雨课堂
Rain Classroom

本次课程是

线上+线下 融合式教学

请**现场**的同学们：

1. 打开雨课堂，点击页面右下角喇叭按钮调至静音状态

请**远程上课**的同学们：

1. 打开雨课堂，点击页面右下角喇叭按钮调至静音状态
2. 打开“腾讯会议”（会议室：824 8461 5333），进入会议室，并关闭麦克风

CHAPTER 10: STRUCTURES AND MACROS

Chapter Overview

- **Structures**
- Macros
- Conditional-Assembly Directives
- Defining Repeat Blocks

Structures - Overview

- Defining Structures
- Declaring Structure Variables
- Referencing Structure Variables
- Example: Displaying the System Time
- Nested Structures
- Example: Drunkard's Walk
- Declaring and Using Unions

Structure

- A template or pattern given to a logically related group of variables.
- field - structure member containing data
- Program access to a structure:
  - entire structure as a complete unit
  - individual fields
- Useful way to pass multiple related arguments to a procedure
  - example: file directory information

Using a Structure

Using a structure involve ? sequential steps:

1. Define the structure.
2. Declare one or more variables of the structure type, called structure variables.
3. Write runtime instructions that access the structure.

Structure Definition Syntax

```
name STRUCT
    field-declarations
name ENDS
```

- Field-declarations are identical to variable declarations

COORD Structure

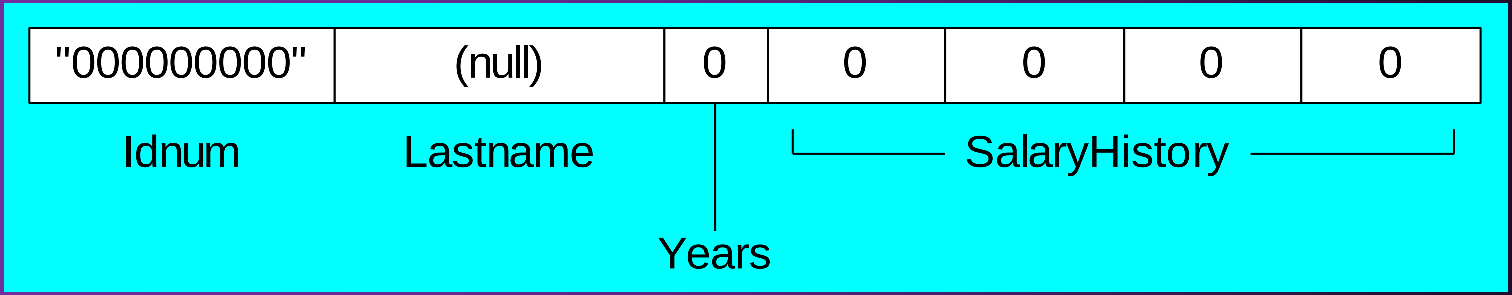
- The COORD structure used by the MS-Windows programming library identifies X and Y screen coordinates

```
COORD STRUCT
    X WORD ?            ; offset 00
    Y WORD ?            ; offset 02
COORD ENDS
```

Employee Structure

A structure is ideal for combining fields of different types:

```
Employee STRUCT
   IdNum BYTE "000000000"
   LastName BYTE 30 DUP(0)
   Years WORD 0
   SalaryHistory DWORD 0,0,0,0
Employee ENDS
```



Declaring Structure Variables

- Structure name is a user-defined type
- Insert replacement initializers between brackets:

  < . . . >

- Empty brackets <> retain the structure's default field initializers
- Examples:

```
.data
point1 COORD <5,10>
point2 COORD <>
worker Employee <>
```

Initializing Array Fields

- Use the DUP operator to initialize one or more elements of an array field:

```
.data
emp Employee <,,,2 DUP(20000)>
```

Array of Structures

- An array of structure objects can be defined using the DUP operator.
- Initializers can be used

```
NumPoints = 3
AllPoints COORD NumPoints DUP(<0,0>)

RD_Dept Employee 20 DUP(<>)

accounting Employee 10 DUP(<,,,4 DUP(20000) >)
```

Referencing Structure Variables

```
Employee STRUCT                               ; bytes
    IdNum BYTE "000000000"                    ; 9
    LastName BYTE 30 DUP(0)                   ; 30
    Years WORD 0                              ; 2
    SalaryHistory DWORD 0,0,0,0               ; 16
Employee ENDS                                 ; 57
```

```
.data
worker Employee <>

mov eax,TYPE Employee                         ; 57
mov eax,SIZEOF Employee                       ; 57
mov eax,SIZEOF worker                         ; 57
mov eax,TYPE Employee.SalaryHistory           ; 4
mov eax,LENGTHOF Employee.SalaryHistory       ; 4
mov eax,SIZEOF Employee.SalaryHistory         ; 16
```

. . . continued

```
mov dx,worker.Years
mov worker.SalaryHistory,20000         ; first salary
mov worker.SalaryHistory+4,30000       ; second salary
mov edx,OFFSET worker.LastName

mov esi,OFFSET worker
mov ax,(Employee PTR [esi]).Years

mov ax,[esi].Years    ; invalid operand (ambiguous)
```

Nested Structures (1 of 2)

- Define a structure that contains other structures.
- Used nested braces (or brackets) to initialize each COORD structure.

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

```
Rectangle STRUCT
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS

.data
rect1 Rectangle { {10,10}, {50,20} }
rect2 Rectangle < <10,10>, <50,20> >
```

Nested Structures (2 of 2)

- Use the dot (.) qualifier to access nested fields.
- Use indirect addressing to access the overall structure or one of its fields

```
mov rect1.UpperLeft.X, 10
mov esi,OFFSET rect1
mov (Rectangle PTR [esi]).UpperLeft.Y, 10

// use the OFFSET operator
mov edi,OFFSET rect2.LowerRight
mov (COORD PTR [edi]).X, 50
mov edi,OFFSET rect2.LowerRight.X
mov WORD PTR [edi], 50
```

Declaring and Using Unions

- A union is similar to a structure in that it contains multiple fields
- All of the fields in a union begin at the same offset
  - (differs from a structure)
- Provides alternate ways to access the same data
- Syntax:

```
unionname UNION
    union-fields
unionname ENDS
```

Integer Union Example

The Integer union consumes 4 bytes (equal to the largest field)

```
Integer UNION
   D DWORD 0
   W WORD 0
   B BYTE 0
Integer ENDS
```

D, W, and B are often called variant fields.

Integer can be used to define data:

```
.data
val1 Integer <12345678h>
val2 Integer <100h>
val3 Integer <>
```

What's Next

- Structures
- **Macros**
- Conditional-Assembly Directives
- Defining Repeat Blocks

Macros

- Introducing Macros
- Defining Macros
- Invoking Macros
- Macro Examples
- Nested Macros
- Example Program: Wrappers

Introducing Macros

- A macro[1] is a named block of assembly language statements.
- Once defined, it can be invoked (called) one or more times.
- During the assembler's preprocessing step, each macro call is expanded into a copy of the macro.
- The expanded code is passed to the assembly step, where it is checked for correctness.

[1]Also called a macro procedure.

Defining Macros

- A macro must be defined before it can be used.
- Parameters are optional.
- Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Syntax:

```
macroname MACRO [parameter-1, parameter-2,...]
    statement-list
ENDM
```

mNewLine Macro Example

This is how you define and invoke a simple macro.

```
mNewLine MACRO                      ; define the macro
    call Crlf
ENDM
.data

.code
mNewLine                            ; invoke the macro
```

The assembler will substitute "call crlf" for "mNewLine".

mPutChar Macro

Writes a single character to standard output.

Definition:

```
mPutchar MACRO char
    push eax
    mov al,char
    call WriteChar
    pop eax
ENDM
```

Invocation:

```
.code
mPutchar 'A'
```

Expansion:

```
1       push eax
1       mov al,'A'
1       call WriteChar
1       pop eax
```

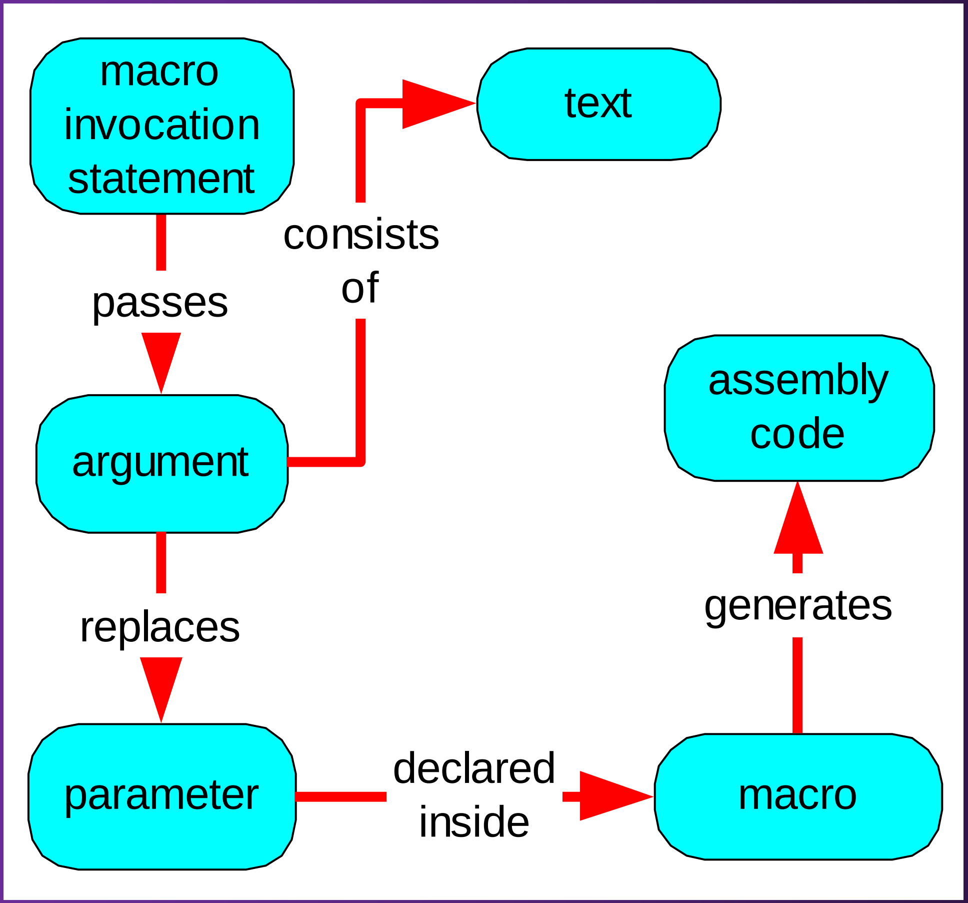
viewed in the listing file

Invoking Macros (1 of 2)

- When you invoke a macro, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code.
- Arguments are treated as simple text by the preprocessor.

Invoking Macros (2 of 2)

Relationships between macros, arguments, and parameters:



mWriteStr Macro (1 of 2)

Provides a convenient way to display a string, by passing the string name as an argument.

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
.data
str1 BYTE "Welcome!",0
.code
mWriteStr str1
```

mWriteStr Macro (2 of 2)

The expanded code shows how the str1 argument replaced the parameter named buffer:

```
mWriteStr MACRO buffer
     push edx
     mov  edx,OFFSET buffer
     call WriteString
     pop  edx
ENDM
```

```
1    push edx
1    mov  edx,OFFSET str1
1    call WriteString
1    pop  edx
```

Invalid Argument

- If you pass an invalid argument, the error is caught when the expanded code is assembled.
- Example:

```
.code
mPutchar 1234h
```

```
1      push eax
1      mov al,1234h                ; error!
1      call WriteChar
1      pop eax
```

Blank Argument

- If you pass a blank argument, the error is also caught when the expanded code is assembled.
- Example:

```
.code
mPutchar
```

```
1       push eax
1       mov al,
1       call WriteChar
1       pop eax
```

What's Next

- Structures
- Macros
- **Conditional-Assembly Directives**
- Defining Repeat Blocks

Conditional-Assembly Directives

- Checking for Missing Arguments
- Default Argument Initializers
- Boolean Expressions
- IF, ELSE, and ENDIF Directives
- The IFIDN and IFIDNI Directives
- Special Operators
- Macro Functions

Checking for Missing Arguments

- The IFB directive returns true if its argument is blank. For example:

```
IFB <row>                ;; if row is blank,
  EXITM                  ;; exit the macro
ENDIF
```

mWriteString Example

Display a message during assembly if the string parameter is empty:

```
mWriteStr MACRO string
    IFB <string>
        ECHO ---------------------------------------
        ECHO * Error: parameter missing in mWriteStr
        ECHO * (no code generated)
        ECHO ---------------------------------------
        EXITM
    ENDIF
    push edx
    mov edx,OFFSET string
    call WriteString
    pop edx
ENDM
```

Default Argument Initializers

- A default argument initializer automatically assigns a value to a parameter when a macro argument is left blank. For example, mWriteln can be invoked either with or without a string argument:

```
mWriteLn MACRO text:=<" ">
   mWrite text
   call Crlf
ENDM
.code
mWriteln "Line one"
mWriteln
mWriteln "Line three"
```

Sample output:

```
Line one

Line three
```

Boolean Expressions

A boolean expression can be formed using the following operators:

- LT - Less than
- GT - Greater than
- EQ - Equal to
- NE - Not equal to
- LE - Less than or equal to
- GE - Greater than or equal to

Only assembly-time constants may be compared using these operators.

IF, ELSE, and ENDIF Directives

A block of statements is assembled if the boolean expression evaluates to true. An alternate block of statements can be assembled if the expression is false.

```
IF boolean-expression
      statements
[ELSE
      statements]
ENDIF
```

Simple Example

The following IF directive permits two MOV instructions to be assembled if a constant named RealMode is equal to 1:

```
IF RealMode EQ 1
  mov ax,@data
  mov ds,ax
ENDIF
```

RealMode can be defined in the source code any of the following ways:

```
RealMode = 1
RealMode EQU 1
RealMode TEXTEQU 1
```

The IFIDN and IFIDNI Directives

- IFIDN compares two symbols and returns true if they are equal (case-sensitive)
- IFIDNI also compares two symbols, using a case-insensitive comparison
- Syntax:

```
IFIDNI <symbol>, <symbol>
      statements
ENDIF
```

Can be used to prevent the caller of a macro from passing an argument that would conflict with register usage inside the macro.

IFIDNI Example

Prevents the user from passing EDX as the second argument to the mReadBuf macro:

```
mReadBuf MACRO bufferPtr, maxChars
  IFIDNI <maxChars>,<EDX>
    ECHO Warning: Second argument cannot be EDX
    ECHO **************************************
    EXITM
  ENDIF
  .
  .
ENDM
```

Special Operators

- The substitution (&) operator resolves ambiguous references to parameter names within a macro.
- The expansion operator (%) expands text macros or converts constant expressions into their text representations.
- The literal-text operator (<>) groups one or more characters and symbols into a single text literal. It prevents the preprocessor from interpreting members of the list as separate arguments.
- The literal-character operator (!) forces the preprocessor to treat a predefined operator as an ordinary character.

替換操作符 Substitution (&)

Text passed as regName is substituted into the literal string definition:

```
ShowRegister MACRO regName
.data
tempStr BYTE "  &regName=",0
.
.
.code
ShowRegister EDX             ; invoke the macro
```

Macro expansion:

```
tempStr BYTE "  EDX=",0
```

展开操作符 Expansion (%)

Forces the evaluation of an integer expression. After the expression has been evaluated, its value is passed as a macro argument:

```
mGotoXY %(5 * 10),%(3 + 4)

The preprocessor generates the following code:

1 push edx
1 mov dl,50
1 mov dh,7
1 call Gotoxy
1 pop edx
```

文本操作符 Literal-Text (<>)

The first macro call passes three arguments. The second call passes a single argument:

```
mWrite "Line three", 0dh, 0ah

mWrite <"Line three", 0dh, 0ah>
```

特殊文本字符操作符 Literal-Character (!)

The following declaration prematurely ends the text definition when the first > character is reached.

```
BadYValue TEXTEQU Warning: <Y-coordinate is > 24>
```

The following declaration continues the text definition until the final > character is reached.

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Macro Functions (1 of 2)

- A macro function returns an integer or string constant
- The value is returned by the EXITM directive
- Example: The IsDefined macro acts as a wrapper for the IFDEF directive.

```
IsDefined MACRO symbol
    IFDEF symbol
       EXITM <-1>                    ;; True
    ELSE
       EXITM <0>                     ;; False
    ENDIF
ENDM
```

Notice how the assembler defines True and False.

Macro Functions (2 of 2)

- When calling a macro function, the argument(s) must be enclosed in parentheses
- The following code permits the two MOV statements to be assembled only if the RealMode symbol has been defined:

```
IF IsDefined( RealMode )
    mov ax,@data
    mov ds,ax
ENDIF
```

What's Next

- Structures
- Macros
- Conditional-Assembly Directives
- **Defining Repeat Blocks**

Defining Repeat Blocks

- WHILE Directive
- REPEAT Directive
- FOR Directive
- FORC Directive
- Example: Linked List

WHILE Directive

- The WHILE directive repeats a statement block as long as a particular constant expression is true.
- Syntax:

```
WHILE constExpression
    statements
ENDM
```

WHILE Example

Generates Fibonacci integers between 1 and F0000000h at assembly time:

```
.data
val1 = 1
val2 = 1
DWORD val1                          ; first two values
DWORD val2
val3 = val1 + val2

WHILE val3 LT 0F0000000h
    DWORD val3
    val1 = val2
    val2 = val3
    val3 = val1 + val2
ENDM
```

REPEAT Directive

- The REPEAT directive repeats a statement block a fixed number of times.
- Syntax:

```
REPEAT constExpression
    statements
ENDM
```

ConstExpression, an unsigned constant integer expression, determines the number of repetitions.

REPEAT Example

The following code generates 100 integer data definitions in the sequence 10, 20, 30, . . .

```
iVal = 10
REPEAT 100
    DWORD iVal
    iVal = iVal + 10
ENDM
```

FOR Directive

- The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols.
- Each symbol in the list causes one iteration of the loop.
- Syntax:

```
FOR parameter,<arg1,arg2,arg3,...>
    statements
ENDM
```

FOR Example

The following Window structure contains frame, title bar, background, and foreground colors. The field definitions are created using a FOR directive:

```
Window STRUCT
  FOR color,<frame,titlebar,background,foreground>
     color DWORD ?
  ENDM
Window ENDS
```

Generated code:

```
Window STRUCT
    frame DWORD ?
    titlebar DWORD ?
    background DWORD ?
    foreground DWORD ?
Window ENDS
```

FORC Directive

- The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string causes one iteration of the loop.
- Syntax:

```
FORC parameter, <string>
    statements
ENDM
```

FORC Example

Suppose we need to accumulate seven sets of integer data for an experiment. Their label names are to be Group_A, Group_B, Group_C, and so on. The FORC directive creates the variables:

```
FORC code,<ABCDEFG>
    Group_&code WORD ?
ENDM
```

Generated code:

```
  Group_A WORD ?
  Group_B WORD ?
  Group_C WORD ?
  Group_D WORD ?
  Group_E WORD ?
  Group_F WORD ?
  Group_G WORD ?
```

CHAPTER 11: MS-WINDOWS PROGRAMMING

Chapter Overview

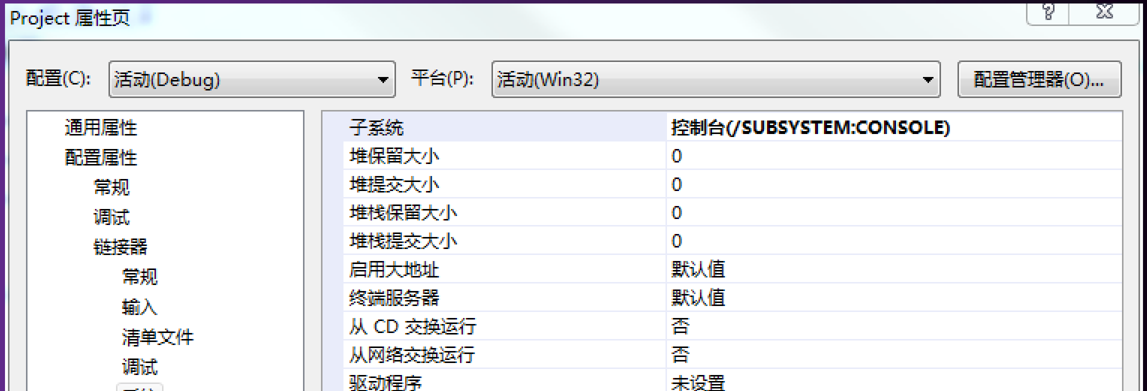
- **Win32 Console Programming**
- Writing a Graphical Windows Application
- Dynamic Memory Allocation
- x86 Memory Management

Win32 Console Programming

- Background Information
  - Win32 Console Programs
  - API and SDK
  - Windows Data Types
  - Standard Console Handles
- Console Input
- Console Output
- Reading and Writing Files
- Console Window Manipulation
- Controlling the Cursor
- Controlling the Text Color
- Time and Date Functions

Win32 Console Programs

- Run in Protected mode
- Emulate MS-DOS
- Standard text-based input and output
- Linker option : /SUBSYSTEM:CONSOLE
- The console input buffer contains a queue of input records, each containing data about an input event.
- A console screen buffer is a two-dimensional array of character and color data that affects the appearance of text in the console window.



配置(C): 活动(Debug)　平台(P): 活动(Win32)

子系统	控制台(/SUBSYSTEM:CONSOLE)
堆保留大小	0
堆提交大小	0
堆栈保留大小	0
堆栈提交大小	0
启用大地址	默认值
终端服务器	默认值
从 CD 交换运行	否
从网络交换运行	否
驱动程序	未设置

Classifying Console Functions

- Text-oriented (high-level) console functions
  - Read character streams from input buffer
  - Write character streams to screen buffer
  - Redirect input and output
- Event-oriented (low-level) console functions
  - Retrieve keyboard and mouse events
  - Detect user interactions with the console window
  - Control window size & position, text colors

Translating Windows Data Types

Windows Type(s)	MASM Type
BOOL	DWORD
LONG	SDWORD
COLORREF, HANDLE, LPARAM, LPCTSTR, LPTSTR, LPVOID, LRESULT, UINT, WNDPROC, WPARAM	DWORD
BSTR, LPCSTR, LPSTR	PTR BYTE
WORD	WORD
LPCRECT	PTR RECT

Standard Console Handles

A handle is an unsigned 32-bit integer. The following MS-Windows constants are predefined to specify the type of handle requested:

- STD_INPUT_HANDLE
  - standard input
- STD_OUTPUT_HANDLE
  - standard output
- STD_ERROR_HANDLE
  - standard error output

GetStdHandle

- GetStdHandle returns a handle to a console stream
- Specify the type of handle (see previous slide)
- The handle is returned in EAX
- Prototype:

```
GetStdHandle PROTO,
  nStdHandle:DWORD           ; handle type
```

- Sample call:

```
INVOKE GetStdHandle, STD_OUTPUT_HANDLE
mov myHandle, eax
```

Console Input

- The ReadConsole function provides a convenient way to read text input and put it in a buffer.
- Prototype:

```
ReadConsole PROTO,
    handle:DWORD,                ; input handle
    pBuffer:PTR BYTE,            ; pointer to buffer
    maxBytes:DWORD,              ; number of chars to read
    pBytesRead:PTR DWORD,        ; ptr to num bytes read
    notUsed:DWORD                ;  (not used)
```

Single-Character Input

Here's how to input single characters:

- Get a copy of the current console flags by calling GetConsoleMode. Save the flags in a variable.
- Change the console flags by calling SetConsoleMode.
- Input a character by calling ReadConsole.
- Restore the previous values of the console flags by calling SetConsoleMode.

COORD and SMALL_RECT

- The COORD structure specifies X and Y screen coordinates in character measurements, which default to 0-79 and 0-24.
- The SMALL_RECT structure specifies a window's location in character measurements.

```
COORD STRUCT
    X WORD ?
    Y WORD ?
COORD ENDS
```

```
SMALL_RECT STRUCT
  Left   WORD ?
  Top    WORD ?
  Right  WORD ?
  Bottom WORD ?
SMALL_RECT ENDS
```

WriteConsole

- The WriteConsole function writes a string to the screen, using the console output handle. It acts upon standard ASCII control characters such as tab, carriage return, and line feed.
- Prototype:

```
WriteConsole PROTO,
    handle:DWORD,                   ; output handle
    pBuffer:PTR BYTE,               ; pointer to buffer
    bufsize:DWORD,                  ; size of buffer
    pCount:PTR DWORD,               ; output count
    lpReserved:DWORD                ; (not used)
```

Example: Console1.asm

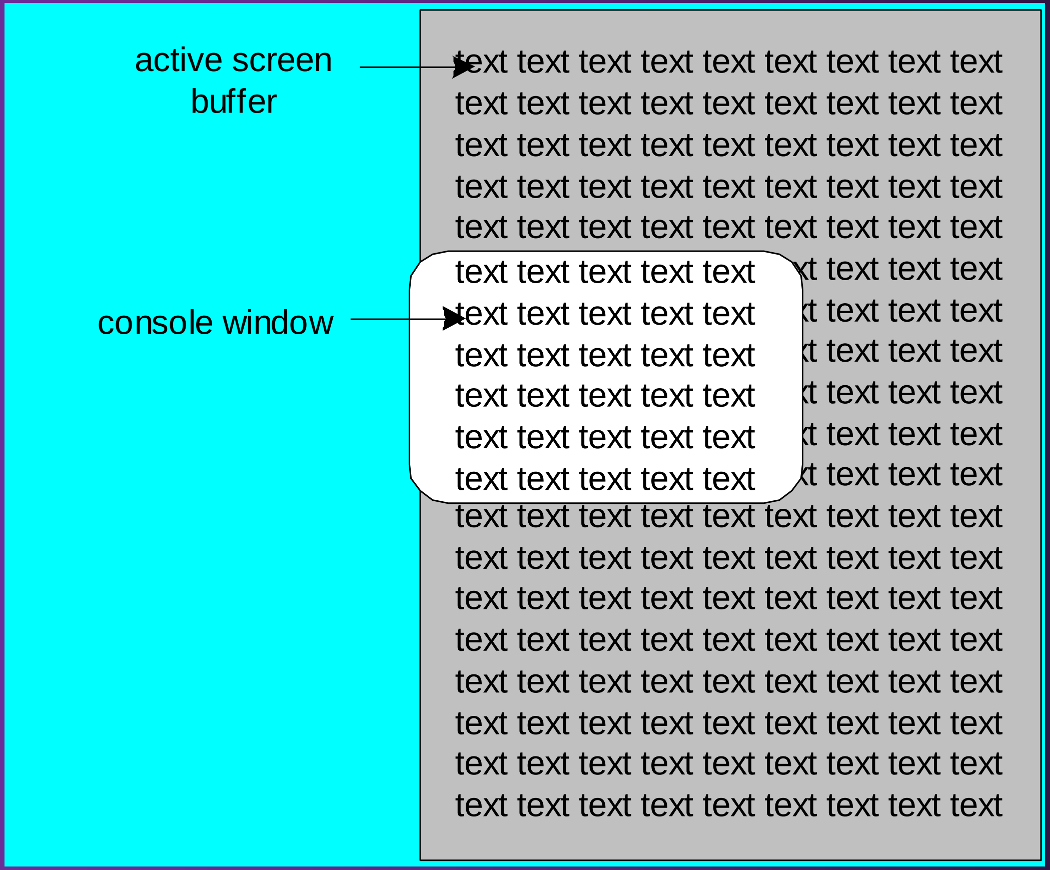
```
 main c1 PROC
; Get the console output handle:
      INVOKE GetStdHandle, STD_OUTPUT_HANDLE
      mov consoleHandle,eax
      mov ebx, messageSize

  ; Write a string to the console:
      INVOKE WriteConsole,
        consoleHandle,          ; console output handle
        ADDR message,           ; string pointer
        ebx,                    ; string length
        ADDR bytesWritten,      ; ret num bytes written
        0                       ; not used

      INVOKE ExitProcess,0
mainc1 ENDP
```

Console Window Manipulation

- Screen buffer
- Console window
- Controlling the cursor
- Controlling the text color

Screen Buffer and Console Window

- The active screen buffer (屏幕缓冲区) includes data displayed by the console window (控制台窗口).



SetConsoleTitle

SetConsoleTitle changes the console window's title. Pass it a null-terminated string:

```
.data
titleStr BYTE "Console title",0
.code
INVOKE SetConsoleTitle, ADDR titleStr
```

GetConsoleScreenBufferInfo

GetConsoleScreenBufferInfo returns information about the current state of the console window. It has two parameters: a handle to the console screen, and a pointer to a structure that is filled in by the function:

```
.data
outHandle DWORD ?
consoleInfo CONSOLE_SCREEN_BUFFER_INFO <>
.code
   INVOKE GetConsoleScreenBufferInfo,
       outHandle,
       ADDR consoleInfo
```

CONSOLE_SCREEN_BUFFER_INFO

```
CONSOLE_SCREEN_BUFFER_INFO STRUCT
  dwSize                COORD <>
  dwCursorPos           COORD <>
  wAttributes           WORD ?
  srWindow              SMALL_RECT <>
  maxWinSize            COORD <>
CONSOLE_SCREEN_BUFFER_INFO ENDS
```

- dwSize - size of the screen buffer (char columns and rows)
- dwCursorPos - cursor location
- wAttributes - colors of characters in console buffer
- srWindow - coords of console window relative to screen buffer
- maxWinSize - maximum size of the console window

SetConsoleWindowInfo

- SetConsoleWindowInfo lets you set the size and position of the console window relative to its screen buffer.
    - If bAbsolute is true, the coordinates specify the new upper left and lower right corners of the console window.
    - If bAbsolute is false, the coordinates will be added to the current window coordinates.
- Prototype:

```
SetConsoleWindowInfo PROTO,
   nStdHandle:DWORD,                ; screen buffer handle
   bAbsolute:DWORD,                 ; coordinate type.绝/相对
   pConsoleRect:PTR SMALL_RECT      ; window rectangle
```

SetConsoleScreenBufferSize

- SetConsoleScreenBufferSize lets you set the screen buffer size to X columns by Y rows.
- Prototype:

```
SetConsoleScreenBufferSize PROTO,
  outHandle:DWORD,           ; handle to screen buffer
  dwSize:COORD               ; new screen buffer size
```

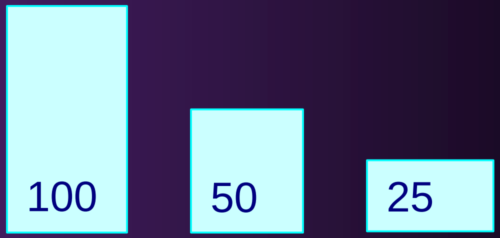
Controlling the Cursor

- GetConsoleCursorInfo
  - returns the size and visibility of the console cursor
- SetConsoleCursorInfo
  - sets the size and visibility of the cursor
- SetConsoleCursorPosition
  - sets the X, Y position of the cursor

CONSOLE_CURSOR_INFO

- Structure containing information about the console's cursor size and visibility
  - dwSize: Percentage 1 to 100 of the character cell
  - bVisible: TRUE(1) or FALSE(0)

```
CONSOLE_CURSOR_INFO STRUCT
  dwSize    DWORD ?
  bVisible DWORD ?
CONSOLE_CURSOR_INFO ENDS
```



SetConsoleTextAttribute

- Sets the foreground and background colors of all subsequent text written to the console.
- Prototype:

```
SetConsoleTextAttribute PROTO,
   hConsoleOutput:HANDLE,  ; console output handle
   wAttributes:WORD        ; color attribute
```

WriteConsoleOutputAttribute

- Copies an array of attribute values to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputAttribute PROTO,
    hConsoleOutput:DWORD,        ; output handle
    lpAttribute:PTR WORD,        ; write attributes颜色属性数组
    nLength:DWORD,               ; number of cells 颜色属性数量
    dwWriteCoord:COORD,          ; first cell coordinates
    lpNumberOfAttrsWritten:PTR DWORD     ; output count
```

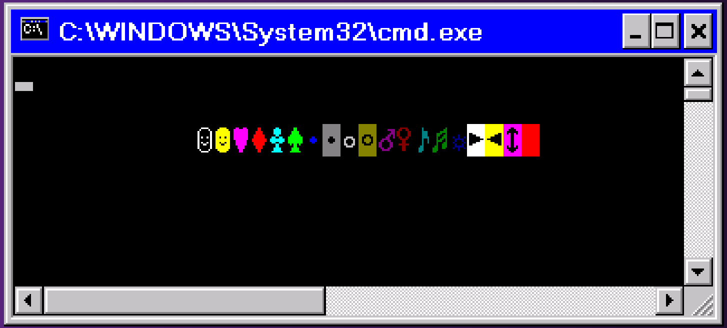
WriteConsoleOutputCharacter

- The WriteConsoleOutputCharacter function copies an array of characters to consecutive cells of the console screen buffer, beginning at a specified location.
- Prototype:

```
WriteConsoleOutputCharacter PROTO,
  hConsoleOutput:HANDLE,      ; console output handle
  lpCharacter:PTR BYTE,       ; pointer to buffer
  nLength:DWORD,              ; size of buffer
  dwWriteCoord:COORD,         ; first cell coordinates
  lpNumberOfCharsWritten:PTR DWORD   ; output count
```

WriteColors Program

- Creates an array of characters and an array of attributes, one for each character
- Copies the attributes to the screen buffer
- Copies the characters to the same screen buffer cells as the attributes
- Sample output:



(starts in row 2, column 10)

View the source code

File Manipulation

- Win32 API Functions that create, read, and write to files:
  - CreateFile
  - ReadFile
  - WriteFile
  - SetFilePointer

CreateFile

- CreateFile either creates a new file or opens an existing file. If successful, it returns a handle to the open file; otherwise, it returns a special constant named INVALID_HANDLE_VALUE.
- Prototype:

```
CreateFile PROTO,
  pFilename:PTR BYTE,            ; ptr to filename
  desiredAccess:DWORD,           ; access mode
  shareMode:DWORD,               ; share mode
  lpSecurity:DWORD,              ; ptr to security attribs
  creationDisposition:DWORD,     ; file creation options
  flagsAndAttributes:DWORD,      ; file attributes
  htemplate:DWORD                ; handle to template file
```

CreateFile Examples (1 of 3)

Opens an existing file for reading:

```
INVOKE CreateFile,
    ADDR filename,              ; ptr to filename
    GENERIC_READ,               ; access mode
    DO_NOT_SHARE,               ; share mode
    NULL,                       ; ptr to security attributes
    OPEN_EXISTING,              ; file creation options
    FILE_ATTRIBUTE_NORMAL,      ; file attributes
    0                           ; handle to template file
```

CreateFile Examples (2 of 3)

Opens an existing file for writing:

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,               ; access mode
    DO_NOT_SHARE,
    NULL,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,
    0
```

CreateFile Examples (3 of 3)

Creates a new file with normal attributes, erasing any existing file by the same name:

```
INVOKE CreateFile,
    ADDR filename,
    GENERIC_WRITE,
    DO_NOT_SHARE,
    NULL,
    CREATE_ALWAYS,          ; overwrite existing file
    FILE_ATTRIBUTE_NORMAL,
    0
```

ReadFile

- ReadFile reads text from an input file
- Prototype:

```
ReadFile PROTO,
    handle:DWORD,                   ; handle to file
    pBuffer:PTR BYTE,               ; ptr to buffer
    nBufsize:DWORD,                 ; num bytes to read
    pBytesRead:PTR DWORD,           ; bytes actually read
    pOverlapped:PTR DWORD           ; ptr to asynch info
```

WriteFile

- WriteFile writes data to a file, using an output handle. The handle can be the screen buffer handle, or it can be one assigned to a text file.
- Prototype:

```
WriteFile PROTO,
    fileHandle:DWORD,                ; output handle
    pBuffer:PTR BYTE,                ; pointer to buffer
    nBufsize:DWORD,                  ; size of buffer
    pBytesWritten:PTR DWORD,         ; num bytes written
    pOverlapped:PTR DWORD            ; ptr to asynch info
```

SetFilePointer

SetFilePointer moves the position pointer of an open file. You can use it to append data to a file, and to perform random-access record processing:

```
SetFilePointer PROTO,
  handle:DWORD,                  ; file handle
  nDistanceLo:SDWORD,            ; bytes to move pointer
  pDistanceHi:PTR SDWORD,        ; ptr to bytes to move
  moveMethod:DWORD               ; starting point
```

Example:

```
; Move to end of file:

INVOKE SetFilePointer,
  fileHandle,0,0,FILE_END
```

Time and Date Functions

- GetLocalTime, SetLocalTime
- GetTickCount, Sleep
- GetDateTime
- SYSTEMTIME Structure
- Creating a Stopwatch Timer

GetLocalTime, SetLocalTime

- GetLocalTime returns the date and current time of day, according to the system clock.
- SetLocalTime sets the system’s local date and time.

```
GetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
```

```
SetLocalTime PROTO,
  pSystemTime:PTR SYSTEMTIME
```

SYSTEMTIME Structure

- SYSTEMTIME is used by date and time-related Windows API functions:

```
SYSTEMTIME STRUCT
    wYear WORD ?                 ; year (4 digits)
    wMonth WORD ?                ; month (1-12)
    wDayOfWeek WORD ?            ; day of week (0-6)
    wDay WORD ?                  ; day (1-31)
    wHour WORD ?                 ; hours (0-23)
    wMinute WORD ?               ; minutes (0-59)
    wSecond WORD ?               ; seconds (0-59)
    wMilliseconds WORD ?         ; milliseconds (0-999)
SYSTEMTIME ENDS
```

GetTickCount, Sleep

- GetTickCount function returns the number of milliseconds that have elapsed since the system was started.
- Sleep pauses the current program for a specified number of milliseconds.

```
GetTickCount PROTO     ; return value in EAX
```

```
Sleep PROTO,
   dwMilliseconds:DWORD
```

What's Next

- Win32 Console Programming
- **Writing a Graphical Windows Application**
- Dynamic Memory Allocation
- x86 Memory Management

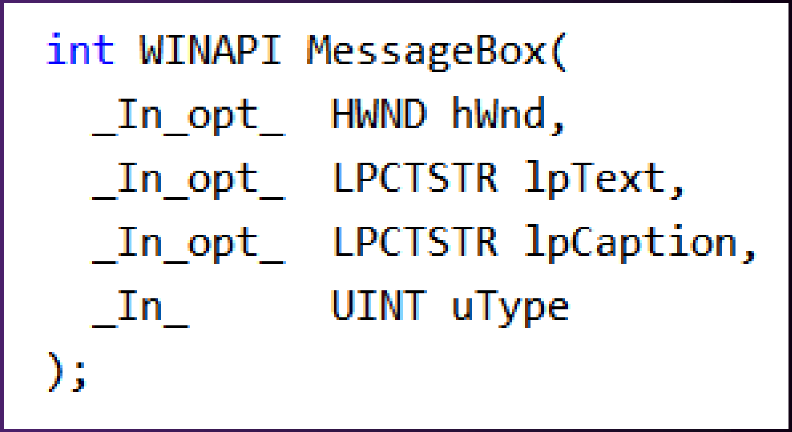
Writing a Graphical Windows Application

- Required Files
- POINT, RECT Structures
- MSGStruct, WNDCLASS Structures
- MessageBox Function
- WinMain, WinProc Procedures
- ErrorHandler Procedure
- Message Loop & Processing Messages
- Program Listing

MessageBox Function

Displays text in a box that pops up and waits for the user to click on a button:

```
MessageBox PROTO,
    hWnd:DWORD,
    lpText:PTR BYTE,
    lpCaption:PTR BYTE,
    uType:DWORD
```



```
int WINAPI MessageBox(
  _In_opt_  HWND hWnd,
  _In_opt_  LPCTSTR lpText,
  _In_opt_  LPCTSTR lpCaption,
  _In_      UINT uType
);
```

hWnd is a handle to the current window.
lpText points to a null-terminated string that will appear inside the box.
lpCaption points to a null-terminated string that will appear in the box's caption bar.
uTyle is an integer that describes both the dialog box's icon (optional) and the buttons (required).

MessageBox Example

Displays a message box that shows a question, including an OK button and a question-mark icon:

```
.data
hMainWnd      DWORD ?
QuestionText  BYTE "Register this program now?",0
QuestionTitle BYTE "Trial Period Has Expired",0

.code
INVOKE MessageBox,
    hMainWnd,
    ADDR QuestionText,
    ADDR QuestionTitle,
    MB_OK + MB_ICONQUESTION
```

Required Files

- make32.bat - Batch file specifically for building this program
- WinApp.asm - Program source code
- GraphWin.inc - Include file containing structures, constants, and function prototypes used by the program
- kernel32.lib - Same MS-Windows API library used earlier in this chapter
- user32.lib - Additional MS-Windows API functions

When linking the program, use /SUBSYSTEM:WINDOWS
Not /SUBSYSTEM:CONSOLE

POINT and RECT Structures

- POINT - X, Y screen coordinates
- RECT - Holds the graphical coordinates of two opposing corners of a rectangle

```
POINT STRUCT
  ptX  DWORD ?
  ptY  DWORD ?
POINT ENDS
```

```
RECT STRUCT
  left      DWORD ?
  top       DWORD ?
  right     DWORD ?
  bottom    DWORD ?
RECT ENDS
```

MSGStruct Structure

MSGStruct - holds data for MS-Windows messages (usually passed by the system and received by your application):

```
MSGStruct STRUCT
   msgWnd                  DWORD ?
   msgMessage              DWORD ?
   msgWparam               DWORD ?
   msgLparam               DWORD ?
   msgTime                 DWORD ?
   msgPt                   POINT <>
MSGStruct ENDS
```

WNDCLASS Structure (1 of 2)

Each window in a program belongs to a class, and each program defines a window class for its main window:

```
WNDCLASS STRUC
  style         DWORD ?     ; window style options
  lpfnWndProc   DWORD ?     ; WinProc function pointer
  cbClsExtra    DWORD ?     ; shared memory
  cbWndExtra    DWORD ?     ; number of extra bytes
  hInstance     DWORD ?     ; handle to current program
  hIcon         DWORD ?     ; handle to icon
  hCursor       DWORD ?     ; handle to cursor
  hbrBackground DWORD ?     ; handle to background brush
  lpszMenuName  DWORD ?     ; pointer to menu name
  lpszClassName DWORD ?     ; pointer to WinClass name
WNDCLASS ENDS
```

WNDCLASS Structure (2 of 2)

- style is a conglomerate (组合) of different style options, such as WS_CAPTION and WS_BORDER, that control the window’s appearance and behavior.
- lpfnWndProc is a pointer to a function (in our program) that receives and processes event messages triggered by the user.
- cbClsExtra refers to shared memory used by all windows belonging to the class. Can be null.
- cbWndExtra specifies the number of extra bytes to allocate following the window instance.
- hInstance holds a handle to the current program instance.
- hIcon and hCursor hold handles to icon and cursor resources for the current program.
- hbrBackground holds a background (color) brush.
- lpszMenuName points to a menu string.
- lpszClassName points to a null-terminated string containing the window’s class name.

WinMain Procedure

Every Windows application needs a startup procedure, usually named WinMain, which is responsible for the following tasks:

- Get a handle to the current program
- Load the program's icon and mouse cursor
- Register the program's main window class and identify the procedure that will process event messages for the window
- Create the main window
- Show and update the main window
- Begin a loop that receives and dispatches messages

WinProc Procedure

- WinProc receives and processes all event messages relating to a window
  - Some events are initiated by clicking and dragging the mouse, pressing keyboard keys, and so on
- WinProc decodes each message, carries out application-oriented tasks related to the message

```
WinProc PROC,
    hWnd:DWORD,          ; handle to the window
    localMsg:DWORD,      ; message ID
    wParam:DWORD,        ; parameter 1 (varies)
    lParam:DWORD         ; parameter 2 (varies)
```

(Contents of wParam and lParam vary, depending on the message.)

Sample WinProc Messages

- In the example program from this chapter, the WinProc procedure handles three specific messages:
  - WM_LBUTTONDOWN, generated when the user presses the left mouse button
  - WM_CREATE, indicates that the main window was just created
  - WM_CLOSE, indicates that the application's main window is about to close

(many other messages are possible)

ErrorHandler Procedure

- The ErrorHandler procedure has several important tasks to perform:
  - Call GetLastError to retrieve the system error number
  - Call FormatMessage to retrieve the appropriate system-formatted error message string
  - Call MessageBox to display a popup message box containing the error message string
  - Call LocalFree to free the memory used by the error message string

$\longrightarrow$

(sample)

ErrorHandler Sample

```
INVOKE GetLastError          ; Returns message ID in EAX
mov messageID,eax

; Get the corresponding message string.
INVOKE FormatMessage, FORMAT_MESSAGE_ALLOCATE_BUFFER + \
  FORMAT_MESSAGE_FROM_SYSTEM, NULL, messageID, NULL,
  ADDR pErrorMsg, NULL, NULL

; Display the error message.
INVOKE MessageBox, NULL, pErrorMsg, ADDR ErrorTitle,
  MB_ICONERROR + MB_OK

; Free the error message string.
INVOKE LocalFree, pErrorMsg
```

Message Loop

In WinMain, the message loop receives and dispatches (relays) messages:

```
Message_Loop:
    ; Get next message from the queue.
    INVOKE GetMessage, ADDR msg, NULL,NULL,NULL

    ; Quit if no more messages.
    .IF eax == 0
      jmp Exit_Program
    .ENDIF

    ; Relay the message to the program's WinProc.
    INVOKE DispatchMessage, ADDR msg

    jmp Message_Loop
```

Processing Messages

WinProc receives each message and decides what to do with it:

```
WinProc PROC, hWnd:DWORD, localMsg:DWORD,
    wParam:DWORD, lParam:DWORD
    mov eax, localMsg
    .IF eax == WM_LBUTTONDOWN       ; mouse button?
      INVOKE MessageBox, hWnd, ADDR PopupText,
             ADDR PopupTitle, MB_OK
      jmp WinProcExit
    .ELSEIF eax == WM_CREATE        ; create window?
      INVOKE MessageBox, hWnd, ADDR AppLoadMsgText,
             ADDR AppLoadMsgTitle, MB_OK
      jmp WinProcExit
    (etc.)
```

- (WinApp.asm)

Program Listing

- View the program listing (WinApp.asm)
- Run the program

When linking the program, remember to replace

/SUBSYSTEM:CONSOLE

with: /SUBSYSTEM:WINDOWS

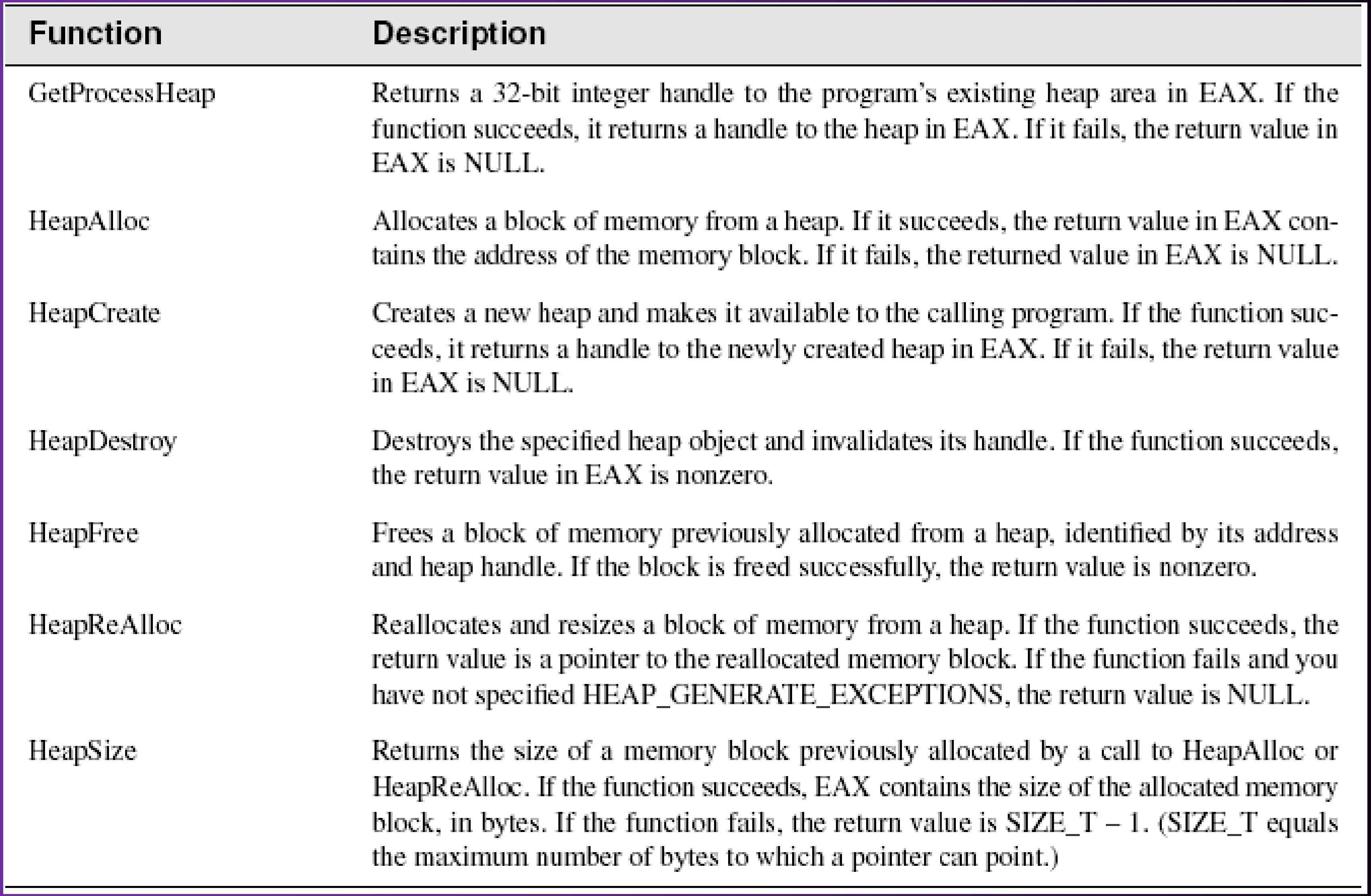
What's Next

- Win32 Console Programming
- Writing a Graphical Windows Application
- **Dynamic Memory Allocation**
- IA-32 Memory Management

Dynamic Memory Allocation

- Reserving memory at runtime for objects
  - aka *heap allocation*
  - standard in high-level languages (C++, Java)
- Heap manager
  - allocates large blocks of memory
  - maintains free list of pointers to smaller blocks
  - manages requests by programs for storage

Windows Heap-Related Functions



Function	Description
GetProcessHeap	Returns a 32-bit integer handle to the program's existing heap area in EAX. If the function succeeds, it returns a handle to the heap in EAX. If it fails, the return value in EAX is NULL.
HeapAlloc	Allocates a block of memory from a heap. If it succeeds, the return value in EAX contains the address of the memory block. If it fails, the returned value in EAX is NULL.
HeapCreate	Creates a new heap and makes it available to the calling program. If the function succeeds, it returns a handle to the newly created heap in EAX. If it fails, the return value in EAX is NULL.
HeapDestroy	Destroys the specified heap object and invalidates its handle. If the function succeeds, the return value in EAX is nonzero.
HeapFree	Frees a block of memory previously allocated from a heap, identified by its address and heap handle. If the block is freed successfully, the return value is nonzero.
HeapReAlloc	Reallocates and resizes a block of memory from a heap. If the function succeeds, the return value is a pointer to the reallocated memory block. If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.
HeapSize	Returns the size of a memory block previously allocated by a call to HeapAlloc or HeapReAlloc. If the function succeeds, EAX contains the size of the allocated memory block, in bytes. If the function fails, the return value is SIZE_T – 1. (SIZE_T equals the maximum number of bytes to which a pointer can point.)

Sample Code

- Get a handle to the program's existing heap or Create a handle to a private heap.

```
HEAP_START = 2000000           ;   2 MB
HEAP_MAX = 400000000           ; 400 MB
.data
hHeap HANDLE ?

.code
INVOKE HeapCreate, 0, HEAP_START, HEAP_MAX
.IF eax == NULL                ; cannot get handle
   jmp quit
.ELSE
   mov hHeap,eax               ; handle is OK
.ENDIF
```

Sample Code

- Allocate block of memory from existing heap:

```
.data
hHeap HANDLE ?        ; heap handle
pArray DWORD ?        ; pointer to array

.code
INVOKE HeapAlloc, hHeap, HEAP_ZERO_MEMORY, 1000
.IF eax == NULL
  mWrite "HeapAlloc failed"
  jmp quit
.ELSE
  mov pArray,eax
.ENDIF
```

Sample Code

- Free a block of memory previously created by calling HeapAlloc:

```
.data
hHeap HANDLE ?                ; heap handle
pArray DWORD ?                ; pointer to array

.code
INVOKE HeapFree,
   hHeap,                     ; handle to heap
   0,                         ; flags
   pArray                     ; pointer to array
```

CHAPTER 14:
16-BIT MS-DOS PROGRAMMING

Chapter Overview

- **MS-DOS and the IBM-PC**
- MS-DOS Function Calls (INT 21h)
- Standard MS-DOS File I/O Services

MS-DOS and the IBM-PC

- Real-Address Mode
- MS-DOS Memory Organization
- MS-DOS Memory Map
- Redirecting Input-Output
- Software Interrupts
- INT Instruction
- Interrupt Vectoring Process
- Common Interrupts

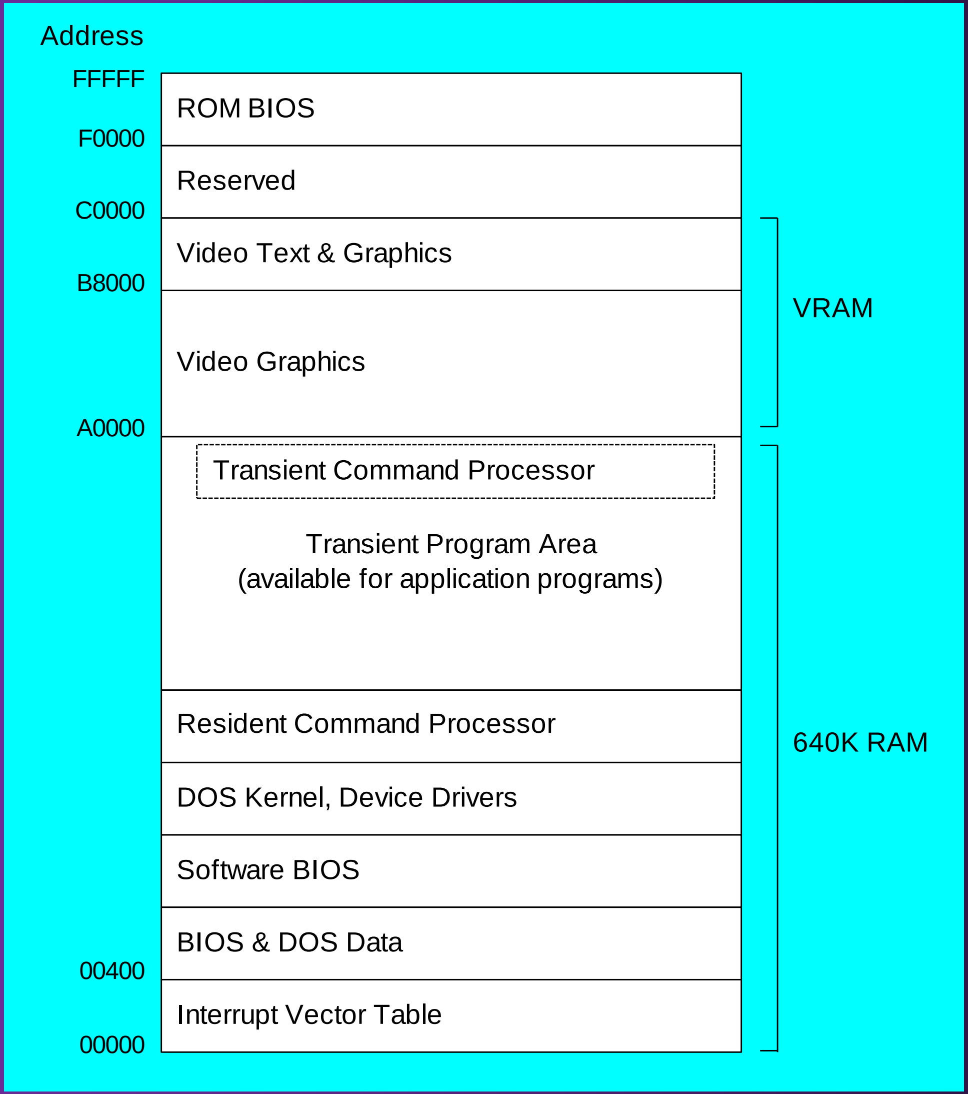
Real-Address Mode

- Real-address mode (16-bit mode) programs have the following characteristics:
  - Max 1 megabyte addressable RAM
  - Offsets are 16 bits
  - No memory boundary protection
  - Single tasking
- IBM PC-DOS: first Real-address OS for IBM-PC
  - Has roots in Gary Kildall's highly successful Digital Research CP/M
  - Later renamed to MS-DOS, owned by Microsoft

MS-DOS Memory Organization

- Interrupt Vector Table
- BIOS & DOS data
- Software BIOS
- MS-DOS kernel
- Resident command processor
- Transient programs
- Video graphics & text
- Reserved (device controllers)
- ROM BIOS

MS-DOS Memory Map



Summary (part of Chap 10)

- Use a structure to define complex types
  - contains fields of different types
- Macro – named block of statements
  - substituted by the assembler preprocessor
- Conditional assembly directives
  - IF, IFNB, IFIDNI, ...
- Operators: &, %, <>, !
- Repeat block directives (assembly time)
  - WHILE, REPEAT, FOR, FORC

Summary (Chap 11)

- 32-bit console programs
  - read from the keyboard and write plain text to the console window using Win32 API functions
- Important functions
  - ReadConsole, WriteConsole, GetStdHandle, ReadFile, WriteFile, CreateFile, CloseHandle, SetFilePointer
- Dynamic memory allocation
  - HeapAlloc, HeapFree
- x86 Memory management
  - segment selectors, linear address, physical address
  - segment descriptor tables
  - paging, page directory, page tables, page translation

Summary (Chap 14)

- MS-DOS applications
  - 16-bit segments, segmented addressing, running in real-address mode
  - complete access to memory and hardware

Homework

- Reading Chap 10, 11, 14
- Exercises

Thanks!