Responding to a Keypress - 1

- To respond to a WM_CHAR message:
  - Add the ON_WM_CHAR message macro to the message map,
  - Add the message handler prototype OnChar to your window class:

```
afx_msg void OnChar(UINT ch, UINT Count, UINT flags)
```

  - Implement the message handler

Responding to a Keypress - 2

- Every time the user presses a key, a WM_CHAR message is generated, and OnChar is called.
- The ascii value of the key pressed is passed in ch.
- Your application can then process this information in any way you please.

Responding to Mouse Press

- Mouse messages are handled similarly:
  - Add WM_LBUTTONDOWN macro.
  - Add OnLButtonDown() handler prototype to your window class.

```
afx_msg void OnLButtonDown(UINT flags, Cpoint loc);
```

  - Implement handler (right button identical).
  - loc.x and loc.y hold the cursor position.

Device Contexts

- A device context is an output path from your Windows application, through the appropriate driver, to the window.
- Before your application can output information onto the screen, a dc must be obtained.
- Routines which output text or graphics require a dc to be declared, e.g. TextOut().
- A device context is obtained by creating an object of type CClientDC.

Example - 1

```
//This is the main window class
class CMainWin : public CFrameWnd
{
char charEntered;
public:
CMainWin();
afx_msg void OnChar(UINT Ch,UINT Count,UINT flags);
afx_msg void OnLButtonDown(UINT flags,CPoint loc);
DECLARE_MESSAGE_MAP()
};
```

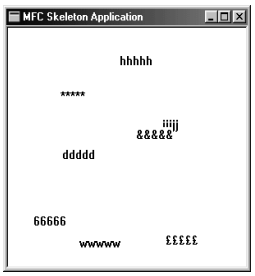
Example - 2

```
//Construct a window
CMainWin::CMainWin()
{
Create(NULL, "MFC Skeleton Application");
charEntered = '*';
}

// This is the application's message map.
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
      ON_WM_CHAR()
      ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()
```

```
//Process a keyhit
afx_msg void CMainWin::OnChar(UINT ch,UINT Count,
                                    UINT Flags)
{
charEntered = ch;
}

//Process a mouse message
afx_msg void CMainWin::OnLButtonDown(UINT flags,
                                    CPoint loc)
{
      CClientDC dc(this);
      char str[5];
      for(int i=0;i<5;i++)
            str[i]=charEntered;
      dc.TextOut(loc.x,loc.y,str,sizeof(str));
}
```

Output From Example.

- This program will output a string of 5 characters whenever the left button is pressed.
- Whenever a character is hit, it will make that the output character.



Problem.

- However, whenever the window is:
  - Minimised, then restored
  - Resized
  - Covered by another window then redisplayed
- ..everything disappears.
  - This is because Windows doesn't keep a record of what a window contains.

The WM_PAINT Message

- When such an event occurs, the window is said to be invalidated.
- Upon invalidation of a window, Windows sends a WM_PAINT message to your program.
  - This also happens the first time your window is displayed.
- When your program receives this message, it must redisplay the contents of the window.

Handling Output to the Window

- It is good practice to perform all your output in response to a WM_PAINT message.
  - This allows the Windows O.S. to better manage the system.
- I.e. when your program has information to output, it requests that a WM_PAINT message be sent when Windows is ready to do so.

Generating a WM_PAINT Message

- We know that a paint message is generated when the window has been invalidated (e.g. resized, minimised etc.)
- We can force a repaint of the window by manually invalidating the contents ourselves.
- **InvalidateRect()** does this, either to the whole window, or to a specified rectangular region of the window.

Changes needed - 1:

```
//This is the main window class
class CMainWin : public CFrameWnd
{
      char charEntered;
      CPoint pos;
public:
      CMainWin();
      afx_msg void OnChar(UINT Ch, UINT Count,
                                    UINT Flags);
      afx_msg void OnLButtonDown(UINT flags,
                                    CPoint loc);
      afx_msg void OnPaint();
      DECLARE_MESSAGE_MAP()
};
```

Changes needed – 2

```
// This is the application's message map.
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
      ON_WM_CHAR()
      ON_WM_LBUTTONDOWN()
      ON_WM_PAINT()
END_MESSAGE_MAP()
```

Changes needed – 3

```
afx_msg void CMainWin::OnLButtonDown(UINT flags,
                                    CPoint loc)
{
      pos.x = loc.x;
      pos.y = loc.y;

      InvalidateRect(NULL);

}
```

Changes needed – 4

```
afx_msg void CMainWin::OnPaint()
{
CPaintDC dc(this);

char str[5];
for(int i=0;i<5;i++)
     str[i]=charEntered;
dc.TextOut(pos.x,pos.y,str,sizeof(str));
}
```

CPaintDC

- Instead of a CClientDC, we obtain a device context by creating a CPaintDC object.
- This has the same effect, but also ensures that the contents of the window are subsequently validated
  - Otherwise it would constantly refresh the contents, causing a noticeable flicker.
- This context should only be used in response to a WM_PAINT message.

PAINTSTRUCT

- CPaintDC includes an important data member: m_ps, which contains a structure of type PAINTSTRUCT
- This structure contains information about the current state of the device context (I.e. window)
- The most important member of this structure is the rcPaint member, of type RECT.

PAINTSTRUCT

- A structure of type RECT contains 4 integer values containing the left, top, right and bottom screen co-ordinates of a rectangular region.
- Therefore, if we want to get the current dimensions of the window (e.g. after resizing), we can do something like:

```
int width = dc.m_ps.rcPaint.right;
int height = dc.m_ps.rcPaint.bottom;
```

Message Boxes

- Create a message box using the MessageBox() method, a member of CWnd.
- Sometimes, you may want to display a message box before your main window has been created.
  - In this case use the MFC global function AfxMessageBox().

Messagebox() – Example of Usage

```
afx_msg void CMainWin::OnRButtonDown(UINT flags,
                                     CPoint loc){
   int i = MessageBox("Press One", "Right Button",
       MB_ABORTRETRYIGNORE|MB_ICONQUESTION);
   switch (i){
   case IDABORT:
       MessageBox("","Abort");
       break;
   case IDRETRY:
       MessageBox("","Retry");
       break;
   case IDIGNORE:
       MessageBox("","Ignore");
   }
}
```

A right button click displays:

A button click displays:

Common values for MBType

MB_ABORTRETRYIGNORE
The message box contains three push buttons: Abort, Retry, and Ignore.
MB_OK
The message box contains one push button: OK. This is the default.
MB_OKCANCEL
The message box contains two push buttons: OK and Cancel.
MB_RETRYCANCEL
The message box contains two push buttons: Retry and Cancel.
MB_YESNO
The message box contains two push buttons: Yes and No.
MB_YESNOCANCEL
The message box contains three push buttons: Yes, No, and Cancel.

MB_ICONEXCLAMATION, exclamation-point icon appears
MB_ICONQUESTION question-mark icon appears MB_ICONSTOP, Etc…..

Menus

- The most common element of control in Windows, and almost every main window has one.
  - Therefore, substantial inbuilt control provided.
  - Top level menu bar, pop-up submenus
- Adding a menu to a window involves:
  1. Define the form of the window in a resource file
  2. Load the menu when your program creates its main window
  3. Process menu selections.

Using Resources

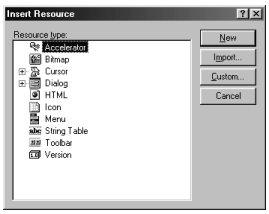
- Resources are objects that are used by your program, but defined **outside** your program.
- They include menus, icons, dialog boxes, and bitmapped graphics
- A resource is created separately from your program, but is added to the .EXE file when your program is linked.

Resource files

- Resources are contained in resource files, which have the extension .RC
- In general, they should have the same name as the executable
  - E.g. PROG.EXE, PROG.RC
- Some may be text-based (e.g. menu) or some are graphics-based (e.g. icon, cursor, bitmap).
- Most Integrated development environments (e.g. VC++ BC++) contain resource editors, and handle the compilation and linking of resource files.

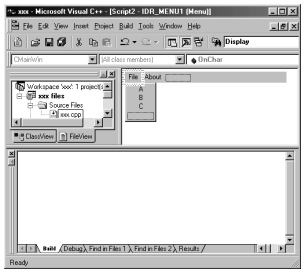
Creating a Menu Resource in VC+

- Insert Resource (from menu bar), and choose menu



Creating a Menu Resource in VC+

- Graphically Design your menu:



Creating a Menu Resource in VC+

- Save the file with the same name as your application, and the extension .rc
- Another file called resource.h will automatically have been created, containing:

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Menu1.rc
//
#define IDR_MENU1                       101
#define ID_FILE_A                       40001
#define ID_FILE_B                       40002
#define ID_FILE_C                       40003
#define ID_ABOUT_A                      40004
#define ID_ABOUT_D                      40005
#define ID_ABOUT_F                      40006

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40007
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Including a Menu in your Program

- Once you have defined a menu, you include it in your program by specifying its name when you create the window:

```
Create(NULL, "Using Menus",
       WS_OVERLAPPEDWINDOW,rectDefault,NULL,
       MAKEINTRESOURCE(IDR_MENU1));
```

- You also need to #include "resources.h" in your main .cpp file
- After this statement executes, the menu with id IDR_MENU1 will be displayed.
- However, the program is not yet ready to respond to it.

WM_Command messages

- Every time the user makes a menu selection, your program is sent a WM_COMMAND message.
- Associated with this message is the ID value of the item selected (as defined in resource.h)
- It is the ID value which determines which handler must be called.
- MFC provides a mechanism that automates this process.

Handling WM_Command messages

- MFC links each ID value with its own handler inside the message map.
- After this is done, each time a WM_COMMAND message is received, the handler associated with the current ID value is executed.
- You add these to your message map using the ON_COMMAND message macro:

ON_COMMAND(*ID,HandlerName*)

Responding to Menu Selections

- To allow your program to respond to menu selections:
  1. Add an ON_COMMAND message macro to your program's message map for each menu ID value (found in resources.h)
  2. Add the prototype for each command handler to your main window class
  3. Implement each command handler.

Example - *in menu.h*

```
//This is the main window class
class CMainWin : public CFrameWnd
{
      char charEntered;
public:
      CMainWin();
      afx_msg void OnA();
      afx_msg void OnB();
      afx_msg void OnC();
      afx_msg void OnD();
      afx_msg void OnE();
      afx_msg void OnF();
      DECLARE_MESSAGE_MAP()
};
```

Example - *in resource.h*

```
//{{NO_DEPENDENCIES}}
//Microsoft Developer Studio generated include file.
//Used by Menu1.rc
//
#define IDR_MENU1  101
#define ID_FILE_A  40001
#define ID_FILE_B  40002
#define ID_FILE_C  40003
#define ID_ABOUT_A 40004
#define ID_ABOUT_D 40005
#define ID_ABOUT_F 40006
```

Etc…

Example - *in menu.cpp*

```
// This is the application's message map.
BEGIN_MESSAGE_MAP(CMainWin, CFrameWnd)
      ON_COMMAND(ID_FILE_A, OnA)
      ON_COMMAND(ID_FILE_B, OnB)
      ON_COMMAND(ID_FILE_C, OnC)
      ON_COMMAND(ID_ABOUT_D, OnD)
      ON_COMMAND(ID_ABOUT_E, OnE)
      ON_COMMAND(ID_ABOUT_F, OnF)
END_MESSAGE_MAP()
```

Example handlers - *in menu.cpp*

```
afx_msg void CMainWin::OnA(){
      MessageBox("A","A");
}

afx_msg void CMainWin::OnB(){
      MessageBox("B","B");
}

afx_msg void CMainWin::OnC(){
      MessageBox("C","C");
}

afx_msg void CMainWin::OnD(){
      MessageBox("D","D");
}
Etc…
```

Working with Graphics

- Windows has a flexible set of graphics functions available.
- By default, the upper-left corner of a window is in the top left corner
  - The x value increases to the right, and the y value increases downwards.

The units for graphics-based functions are pixels.

Graphics position

- Windows maintains a current position that is used and/or updated by certain graphics functions, e.g. MoveTo(), LineTo()
- When your program begins, this is set to be the upper-left corner
- This graphics "cursor" is completely invisible.
- It is simply the next place in the window at which certain graphics functions will begin.

Pens and Brushes

- The windows graphics system is based on two important objects, pens and brushes.
  - Closed graphics shapes, such as rectangles and ellipses, are filled using the currently selected brush.
  - Pens are resources that draw lines and curves. The default pen is black and one pixel thick.

Graphics functions

- MFC encapsulates the Win32 API graphics functions within the CDC class.
- Some examples:
  - SetPixel(x,y,colour) sets the colour of a specific pixel
  - LineTo(x,y) draws a line from the current position to x,y using the currently selected pen.
  - MoveTo(x,y) sets the current position
- Other functions encapsulated by CDC include Arc(), Rectangle(), RoundRect(), Ellipse(), Pie() etc…

Working with Pens

- Graphics objects are drawn using the current pen and brush. (Similar to drawing/painting in real-life).
- In MFC, pens are encapsulated by the CPen class.
  - You can obtain a stock pen using e.g. CreateStockObject(WHITE_PEN), but these are quite limited
- You define your own pen by declaring a Cpen object and then calling

```
CreatePen(style, width, colour)
```

Pen Styles

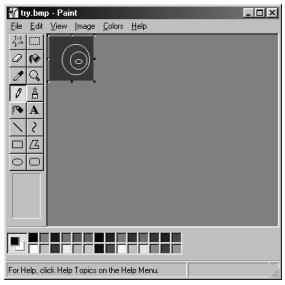
- The Style parameter determines what type of pen is created, E.G:
  - PS_SOLID, PS_DASH, PS_DASH_DOT, PS_DASHDOTDOT, PS_DOT, PS_INSIDEFRAME
- The dotted and/or dashed styles may only be applied to pens that are one unit thick
- You may also use the CPen constructor to create a pen:

```
CPen(int Style, int Width, COLORREF Colour);
```

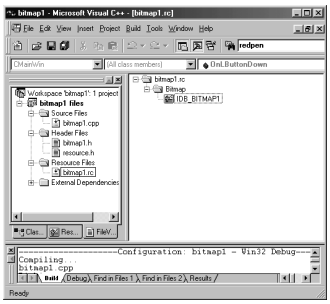
Bitmaps

- A bitmap is a graphics image, encapsulated by the CBitmap class.
- It contains a bit-by-bit representation of the image that will ultimately be displayed on the screen.
- You may create a bitmap using an image editor, and add it to your application as a resource.
- Your program may then load and display it.

1 - Generate the Bitmap



2- Add to project as a resource



3- Loading the bitmap

- To use a bitmap in your program, you must first create a `CBitmap` object.
- You then load the bitmap, using the member method `LoadBitmap()`.
- The following declares a bitmap object, and loads the bitmap with id `IDB_BITMAP1`
  - (Remember to `#include "resource.h"`)

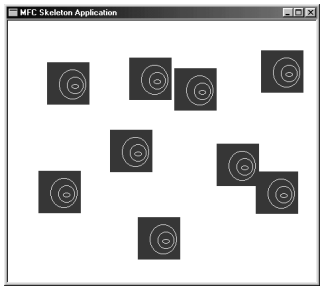
```
CBitmap bit;
bit.LoadBitmap(MAKEINTRESOURCE(IDB_BITMAP1);
```

4- Displaying the Bitmap

- The following steps are necessary to display the bitmap:
  1. Obtain the device context so that your program can output to the window
  2. Obtain an equivalent memory device context that will hold the bitmap until it is displayed. (A bitmap is held in memory until it is copied to your window).
  3. Select the bitmap into the memory device context.
  4. Copy the bitmap from the memory device context to the window device context. This causes the bitmap to actually be displayed.

Example - Displaying the Bitmap

```
afx_msg void CMainWin::OnLButtonDown(UINT flags,
                                     CPoint loc){
CClientDC dc(this);
CDC memDC;
//create a compatible memory DC
memDC.CreateCompatibleDC(&dc);
//Select bitmap into memory DC
memDC.SelectObject(&bit);
//Copy Bitmap to window DC
dc.BitBlt(loc.x,loc.y,64,64,&memDC,0,0,SRCCOPY);
}
```



Details

- Two device contexts are declared:
  - dc holds the current window device context
  - memDC will be used to hold the device context of the memory that holds the bitmap
- Next, a memory device context is created to hold the bitmap, which will be compatible with the device context, using `CreateCompatibleDC()`
  - This memory will be used to construct an image before it is displayed.

Details

- Before a bitmap can be displayed, it must be selected into the memory device context using `SelectObject()`
- Finally, the `BitBlt()` function is used to copy the selected object from the memory device context to the window device context.
  - You can specify how this is to be copied. E.G. SRCCOPY overwrites existing information. SRCAND blends the bitmap with the current background.

The Repaint problem

- Remember, when a window is re-displayed, the current contents are lost.
- It is your program's responsibility to restore the contents of the window upon receipt of a WM_PAINT message.
- There are 3 possible solutions:
  1. Regenerate the output by re-computing everything
  2. Store a record of display events, and "replay" them
  3. Maintain a copy of a ***virtual window***, and copy the contents each time a WM_PAINT message is received.

Virtual Window Theory

- Firstly, a memory device context is created that is compatible with the window DC.
- Then, all output is written to the memory device context.
- Each time a WM_PAINT message is received, the contents of the memory device context are copied into the physical DC.
- This causes output to the screen, and because all output has been written to the memory DC, there is always a record of the current contents of the physical window.

Displaying Graphics

- We use the virtual window method to display graphics.
- Firstly, an uninitialized device context is declared, and a bitmap which will contain the graphics.

```
CDC m_memDC;      //virtual window device context
CBitmap m_bmp;    //virtual window bitmap
```

Displaying Graphics

- Then, in the window constructor method, a device context is declared for the current window, and a compatible memory device context is created

```
CClientDC dc(this);
m_memDC.CreateCompatibleDC(&dc);
```

Displaying Graphics

- A bitmap is created with the same dimensions as the window, and is selected into the memory device context

```
maxX = GetSystemMetrics(SM_CXSCREEN);
maxY = GetSystemMetrics(SM_CYSCREEN);

m_bmp.CreateCompatibleBitmap(&dc,maxX,maxY);
m_memDC.SelectObject(&m_bmp);
```

Displaying Graphics

- All subsequent graphics are drawn to the memory device context:

```
m_memDC.Ellipse(x1,y1,x2,y2);
```

- And this is copied to the physical device context on reponse to a WM_PAINT message:

```
dc.BitBlt(0,0,maxX,maxY,&m_memDC,0,0,SRCCOPY);
```

Logical vs. Physical units

- Physical units are pixels on the screen.
- Up to now, we have defined all our graphics in terms of physical units.
- Under many circumstances, these units are too inflexible.
  - We may want to model objects using real-world units, such as feet and inches
  - If we define a model in pixels, every time we move closer or further away, the model needs to be re-calculated.

Viewing

- Viewing is the process of drawing a view of a model on a 2-dimensional display
- The geometric description of the object or scene provided by the model, is converted into a set of graphical primitives, which are displayed where desired on the 2D display.
- The same abstract model may be viewed in many different ways (e.g. faraway, near, looking down, looking up)

Display co-ordinates

- The integer, (x,y) screen co-ordinates are far too restrictive to be used to describe physical objects and scenes, because:
  - objects and scenes would need to be remodelled for every new view, or if the device resolution is changed
  - The range of coordinate values (e.g: 1000x700 pixels) is inappropriate for many applications

Real World Co-ordinates/Logical Units

- It is logical to use dimensions which are appropriate to the object. e.g.
  - metres for buildings,
  - millimetres for assembly parts,
  - nanometres or microns for molecules, cells, and atoms
- The objects are described with respect to their actual physical size in the real world, and then mapped onto screen co-ordinates
- It is therefore possible to view an object at various sizes by zooming in and out, without actually having to change the model

Viewing Issues

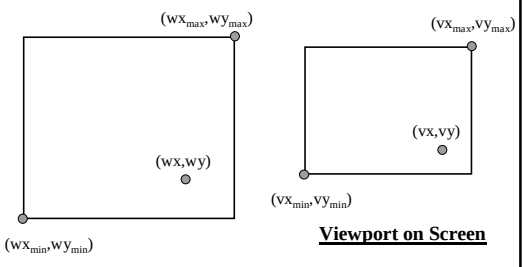
- How much of the model should be drawn?
- Where should it appear on the display?
- How do we convert Real-world co-ordinates into screen co-ordinates?

E.g: We could have a model of a whole room, full of objects such as chairs, tables, and students. We may want to view the whole room in one go, or zoom in on one single object in the room. We may want to display the object or scene on the full screen, or we may only want to display it on a portion of the screen

2-dimensional Views

- Once a model has been constructed, the programmer can specify a view
- A 2-Dimensional view consists of two rectangles:
  - a **Logical Window**, given in real-world co-ordinates, which defines the portion of the model that is to be drawn
  - a **Physical Viewport** given in screen co-ordinates, which defines the portion of the screen on which the contents of the window will be displayed

Windows and Viewports



Window on Real-world

Viewport on Screen

How is point (wx, wy) mapped onto (vx,vy) ?

Window to viewport mapping

Preservation of horizontal ratios implies that:

$$\frac{wx - wx_{min}}{wx_{max} - wx_{min}} = \frac{vx - vx_{min}}{vx_{max} - vx_{min}}$$

So, solving for vx:

$$vx = (wx - wx_{min}) * \frac{vx_{max} - vx_{min}}{wx_{max} - wx_{min}} + vx_{min}$$

Vy can be solved for similarly.

Mapping Modes in MFC

- In MFC, you can specify:
  - A mapping mode, which defines the translation from logical units to physical units
  - The length and width of a window in terms of the logical units that you select.
  - The physical extents of a viewport, i.e. a region within the window on-screen. Once a viewport has been selected, all output is confined within its boundaries.
- The mapping mode and viewport functions are encapsulated by the CDC class.

Setting the Mapping Mode

- **`CDC::SetMapMode(`*`Mode`*`)`** allows you to set the current mapping mode.
- The possible modes include:
  - MM_TEXT (default): maps each logical unit to one device pixel.
  - MM_LOMETRIC: maps each logical unit to 0.1 millimetre.
  - MM_ANISOTROPIC: maps to programmer-defined units with arbitrarily scaled axes.
  - MM_ISOTROPIC: maps to programmer-defined units with equally scaled axes, (this establishes a one-to-one aspect ratio).

Mapping Modes

- There are several reasons why you might want to change the current mapping mode:
  - If your program should display output in millimetres or inches, so you could select e.g. MM_LOMETRIC
  - to define units appropriate to your model.
  - to change the scale of what is displayed (e.g. shrink or enlarge)
  - to establish a one-to-one aspect ration between the X and Y axes, i.e. each X unit represents the same physical distance as each Y unit.

Defining the Window Extents

- Selecting either MM_ISOTROPIC or MM_ANISOTROPIC mapping mode allows you to define the window in terms of logical units.
  - In fact, you must.
- To define the X and Y extents of a window, use:

`CDC::SetWindowExt(`*`Xextent,Yextent`*`)`

  - *Xextent* and *Yextent* specify the new horizontal and vertical extents measured in logical units

Window Extents

- Remember, when you change the logical dimensions of a window, your are not changing the physical size of the window on-screen.
- You are simply defining the size fo the window in terms of logical units that you choose.
- More specifically, you are defining the relationship between the logical units used by the window and the physical units (pixels).

Defining a Viewport

- A viewport is a region within a window that received output
- You define it using:
  `CDC::SetViewPortExt(xExtent,Yextent)`
- Xextent and Yextent specify the new extents in pixels.
- A viewport may be any size you desire: e.g. the whole window, or just a part of it.
- Output is automatically mapped from the window device context (logical units) to the viewport (pixels), and scaled accordingly.

Setting the Viewport Origin

- By default, the origin of the viewport is at (0,0) within the physical window.
- You can change this by using:
  `CDC::SetViewPortOrg(X,Y)`
- ***X*** and ***Y***, specified in pixels, define the new origin of the viewport.
- Changing the origin of the viewport changes where images are drawn in the window.