetc( inFilePtr ) ) != EOF )
               fputc( c, outFilePtr );

         else
            printf( "File \"%s\" co
argv[ 2 ] );
      else
         printf( "File \"%s\" could not be opened\n", argv[ 1
] );

   return 0;
}
```

Notice **argc** and **argv[]** in **main**

argv[1] is the second argument, and is being read.

argv[2] is the third argument, and is being written to.

Loop until **End Of File**. **fgetc** a character from **inFilePtr** and **fputc** it into **outFilePtr**.

1. Initialize variables

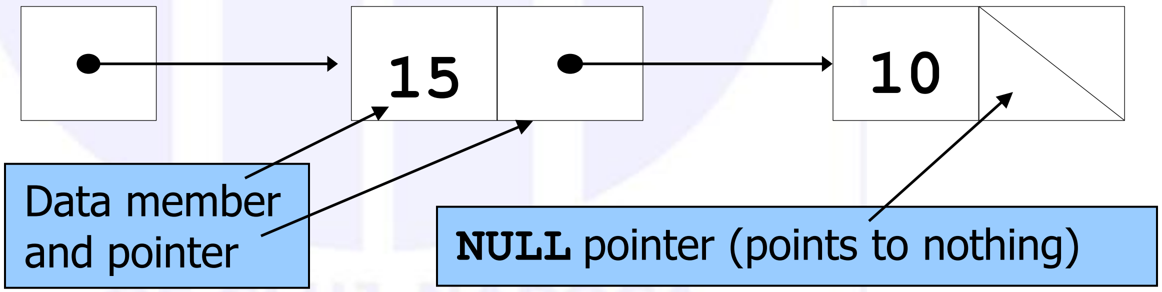
ction calls

open)

2.1 Specify open type (read or write)

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`**)
- Two self-referential structure objects linked together



-
  ```
  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`**

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
  - 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
  - Number of data elements is unpredictable
  - List needs to be sorted

Linked Lists (II)

- Types of linked lists:
  - *singly linked list*
    - Begins with a pointer to the first node
    - Terminates with a null pointer
    - Only traversed in one direction
  - *circular, singly linked*
    - Pointer in the last node points back to the first node
  - *doubly linked list*
    - Two “start pointers”- first element and last element
    - Each node has a forward pointer and a backward pointer
    - Allows traversals both forwards and backwards
  - *circular, doubly linked list*
    - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

```c
/* Operating and maintaining a list */

#include <stdio.h>
#include <stdlib.h>

struct listNode {   /* self-referential structure
*/   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insert( ListNodePtr *, char );
char delete( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );

int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

   instructions();  /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );
```

1. Define struct

1.1 Function prototypes

1.2 Initialize variables

2. Input choice

```c

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insert( &startPtr, item );
            printList( startPtr );
            break;
         case 2:
            if ( !isEmpty( startPtr ) ) {
               printf( "Enter character to be deleted: " );
               scanf( "\n%c", &item );

               if ( delete( &startPtr, item ) ) {
                  printf( "%c deleted.\n", item );
                  printList( startPtr );
               }
               else
                  printf( "%c not found.\n\n", item );
            }
            else
               printf( "List is empty.\n\n" );

            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

/* Print the instructions */
void instructions( void )
{
   printf( "Enter your choice:\n"
          "   1 to insert an element into the list.\n"
          "   2 to delete an element from the list.\n"
          "   3 to end.\n" );
}

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

   newPtr = malloc( sizeof( ListNode ) );

   if ( newPtr != NULL ) {     /* is space available */
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      previousPtr = NULL;
      currentPtr = *sPtr;
```

3. Function definitions

```c
while ( currentPtr != NULL && value > currentPtr->data ) {
   previousPtr = currentPtr;          /* walk to ...   */
   currentPtr = currentPtr->nextPtr;  /* ... next node */
}

if ( previousPtr == NULL ) {
   newPtr->nextPtr = *sPtr;
   *sPtr = newPtr;
      }
      else {
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else
      printf( "%c not inserted. No memory available.\n", value );
}

/* Delete a list element */
char delete( ListNodePtr *sPtr, char value )
{
   ListNodePtr previousPtr, currentPtr, tempPtr;

   if ( value == ( *sPtr )->data ) {
      tempPtr = *sPtr;
      *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
      free( tempPtr );             /* free the de-threaded node */
      return value;
   }
   else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      while ( currentPtr != NULL && currentPtr->data != value ) {
         previousPtr = currentPtr;          /* walk to ...   */
         currentPtr = currentPtr->nextPtr;  /* ... next node */
      }

      if ( currentPtr != NULL ) {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return value;
      }
   }

   return '\0';
}

/* Return 1 if the list is empty, 0 otherwise */
int isEmpty( ListNodePtr sPtr )
{
   return sPtr == NULL;
}

/* Print the list */
void printList( ListNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "List is empty.\n\n" );
   else {
      printf( "The list is:\n" );

   while ( currentPtr != NULL ) {
      printf( "%c --> ", currentPtr->data );
      currentPtr = currentPtr->nextPtr;
   }

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

```
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

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```

Stacks

- Stack
  - New nodes can be added and removed only at the top
  - Similar to a pile of dishes
  - Last-in, first-out (LIFO)
  - Bottom of stack indicated by a link member to **null**
  - Constrained version of a linked list
- push
  - Adds a new node to the top of the stack
- pop
  - Removes a node from the top
  - Stores the popped value
  - Returns **true** if **pop** was successful

```c
/* dynamic stack program */

#include <stdio.h>
#include <stdlib.h>

struct stackNode {   /* self-referential structure */
   int data;
   struct stackNode *nextPtr;
};

typedef struct stackNode StackNode;
typedef StackNode *StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );

int main()
{
   StackNodePtr stackPtr = NULL;  /* points to stack top */
   int choice, value;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );

```

1. Define struct

1.1 Function prototypes

1.1 Initialize variables

2. Input choice

```c
   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:      /* push value onto stack */
            printf( "Enter an integer: " );
            scanf( "%d", &value );
            push( &stackPtr, value );
            printStack( stackPtr );
            break;
         case 2:      /* pop value off stack */
            if ( !isEmpty( stackPtr ) )
               printf( "The popped value is %d.\n",
                      pop( &stackPtr ) );

            printStack( stackPtr );
            break;
         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}
```

```c
/* Print the instructions */
void instructions( void )
{
   printf( "Enter choice:\n"
          "1 to push a value on the stack\n"
          "2 to pop a value off the stack\n"
          "3 to end program\n" );
}

/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr;

   newPtr = malloc( sizeof( StackNode ) );
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else
      printf( "%d not inserted. No memory available.\n",
              info );
}

```

3. Function definitions

```c
/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr;
   int popValue;

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );
   return popValue;
}

/* Print the stack */
void printStack( StackNodePtr currentPtr )
{
         if ( currentPtr == NULL )
            printf( "The stack is empty.\n\n" );
         else {
            printf( "The stack is:\n" );

            while ( currentPtr != NULL ) {
               printf( "%d --> ", currentPtr->data );
               currentPtr = currentPtr->nextPtr;
            }

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

```

3. Function definitions

```c
/* Is the stack empty? */
int isEmpty( StackNodePtr topPtr )
{
   return topPtr == NULL;
}
```

3. Function definitions

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

Program Output

Program Output

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Queues

- Queue
  - Similar to a supermarket checkout line
  - *First-in, first-out (FIFO)*
  - Nodes are removed only from the *head*
  - Nodes are inserted only at the *tail*
- Insert and remove operations
  - Enqueue (insert) and dequeue (remove)
- Useful in computing
  - Print spooling, packets in networks, file server requests

```c
/* Operating and maintaining a queue */


#include <stdio.h>
#include <stdlib.h>

struct queueNode {   /* self-referential structure */
   char data;
   struct queueNode *nextPtr;
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

/* function prototypes */
void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
char dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, char );
void instructions( void );

int main()
{
   QueueNodePtr headPtr = NULL, tailPtr = NULL;
   int choice;
   char item;

   instructions();
   printf( "? " );
   scanf( "%d", &choice );
```

1. Define struct

1.1 Function prototypes

1.1 Initialize variables

2. Input choice

```c

   while ( choice != 3 ) {

      switch( choice ) {

         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            enqueue( &headPtr, &tailPtr, item );
            printQueue( headPtr );
            break;
         case 2:
            if ( !isEmpty( headPtr ) ) {
               item = dequeue( &headPtr, &tailPtr );
               printf( "%c has been dequeued.\n", item );
            }

            printQueue( headPtr );
            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
      }

      printf( "? " );
      scanf( "%d", &choice );
   }

   printf( "End of run.\n" );
   return 0;
}

```

```c
void instructions( void )
{
   printf ( "Enter your choice:\n"
           "   1 to add an item to the queue\n"
           "   2 to remove an item from the queue\n"
           "   3 to end\n" );
}

void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value )
{
   QueueNodePtr newPtr;

   newPtr = malloc( sizeof( QueueNode ) );

   if ( newPtr != NULL ) {
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      if ( isEmpty( *headPtr ) )
         *headPtr = newPtr;
      else
         ( *tailPtr )->nextPtr = newPtr;

      *tailPtr = newPtr;
   }
   else
      printf( "%c not inserted. No memory available.\n",
              value );
}

```

3. Function definitions

```c
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
{
   char value;
   QueueNodePtr tempPtr;

         value = ( *headPtr )->data;
         tempPtr = *headPtr;
         *headPtr = ( *headPtr )->nextPtr;

         if ( *headPtr == NULL )
            *tailPtr = NULL;

         free( tempPtr );
         return value;
      }

      int isEmpty( QueueNodePtr headPtr )
      {
         return headPtr == NULL;
      }

      void printQueue( QueueNodePtr currentPtr )
      {
         if ( currentPtr == NULL )
            printf( "Queue is empty.\n\n" );
         else {
            printf( "The queue is:\n" );
```

3. Function definitions

```c

         while ( currentPtr != NULL ) {
            printf( "%c --> ", currentPtr->data );
            currentPtr = currentPtr->nextPtr;
         }

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

3. Function definitions

Program Output

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 1
Enter a character: A
The queue is:
A --> NULL

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

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

```
? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Program Output

Reading

- Deitel: chapter 12, chapter 14
- Prata: chapter 17
- King: chapter 17

Summary

- Working with time
- I/O redirection
- Variable length argument lists
- Command line arguments
- Self referential structures
  - Lists
  - Stacks
  - Queues