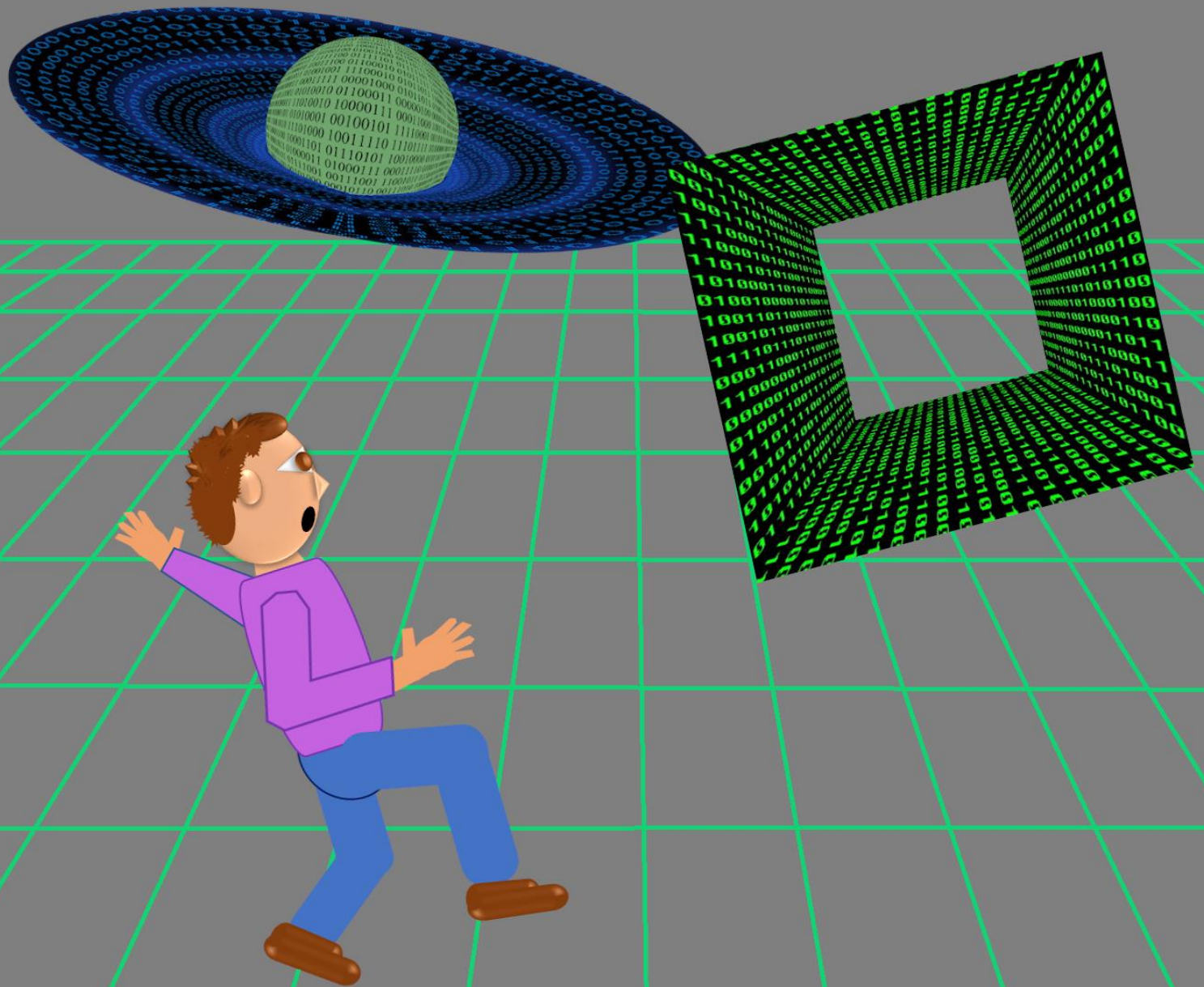


I SEE BYTES: A SIMPLIFIED C++ LIBRARY



İbrahim Cem Baykal
SECOND EDITION

ICBYTES: A SIMPLIFIED C++ LIBRARY

İBRAHİM CEM BAYKAL

2nd EDITION
October 2024

- Copyrights owned by İbrahim Cem Baykal.
- Turkish R. Culture & Turism Ministry Directorate of Copyrights Registration No: 2024/10359
- Digital copies can be distributed freely without altering.
- Printed copies can only be produced once for personal use but can not be sold/distributed.
- Cannot be quoted without citing the source.

INDEX

Preface	1
----------------	----------

Chapter 1: Design and Terminology	3
--	----------

1.1. OpenCV Code Examples	4
1.2. OpenCV Documentation	6
1.3. New Terminology	8
1.3.1. Fluid	8
1.3.2. Widget	8
1.4. Function Naming Rules	9

Chapter 2: Graphics Interface Sublibrary: ICGUI	11
--	-----------

2.1. Hello World	12
2.2. Width, Length, Color and Style	14
2.3. ICGUI and ICBYTES	16
2.4. Reading Text from the Screen (Input)	19
2.5. Sending Parameters to a Button Function	20
2.6. Check Box	23
2.7. Image Loading	23
2.8. List Box	26
2.9. Track Bar	28
2.10. Adding Menus	31

Chapter 3: Matrix Usage	35
--------------------------------	-----------

3.1 Matrix Methods	36
3.2 Matrix Operations	36

3.2.1 Assignment	36
3.2.2 Matrix Variable types	37
3.2.3 Addition and Subtraction	37
3.2.4 Multiplication and Division	38
3.2.5 Comparison	38
3.2.6 Matrix Inverse and Determinant	38
3.2.7 Transpose	38
3.2.8 Dot Product	38
3.2.9 Cross Product	38
3.3 Other Matrix Functions	38
3.4 Special Matrices	39
3.5 Error Diagnosis	40
3.5.1 How to start Troubleshooting?	42
3.6 Matrix Properties	45
3.6.1 Size Inquiry	45
3.7 Accessing Matrix of Unknown Type	46
3.8 Character Access	49

Chapter 4: Graphical Interface Styles **51**

4.1. Picture Button	51
4.2. Multiline Edit Window Types	54
4.3. Single Line Edit Window Types	55
4.4. Picture Frame Styles	56
4.5. Static Text Styles	57
4.6. Track Bar	58
4.7. Progress Bar	58

Chapter 5: Shape Drawing _____ 59

5.1	Button Icon _____	61
5.2	Complex Graphics _____	63
5.3	Image Marking _____	66
5.4	Simple Animation _____	69
5.5	More Complex Figures _____	72

Chapter 6: Device Access _____ 77

6.1.	Sound Output _____	77
6.2.	Reading Sound Files _____	81
6.3.	Sound Recording _____	82
6.4.	Camera Access _____	83
6.5.	System Information Query _____	86
6.6.	Querying Peripheral Devices _____	88
6.7.	File Access Device _____	89
6.8.	A Simple Database _____	91

Chapter 7: Advanced GUI Techniques _____ 95

7.1.	Accessing Mouse Coordinates _____	96
7.2.	Keyboard Input _____	100
7.3.	Continue to Pacman _____	101
7.4.	Window Morphing _____	105
7.5.	Playing Video _____	107
7.5.1.	Fast Image Processing _____	111
7.6.	Image Filtering _____	113
7.7.	Advanced Animation Techniques _____	115

7.8.	Continuous Operation Button _____	118
7.9.	Rich Edit Widget _____	119

APPENDIX -1: ICGUI Functions _____	123
---	------------

APPENDIX -2: ICBYTES Matrix Functions _____	127
--	------------

APPENDIX -3: Creating an ICBYTES Project _____	129
---	------------

Preface

Everyone who is advanced in computer programming has at some point noticed the incompatibility of software technologies. The simplest but most striking example of this is the bare character file. Text files (.txt files) created by the program called "Notepad" in the Windows operating system are not compatible with Linux and MacOS operating systems. The "Line Feed-LF" and "Carriage Return-CR" commands, which are invisible characters added to the end of the line every time you press the "Enter" key while writing those files, are sequenced differently in these three operating systems. Although it does not pose a big problem, the fact that these three operating systems, which have been used dominantly on our planet for a long time, have not been able to develop a common format even on such a simple issue, shows the seriousness of this problem.

As an electronics engineer, I particularly notice this incompatibility in software technologies because hardware technologies are the exact opposite. Popularly described as an assembled computer; Computers built by purchasing desktop computer parts from different manufacturers and assembling them are the most striking example of this. USB, which is also a very successful communication interface, is another example of that. It allows you to connect many devices, from televisions to mouse, without any problems. Another standard is the Ethernet technology. It enables wired and wireless communication of countless types of devices. Moreover, there are even devices that convert these two technologies into each other.

On the other hand, a computer virus that infects Windows cannot infect a MacOS or Linux computer, even if it runs on the same processor. Although this is useful, it actually shows how incompatible these operating systems are. When we dig deeper, the most common problem a programmer encounters occurs while trying to integrate a program written by someone else to his own. If his own program processes the data as a link-list and the other programmer uses a binary tree data structure, he sometimes has to work so hard to integrate these two programs that he has to rewrite his own code that does the same job instead of using someone else's source code or library. The proof of this is that there is an enormous amount of code and libraries that do the same thing in the oldest and longest popular C/C++ language. If you don't believe it, choose a topic and search for its code on the internet. For example, look for a library that does matrix calculations. Why are there so many libraries that do the same job and are free?

This is why the ICBYTES (I See Bytes) library was developed. This library, which takes its name from the data structure at the center of it, uses a single type of variable as the argument used between functions. Thus, it provides unlimited integration. ICBYTES is designed to be able to carry almost all basic data structures within the variable type and to do this dynamically. It can carry many data structures such as matrices of up to four dimensions (x,y,z,w), vector, list, binary tree etc.

The frustrating number of different objects and structures in the OpenCV library, which I started using when I was a PhD student, pushed me to think about this issue, and over the years, I started to design a way to keep many different variable types and data structure definitions in a compressed form. The I-See-Bytes library emerged as a result of this effort.

I assume that, other people with a similar effort have created the Python language, in which variables can hold many different types of objects. Matlab language can also keep different objects in a single variable, although not as many as Python. Recently, these two languages have become the most popular languages for the introductory programming courses of engineering programs at the universities in the USA. However, both Matlab and Python are not suitable for real-time signal, image and video processing applications because these languages run on an interpreter. Real-time image and video processing algorithms must work at high speed, but since interpreters are much slower than compilers, these languages can only be used for research or education purposes in the field of image processing. It is not possible to write industrial quality software in the field of video processing using these languages.

Java language has become extremely popular due to its extensive libraries, but since it runs on a virtual machine, it is much slower than C++ language. Therefore, as you will see in the following sections, although the “I See Bytes” library is written in C++, special precautions have been taken to ensure that it works smoothly on different platforms such as Linux.

İbrahim Cem Baykal

March 2023

1. Design and Terminology

As stated in the Preface, one of the most important reasons for writing the I-See-Bytes library was to create an alternative to the OpenCV library that is much easier to learn and use. If we consider Matlab as a language and not a library, in 2023, around the time this book was written, the OpenCV library was the most accepted Computer Vision (CV) library around the world, especially among academic circles. However, this library is not easy to learn and use. The basic data structure of both Matlab and OpenCV is the matrix. This is because the most suitable data structure for storing and manipulating digital images in memory is a matrix. However, this data structure, called simply "Mat" in OpenCV, is unfortunately very primitive.

This library, which started to be developed in 1999, is widely used especially in academia because it contains a large number of algorithms and is free. However, because it started to be developed 20 years ago, its core still carries the design approach and C++ structures that are 20 years behind. Therefore, it is impractical when writing code. However, in the last 20 years, many additions have been made to the C++ language and design techniques have improved. Another disadvantage is that it is difficult to create professional-looking applications because it is not designed with the Windows operating system in mind. On top of that, its speed slows down when trying to build on it with professional-looking graphical interfaces.

The most primitive part of OpenCV is the fact that the matrix type is determined at compile time, not at run time. When combined with the fact that the data structure is not morphable, OpenCV has no choice but to resort to numerous data definitions and macro definitions to cope with the different data structures and data types it encounters. All these definitions stifle the user and force him/her to constantly refer to description books while writing code. When

OpenCV's documentation deficiencies are added to all these difficulties, it becomes a nightmare for students who are just starting to learn this subject.

1.1. OpenCV Code Examples

If we list the matrix creation methods, which is the basic data structure of OpenCV, it becomes clear how many different definitions the user must keep in mind:

```
Mat M=Mat(3,3, CV_64F); //Creates 3x3 double matrix
M.at<double>(1,3)=5.5; //sets the second row's 4. column to 5.5
```

In the above example, to create a 3x3 double matrix, the user needs to know the macro definition called "CV_64F". If you know that in C++, 32-bit floating point numbers are called "float" and 64-bit floating point numbers are called "double", you may think that there is some logic in this naming. However, if you knew how few of the 4th year computer science students knew how many bits float and double are, you would realize that, students of departments such as electrical, industrial and civil engineering have no chance in hell. Additionally, for those students who don't know what "template" is in object oriented C++, trying to understand what the "<double>" expression creates further confusion.

Here is an even more complicated example:

```
Mat M(400,300, CV_8UC3); //Creates a 300x400 24 bit image
```

In this example, an image with 24 bit color resolution is created. In the computer, colors are produced as a mixture of shades of Red, Green and Blue (RGB). Therefore, every colored pixel should have these 3 components, and every color image should have these three channels. Here, the word "CV_8UC3" is used to create these 3 channels. The first "CV_" means OpenCV, "8UC3" means 8-bit (U→"Unsigned"), that is, positive, and "C3" means 3 channels. Another thing to consider here is the order of definition of image sizes. When talking about image dimensions or resolution, we always write the horizontal resolution first, that is, the resolution in the X → direction, and then the resolution in the Y direction. If we say the size of the image is 400x300, it means it consists of 400 pixels horizontally. However, in OpenCV, the vertical resolution is always written first. This always get new users mixed up. Let's talk about the method of accessing matrix elements. If we want to set the green tone of the pixel at the x:100 y:150 point of this matrix to 255, we need to use a complicated expression like this:

```
M.at<cv::Vec3b>(150, 100)[1]=255;
```

Beginners mostly deal with grayscale (8-bit), color 24-bit or 32-bit images. However, those involved in advanced image processing can deal with any color resolution from 1 bit to 64 bits. When we include floating point as well, the list of some macro definitions that need to be memorized is as follows:

CV_8UC1	CV_16UC1	CV_32SC1	CV_64FC1
CV_8UC2	CV_16UC2	CV_32SC2	CV_64FC2
CV_8UC3	CV_16UC3	CV_32SC3	CV_64FC3
CV_8UC4	CV_16UC4	CV_32SC4	CV_64FC4
CV_8UC(n)	CV_16UC(n)	CV_32SC(n)	CV_64FC(n)
CV_8SC1	CV_16SC1	CV_32FC1	CV_16FC1
CV_8SC2	CV_16SC2	CV_32FC2	CV_16FC2
CV_8SC3	CV_16SC3	CV_32FC3	CV_16FC3
CV_8SC4	CV_16SC4	CV_32FC4	CV_16FC4
CV_8SC(n)	CV_16SC(n)	CV_32FC(n)	CV_16FC(n)

On the other hand, all that is needed to create an image in the ICBYTES library is:

```
ICBYTES M;
```

```
CreateMatrix(M,400,300, ICB_UCHAR); //400x300 grayscale
```

Or,

```
CreateMatrix(M,400,300,3, ICB_UCHAR); //3x8=24 bit color
```

Or,

```
CreateMatrix(M,400,300,4, ICB_UCHAR); //4x8=32 bit color
```

```
CreateMatrix(M,400,300, ICB_UINT); //unsigned int 32 bit color
```

In order to alter the color of the pixel at x:100 y:150 in the grayscale image to white:

```
M.C(100,150)=255;
```

If we want to set the green component to 200 in either 24 and 32 bit images:

```
M.C(100,150,2)=200;
```

would be enough. Additionally, the ICBYTES library provides mechanisms that allow you to access or change the matrix without knowing its type. If you just use parentheses, you can read that element as a double, regardless of the type of matrix.:

```
double d= M(100,150);
```

If your goal is to write into the matrix:

```
Set(100,150,M,5.5);
```

Regardless of the type of the matrix M, the following command will write 5.5 to the (100,150) element. If the matrix is of integer type, it writes 5. If it is float or double, it'll write 5.5.

```
Set(100,150,M,5.5);
```

If you give the last argument an integer, even if it is float or double it will write into M. So, if you use ICBYTES library, you don't need to know the matrix type.

1.2. OpenCV Documentation

Unfortunately, one of the most disastrous aspects of OpenCV is the internet documentation that users turn to learn. For example, the function definition that implements the widely used Shi-Tomasi method is given on the website as follows:

```
void cv::goodFeaturesToTrack(InputArray image,
    OutputArray corners,
    int maxCorners,
    double qualityLevel,
    double minDistance,
    InputArray mask = noArray(),
    int blockSize = 3,
    bool useHarrisDetector = false,
    double k = 0.04
)
```

As can be clearly seen here, an image whose type is defined as "[InputArray](#)" is presented as input to the function. Since the place where [InputArray](#) is written as a link, when you click on it, the website takes you to an explanation that you cannot understand. (At the time of writing, this is what the situation was. They may add better explanations in the future.) However, if we are a little more familiar with OpenCV, we can guess that this is a grayscale matrix, that is, a "Mat" type matrix with 8-bit color resolution. In this case, you would naturally expect the output named "corners", which says "[OutputArray](#)", to be a "Mat" type matrix, right? You would be very wrong. You need to send there the STL (Standard Template Library) vector of a structure in "Point2f" defined in OpenCV. That is, you should create the "image" and "corners" arguments as follows and then send them as arguments to this function:

```
Mat image(Y,X, CV_8UC1);
vector<Point2f> corners;
```

Unfortunately, you can only learn this information from sample program coding, if available. Otherwise, your job is difficult. However, only one type of data structure is used in the I-See-Bytes library, and that is ICBYTES. It can dynamically interpret the bytes within it and transform them into different data structures, hence the origin of its name: **I (only) See Bytes**. Listed below are some of the data structures frequently used in OpenCV. Good luck learning them.

Point2i	Point3i	Point2d	Size2f
Point2l	Point3f	Size2i	Size2d
Point2f	Point3d	Size2l	DataType
Rect2i	Rect2d	Keypoint	Complex
Rect2f	RotatedRect	Range	Scalar

There are also **enum** constants. These actually have no difference in usage from macro variables. However, they are mostly defined for use in special methods and functions, and their number is **enormous**. The table below shows 18 of the around 200 **enum** constants used only in pixel format conversions.

COLOR_BGR2BGRA	COLOR_BGR2XYZ	COLOR_YUV2RGB_NV12
COLOR_RGB2RGBA	COLOR_BGR2YCrCb	COLOR_RGB2HLS_FULL
COLOR_BGRA2RGBA	COLOR_BGR2HSV	COLOR_YUV2RGBA_NV12
COLOR_BGRA2RGBA	COLOR_RGB2Lab	COLOR_YUV2BGR_I420
COLOR_GRAY2RGBA	COLOR_RGB2Luv	COLOR_BayerRGGB2RGBA
COLOR_BGR5552RGBA	COLOR_YUV2BGR	COLOR_BayerRGGB2RGB_EA

The above **enum** constants are used specifically for color-resolution or color space transformations of matrices. Below is how color resolution conversion is done in OpenCV using these constants:

```
cvtColor(input,output,COLOR_RGB2ARGB); //from 24 bit color depth to
                                         32 bit color depth

cvtColor(input, output,COLOR_ARGB2GRAY);//from 32 bit color depth to
                                         8 bit color depth
```

In other words, you need to know not only the output type but also the color structure of the input matrix and select the parameter accordingly. Let's say the input matrix was sent to you by another OpenCV function. Because of the bad documentation, you couldn't find out what color type of matrix this function sent. Your job is hard. If you entered the input matrix type incorrectly, the consequences could be disastrous. Why should we need to know the type of

the input matrix? Why can't this function call "cvtColor()" to detect the type of the input matrix itself? Aren't computers smart yet?

The same is easily done in the I-See-Bytes library as follows:

```
ToRGB32(input, output); //From any color depth to 32 bit color
ToRGB24(input, output); //From any color depth to 24 bit color
Luma(input, output);    // From any color depth to 8 bit grayscale
                        color depth
```

Here we have only given a few examples of OpenCV's cumbersome and complex structure dating back to the Jurassic age. Unfortunately, most of OpenCV actually consists of smaller libraries written by other people, called “contribs” (https://github.com/opencv/opencv_contrib). They use data structures completely at their discretion and provide little or no documentation of this. You can hear nightmarish anecdotes of these from people who are familiar with OpenCV.

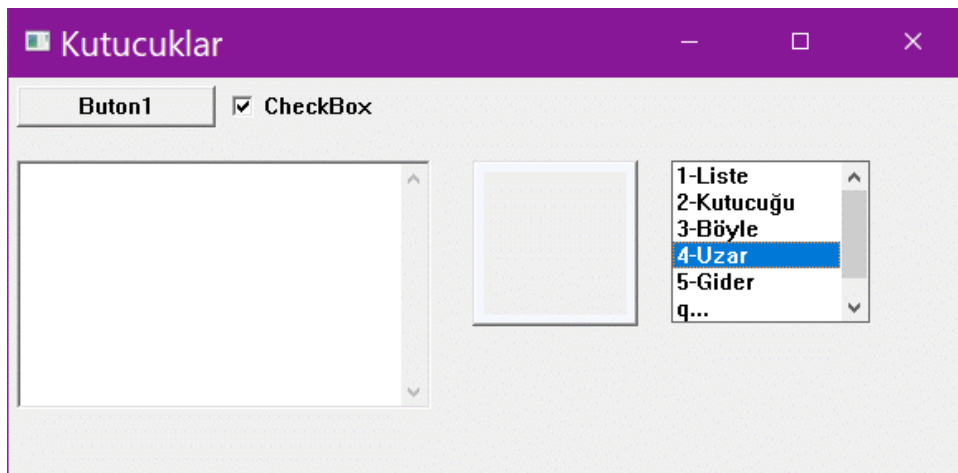
1.3. New Terminology

1.3.1. Fluid

The word "variable", and “object” are frequently used as programming terms. However, the ICBYTES type structure used in the I-See-Bytes library is neither a variable nor an object. In addition to structures such as matrix, binary tree, STL vector, set, it can also contain variable types such as int, float, long long, which are the basic data types in C++. More importantly, it can dynamically transform from one of these structures to another during run time. For the reasons mentioned above, objects of type ICBYTES will be referred to as “fluid” in this book. In this way, it will be ensured that it is differentiated from data types and objects that hold fixed types at run time.

1.3.2. Widget

The image below shows buttons, text boxes, checkboxes, panels and similar tools. In the Windows operating system, all of these tools are derived from an object called "Window". That's where the name of this operating system comes from. Other operating systems call such graphical user interface objects "widgets". This word is generally used more widely. For this reason, the word "widget " will be used in this book as well.



1.4. Function Naming Rules

There are two basic fluids in the I-See-Bytes library. These are:

1. ICBYTES: Holds data and data structures.
2. ICDEVICE: It provides access to any device connected to the computer with which data can be exchanged (for example, a camera).

If at least one of the function arguments is one of these fluids that are unique to our library, the function name is written directly with its meaning specific to it as there will be no conflict of function names. For example:

CreateMatrix(ICBYTES &m, ...)

However, if this is not the case, one of the following prefixes are used at the beginning of the function:

- ICB_
- ICG_

2. The Graphics Interface Sublibrary: ICGUI

The graphical user interface (GUI) has become an indispensable part of computer programming. Since no one in the market writes console programs, there is no need to insist on teaching programmers how to program on the console. Within the I-See-Bytes library, many functions are provided as a sublibrary to enable graphical interface programming. This library is called I-See-GUI. It is briefly written as ICGUI. This library is built on WIN32 API. WIN32 API is the lowest-level, most basic programming interface of the Windows operating system. As a result, it can run on every Windows machine without the need for any extra files such as DLLs and etc.

Since WIN32 API is the most basic programming interface of Windows, it works extremely fast, which is one of the most important goals of the I-See-Bytes library. It is supported by Microsoft Visual Studio and GNU C++ (MinGW) compilers, which are the most popular C++ compilers.

However, there is another very important advantage of using the WIN32 API. By using the WINE (Wine Is Not an Emulator) compatibility layer, it can run smoothly and at almost the same speed on Linux and MacOS operating systems. WINE is an open-source compatibility layer. In other words, your code runs directly on the real processor, not on the virtual processor as in Java. Therefore, it is very fast. As a result, the EXE files you create using the I-See-Bytes library will run smoothly on Linux with WINE installed. This significantly reduces the need for Java.

2.1. Hello World

The code below demonstrates the simplicity of the I-See-GUI library. As we all know, every C++ program starts with the “main()” function. In the ICGUI library, there are two functions that initiate the program:

```
void ICGUI_Create(){}

```

```
void ICGUI_main(){}

```

Even if these two functions are called (or rather defined) empty, as shown in the code below, they create a window of 800x600 pixels. But of course, there will not be another widget on this window. In order to create widgets, the functions that create widgets must be called in ICGUI_main().

```
#include"icb_gui.h"

int MLE; // Multi-Line text editor window handle

void ButtonFunction(); //Button function declared

void ICGUI_Create()//Main window created
{ }

void ICGUI_main()//<main> is a must in C++
{
    ICG_Button(5, 5, 120, 25, "BUTTON", ButtonFunction);
    MLE = ICG_MLEditSunken(5, 80, 400, 400, "", SCROLLBAR_V);
}

void ButtonFunction ()//These will happen when the button is pressed
{
    for(int x=1;x<11;x++)
        ICG_printf(MLE, "%d-Hello World!\n",x);
}
```

In the code above, a button (ICG_Button) and a text editor (ICG_MultilineEditSunken) widgets are created. All functions that create widgets return an integer. This integer becomes the handle of that box. By using this identifier, it is possible to access that widget in another part of the program. Generally, these identifiers are defined globally so that they can be accessed by any function.

```
ICG_Button(5, 5, 120, 25, "BUTTON", ButonFunction);

```

Descriptors for buttons are generally not needed because once they are created, it is not necessary to make any changes to them until the program terminates. The first two arguments of the widget creating functions are always coordinate information. These are the pixel coordinates of where the upper left corner of the widget will be located in our window. The next two arguments are the width and height, also in pixels. The fifth argument is the text that will be written on or inside the widget when it is first created, and it can be written in quotes ("BUTTON") or it can be a pointer in char. The last argument is the name of the function that will be called when the button is pressed.

```
MLE = ICG_MLEditSunken(5, 80, 400, 400, "", SCROLLBAR_V);
```

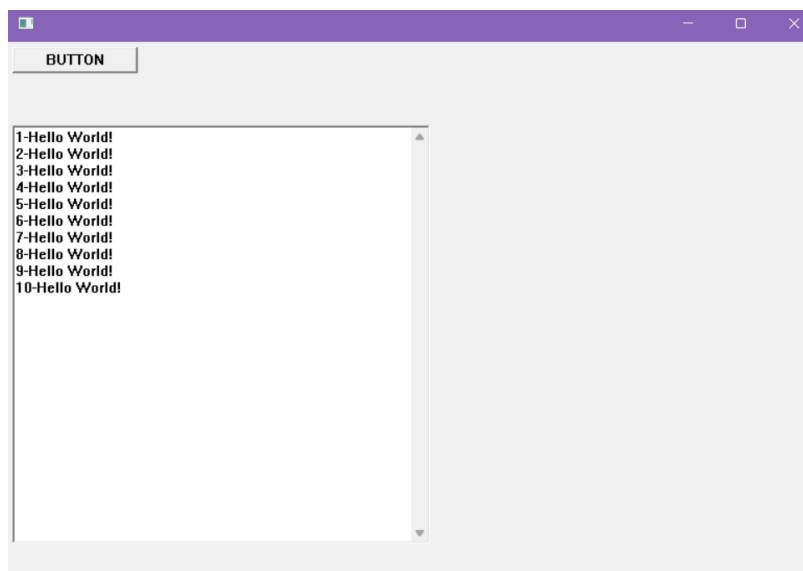
By calling this function, we create a “multi-line” (Multi Line Edit (MLEdit)) text editing box that looks like its edges are caved in. If we do not want anything to be written in it when it is first created, we specify the fifth argument as two consecutive quotes (""), without spaces. The last argument states that we want to have a vertical scrollbar.

When the button is pressed, the function named “Buttonfunction()” is called. This function is first declared and then defined. With a simple for loop, “Hello World” is printed on the screen 10 times.

Here is the function used to print inside the edit box:

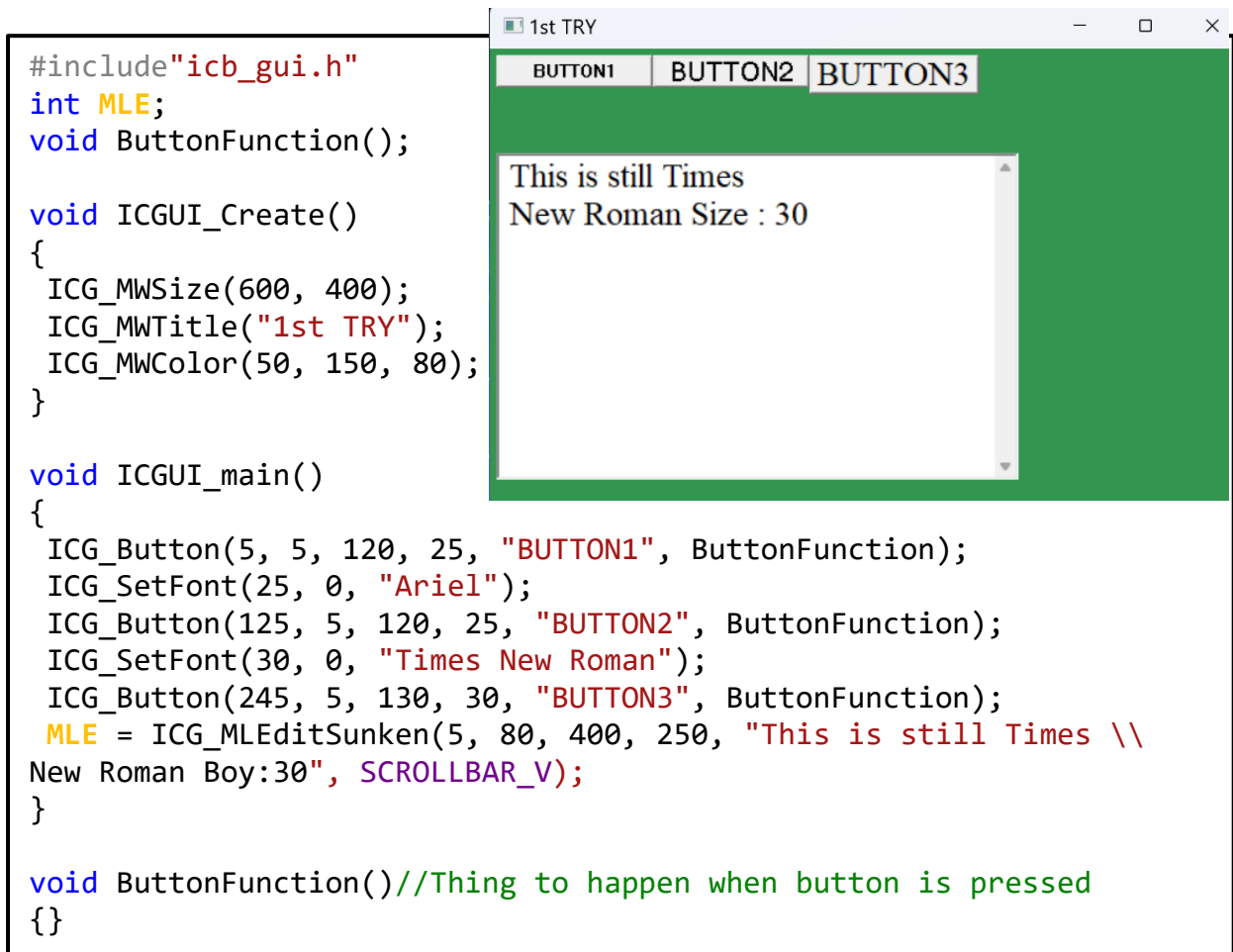
```
ICG_printf(MLE, "Hello World!\n");
```

It is almost the same as the “printf()” function that C++ programmers know very well. The only difference is that its first argument is the handle of the edit box. You may have more than one edit boxes in your window. By giving the handle as input, you specify which box you want it to write in. Below is the result of this program.



2.2. Width, Length, Color and Style

In the code example below, we see that three functions are called within the “ICG_GUICreate()” function, which creates our main window:



As seen above, there have been significant changes in the appearance of our application. The most noticeable of these changes is the main window color being green. Additionally, the size of the main window has also decreased. The functions that reveal these changes are realized by three functions within the “ICGUI_Create()” function. Now let's examine them one by one:

ICG_MWSize(600, 400)

This function ensures that the main window of our application is 600x400 pixels tall. When we do not use this function, the default size of our application is 800x600 pixels.

ICG_MWTitle("1st TRY")

This function is used to change the name of our main window. If we do not use this function, the default “ICB_GUI” is used as in the previous example.

ICG_MWColor(50, 150, 80)

Function that allows the color of our main window to change. In the computer, all colors are created as a mixture of red, green and blue. The inputs of this function specify what brightness the window color will be as a mixture of these three colors. 50 for red, 150 for green and 80 for blue. These values must be between 0 and 255. When all are 0 or close to 0, black color appears. If they are all the same, shades of gray occur. When all are 255, white color is obtained.

Another noticeable change in this application is the use of different font types and sizes on the buttons. While the stock system font was used for Button1, Ariel font size 25 was used for Button2, and Times New Roman font size 30 was used for Button3. Below is the function which created these font changes:

```
ICG_SetFont(25, 0, "Ariel");
```

The first two inputs of this function are the height and width of the font. However, if the width is given as zero, the height value is used for both. The third entry specifies the name of the font. If you pay attention, you will see that this function only affects the widgets created after it. So, after calling this function once, all the boxes you create, such as buttons or text boxes, will use the same font. An example of this is seen when we create a text box after Button3 is created. This text box still uses Times New Roman font.

2.3. ICGUI and ICBYTES

In this section, we will give examples of how ICGUI, a graphical interface library, and the ICBYTES library, which creates fluids, work together. With these examples, we will learn how different purposes fluids created with the ICBYTES type can be used. In our first example, we will see the functions and interfaces related to the matrix structure, which is one of the main purposes of use of ICBYTES. In the code example below, two buttons are defined. Two different matrix creation methods are shown with the functions called when these buttons are pressed.

```
#include"icb_gui.h"
int MLE,FRM;
void PrintMatrix(); //Matrix Button function decleration
void Image(); // Image Button function decleration

void ICGUI_Create()
{
    ICG_MWSize(600, 400);
    ICG_MWTitle("ICGUI and ICBYTES");
}

void ICGUI_main()
{
    ICG_SetFont(25, 0, "Consolas");
    ICG_Button(5, 5, 120, 25, "Matrix", PrintMatrix);
    ICG_Button(125, 5, 120, 25, "Image", Image);
    MLE = ICG_MLEditSunken(5, 80, 250, 250, "", SCROLLBAR_V);
    FRM = ICG_FramePanel(260, 80, 250, 250);
}

void PrintMatrix()//Things to happen when Matrix button is pressed
{
    ICBYTES A = {{1,2,3},{4,5,6},{7,8,9}};
    DisplayMatrix(MLE, A);
    A= "ABCDEF";
    DisplayMatrix(MLE, A);
    Print(MLE, A);
}

void Image()//Things to happen when Image button is pressed
{
    ICBYTES A;
    CreateMatrix(A, 250, 250, 3, ICB_UCHAR);
    A = 100;
    for (int x = 50; x < 200; x++)
        A.B(x, 100, 2) = 255;
    DisplayImage(FRM, A);
}
```

In this example, a new type of widget is created within the “ICGUI_Main()” function. This box is used to display images and is called Frame. Frame means framework in English. So, we create a frame that displays the picture. The name of this function is “FramePanel” because its shape is outward like a panel. There are different styles of frameworks in ICGUI. These will be mentioned in the following sections.

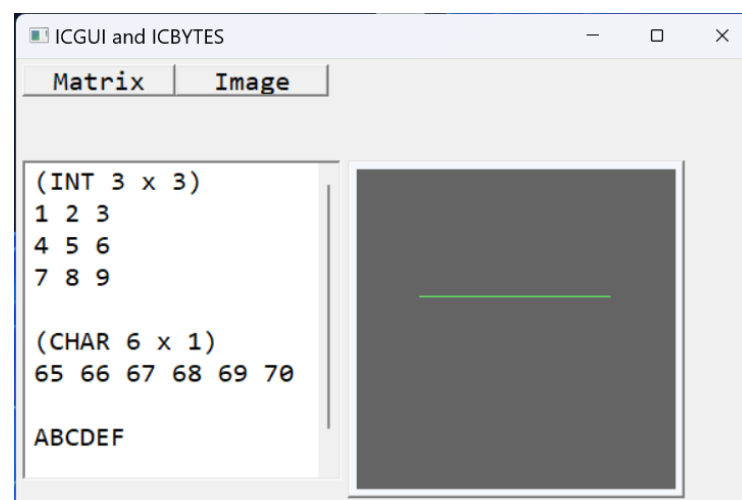
As for the first button, in the code of this button, a fluid in ICBYTES is created, and a matrix consisting of integers of size 3x3 is created in it. The line that does this is shown again:

```
ICBYTES A = {{1,2,3},{4,5,6},{7,8,9}};
```

Then, the function named “DisplayMatrix()” was used to print this matrix in the multi-line text editor. Immediately afterwards, the characters “ABCDEF” were written into fluid A:

```
A= "ABCDEF";
```

In other words, while fluid A was carrying a two-dimensional array, it suddenly started carrying a character array (string). Then, fluid A was printed in two different ways in the multi-line text box as shown below.



In the function called “Image()” belonging to the second button, the matrix is created with a different method:

```
ICBYTES A;  
CreateMatrix(A, 250, 250, 3, ICB_UCHAR);
```

First, a fluid named "A" is created, and then this fluid is asked to carry a matrix of 250x250x3 dimensions and of type "unsigned char". In the previous section, we said that all colors are created on the computer by mixing shades of red, green and

blue. Therefore, if we want to create a color image of 250x250 pixels, the X and Y dimensions of our matrix must be 250. Since there must be 3 values for each color (for red, green and blue), it is understandable why the size of our matrix should be 250x250x3.

Matrices in ICBYTES can have up to four dimensions, and dimensions are labelled (X,Y,Z,W). This sorting method is different from Matlab and OpenCV and users of these software should be careful about this. C/C++ users should also remember that matrix indexes start from one, not zero, unlike C/C++ arrays.

In the next line:

```
A = 100;
```

With this assignment, all elements in the matrix are set to 100. So, in this single line, 100 is written into 250x250x3=187,500 elements. As a result, our entire image has a dark gray tone. As we have stated before, if we assign the same values to the tones of red, green and blue (hereinafter abbreviated as RGB), we obtain shades of gray.

Then, let's examine the lines that draw a horizontal green line on this gray color:

```
for (int x = 50; x < 200; x++)
    A.B(x, 100, 2) = 255;
```

Here, we see that the x value is increased from 50 to 199 with a "for" loop, and each time the horizontal pixels 50, 51, 52, ... of the 100th row of matrix A are reached and the second, that is, green tone, is increased to the maximum value of 255. This causes the pixel color from (50,100) to (199,100) to change from dark gray to green, causing these tiny pixels adjacent to each other to appear as a line to our eyes. An important point that we should not forget here is the expression we use to access the matrix elements:

```
A.B()
```

Putting a dot after the fluid "A" and writing B() shows that we want to access the A matrix as (B)yte, that is, "unsigned char". In this way, we can read any elements of matrix A or change their values. In the last line of our code:

```
DisplayImage(FRM, A);
```

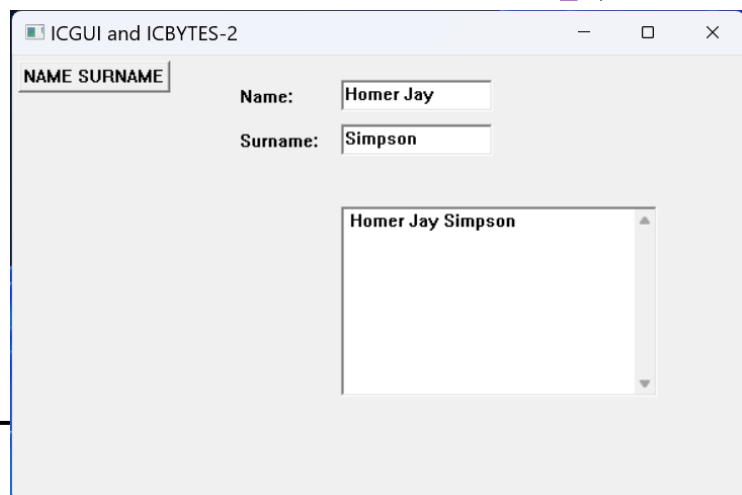
This line gives the command to display the "A" matrix as a picture (image) in the "FRM" frame. Remember, this is the frame we created by calling the "ICG_FramePanel()" function in ICGUI_Main().

2.4. Reading Text from the Screen (Input)

Almost every program needs to receive information from the user. The ICBYTES library also makes it easier to get information from the screen. The string of characters in any text box can be easily copied by the ICBYTES fluid. In the example below, the user enters his/her name and surname into single-line text boxes. Then, when the "NAME SURNAME" button is pressed, our program writes them in a multi-line text box with a space between them.

```
#include "icb_gui.h"
int SLE1, SLE2, MLE;
void namesurname(); // NAME SURNAME button function declared
void ICGUI_Create()
{
    ICG_MWSize(600, 400);
    ICG_MWTitle("ICGUI and ICBYTES-2");
}
void ICGUI_main()
{
    ICG_Button(5, 5, 120, 25, "NAME SURNAME", namesurname);
    ICG_Static(180, 25, 60, 25, "Name:");
    ICG_Static(180, 60, 80, 25, "Surname:");
    SLE1 = ICG_SLEditSunken(260, 20, 120, 25, "");
    SLE2 = ICG_SLEditSunken(260, 55, 120, 25, "");
    MLE = ICG_MLEditSunken(260, 120, 250, 150, "", SCROLLBAR_V);
}

void namesurname ()
{
    ICBYTES name, surname;
    GetText(SLE1, name);
    GetText(SLE2, surname);
    Print(ML, name);
    ICG_printf(ML, " ");
    Print(ML, surname);
}
```



When we examine our program, we encounter a new function in ICGUI_main():

```
ICG_Static(180, 25, 60, 25, "Name:");
ICG_Static(180, 60, 80, 25, "Surname:");
```

This function prints "Name:" on the gray background of our window and "Surname:" below it. In fact, this type of widget is called a "static" box because it

allows writing static text in an invisible window, that is, something that the user cannot change (not the programmer). The coordinates, width and length of the window are determined by the first four arguments of the function. The single-line (Single Line Edit (SLEdit)) text boxes located right next to these indicate to the user which boxes they should write their names and surnames in.

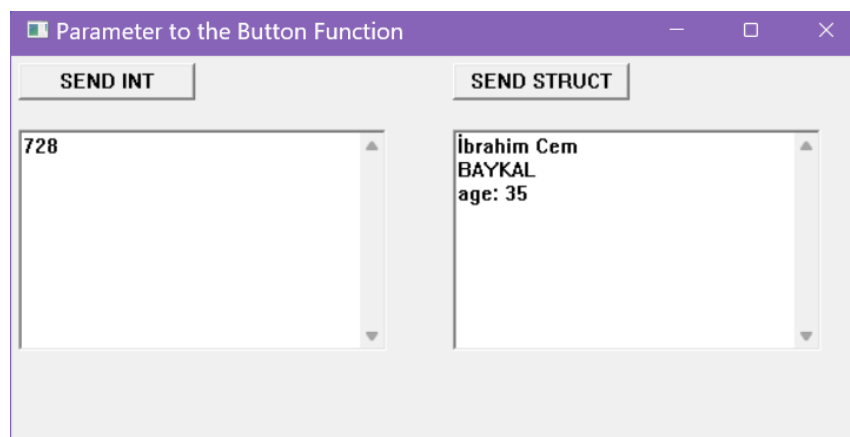
As for the function called "namesurname()" that is called when the button is pressed, we encounter a new function here:

```
GetText(SLE1, ad);
```

In this function, the text in the single-line text box written by the user as "Reşat Nuri" is copied, that is, the text "Reşat Nuri", to the fluid named "ad". In the next line, the same function copies the text "Güntekin" in the second single-line text box to the fluid named "surname". The functions in the next three lines print these fluids in a multi-line text box (Multi Line Edit (MLEdit)) with a space between them..

2.5. Sending Parameters to a Button Function

In the programs explained so far, it has been shown how to define the function that is called when a button is pressed, but it has not been mentioned how to send a parameter to this function from ICGUI_main(). If desired, only one parameter can be sent to the button function and this parameter must be a void pointer. A pointer is the address of any variable or object in memory. In other words, instead of sending variables or fluids, their addresses are sent. Users usually send more than one parameter to functions. Moreover, these are generally int, float or double types. That being the case, why does the ICGUI library allow only one parameter and want it to be a pointer? Because the pointer is an address in memory and numerous variables that are sequentially located at that address can be passed to the function with this method.



There are two buttons in the application whose screenshot is given above. When you press the "SEND INT" button, 728 is written in the text box on the left. This number is actually created in ICGUI_main() as an "int" variable, then sent to the function of this button and printed by this function in the text box on the left. The "SEND STRUCT" button on the right sends two-character strings and one integer, and its function also prints them in the text box on the right. So how does this happen? The code example below shows step by step how to do this.

```
#include"icb_gui.h"
#include<string.h>

int MLE1, MLE2;

struct human
{
    char name[32];
    char surname[32];
    unsigned int age;
};

void intfunction (void*p);
void structfunction (void* p);

void ICGUI_Create()
{
    ICG_MWSize(600, 400);
    ICG_MWTitle("Parameter to the Button Function");
}
```

At the beginning of our program, the handles of multi-line text boxes (MLE1, MLE2) are defined. Then we see that a structure (struct) is defined. The type definition of this struct is "human" and it has three members. The first member is a 32-letter long character string whose name is "name". The second is an index named "surname" with the same properties, and the third is a positive integer named "age".

After the definition of this struct, we see the definitions of the button functions. In the definitions this time, we can see that the functions take a pointer of type void as input. After these definitions, an ICGUI_Create() function with classical content comes. Now let's come to the main function of our program:

```

void ICGUI_main()
{
    static int i=728;
    static human C;
    strcpy_s(C.name, "İbrahim Cem");
    strcpy_s(C.surname, "BAYKAL");
    C.age = 35;
    ICG_Button(5, 5, 120, 25, "SEND INT",intfunction,(void*)&i);
    ICG_Button(300,5,120,25,"SEND STRUCT",structfunction,(void*)&C);
    MLE1 = ICG_MLEditSunken(5, 50, 250, 150, "", SCROLLBAR_V);
    MLE2 = ICG_MLEditSunken(300, 50, 250, 150, "", SCROLLBAR_V);
}

```

We see that an integer named “i” is created within the main function. Any variable that will be sent as input to button functions must be defined as "static". The reason for this is that these variables must continue to exist even after the “ICGUI_main()” main function ends. The main function may end after creating the widgets, as in this example. This does not mean that the program will end. Here, after creating the integer “i”, we write 728 into it and copy its address to the button function. not himself.

After this, we create a struct of type "human" whose name is "C". We do not forget to use the word “static” when creating this. Then, using the sentence copy function of the C++ language, we write a name into C's "name" directory and a surname into the "surname" directory. When you type 35 in C's age element, this struct is ready to be sent to the button function.

When we examine the button functions, we see that type casting is applied in two different ways:

```

void intfunction(void *p)
{
    ICG_printf(MLE1, "%d ", *(int*)p);
}

void structfunction (void* p)
{
    human *i = (human*)p;
    ICG_printf(MLE2, "%s\n", i->name);
    ICG_printf(MLE2, "%s\n", i->surname);
    ICG_printf(MLE2, "age: %d\n", i->age);
}

```

In the first function, the following expression is used for type shifting:

```
*(int*)p
```

In this expression, the expression “(int*)” in parentheses in front of the “p” pointer shifts p from a void type pointer to an integer type pointer. Then, the asterisk “*” in front of the expression “(int*)” gives us the integer value shown by this pointer, and this value is printed in the text box on the left.

In the second function, our void type pointer p is converted to a consecutive pointer of type "human" using type shifting and copied to the pointer "i" created in the function of the same type. After this, the elements of the "human" sequence are accessed via "i" and they are printed one by one in the text box on the right.

2.6. Check Box

You may create a check box just like a button. The function of a checkbox must have an integer (int) argument:

```
void Check(int k)
{
    ICG_printf(MLE, "%d ", k);
}

void ICGUI_main()
{
    ICG_CheckBox(615, 5, 135, 25, "CHECK BOX", Check);
}
```

☐ CHECK BOX

☒ CHECK BOX

2.7. Image Loading

I-See-Bytes student version can read jpeg and bitmap images and convert them into matrix. The code example below shows how to do this. In this example, the “icbimage.h” header file has also been added. This file contains the definitions of the functions that will allow us to load the image. Two handles are defined.

```
#include "icb_gui.h"

int MLE1, FRM1;

void buttonfunc();
void ICGUI_Create(){}
```

The rest of the code contains most familiar functions. We only see different commands in the button function.

```

void ICGUI_main()
{
    ICG_Button(5, 5, 120, 25, "Button1", buttonfunc);
    MLE1 = ICG_MLEditSunken(5, 50, 250, 150, "", SCROLLBAR_V);
    FRM1=ICG_FramePanel(280, 50, 100, 100);
}

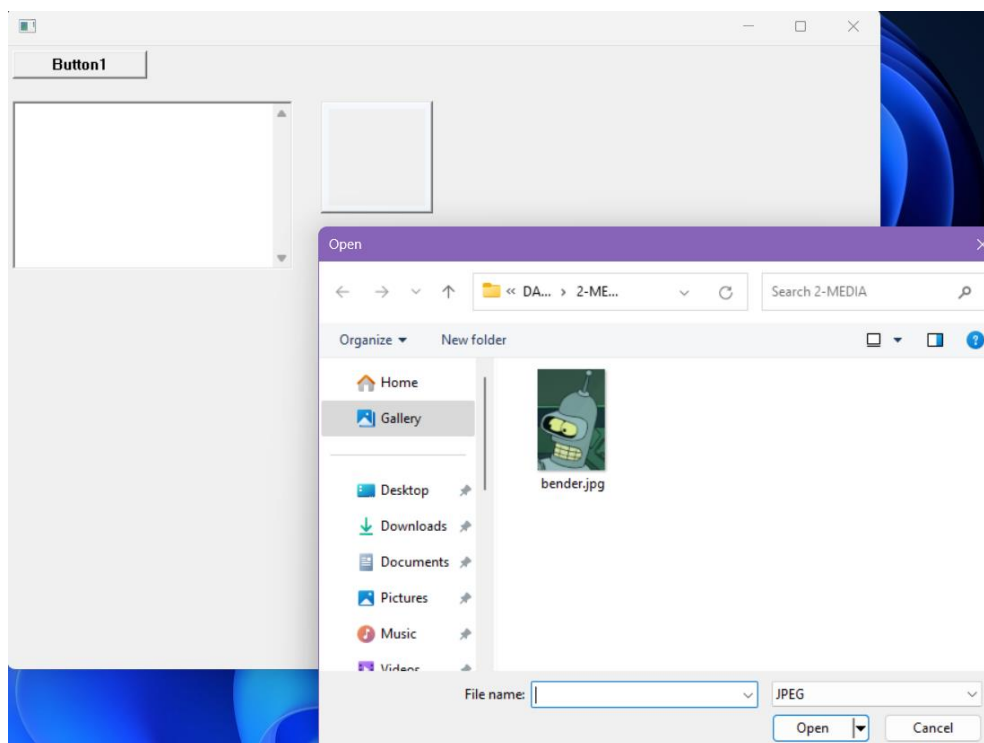
void buttonfunc ()
{
    ICBYTES path,image;
    ReadImage(OpenFileMenu(path, "JPEG\0*.JPG\0"), image);
    Print(MLE1, path);
    DisplayImage(FRM1, image);
}

```

Two fluids are created in the button function. The purpose for which they will be used is already clear from their names. Next, we see two nested functions. First, let's look at the definition in "icb_gui.h" since the function inside will be run:

```
char* OpenFileMenu(ICBYTES& path, const char* type);
```

This function takes a fluid and a character pointer as input. When this command is run, the following file selection window opens:



If you pay attention to the bottom corner of this window, you will see that it only allows you to select **jpeg** type image files. The reason for this is that the

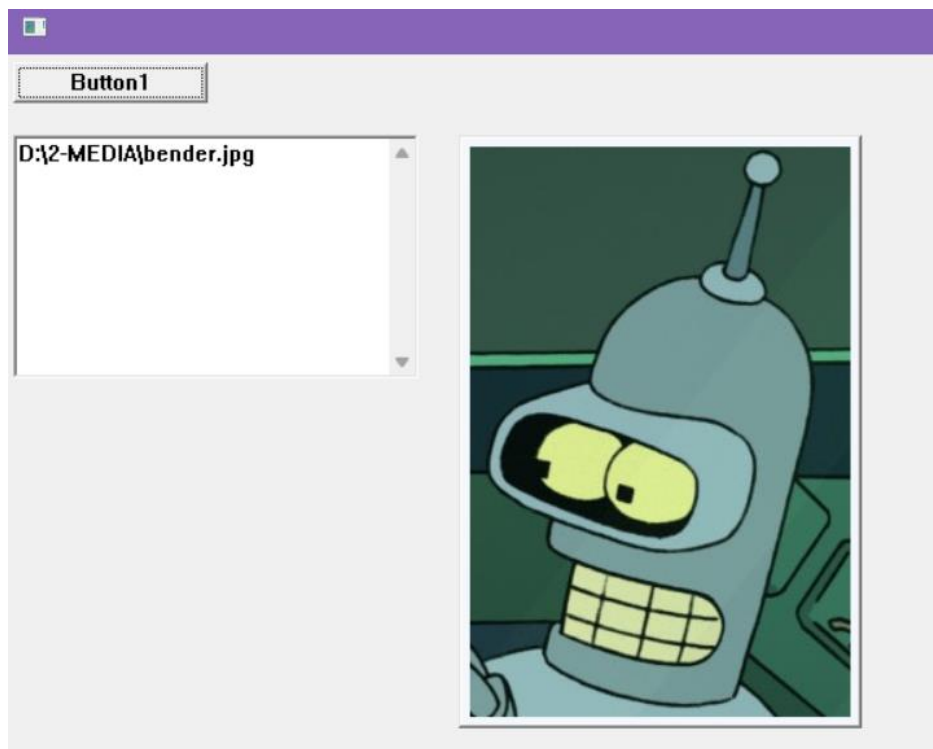
character string "JPEG\0*.JPG\0" is given as the second argument to this function. This string indicates that only files with the "*.JPG" extension can be selected. The name and file path of the selected file are placed in the first argument, fluid. This file path is also returned by the function as a pointer to a character string. This pointer is used as input to the external "ICB_ReadImage()" function. If we examine the definition of this function in the "icbimage.h" header file,

```
bool ICB_ReadImage(const char * filepath, ICBYTES& i);
```

We can see that the first input is a character pointer and the second is in terms of an ICBYTES fluid. As the English name suggests, the character pointer points to a string containing the name and full file path of a file. This pointer is the pointer returned by the "OpenFileMenu()" function. If the file is valid and the jpeg or bitmap is the file of a file, the "ReadImage()" function opens this file and loads it into a matrix and loads this matrix into the fluid named "image". The following two lines create the screenshot below.

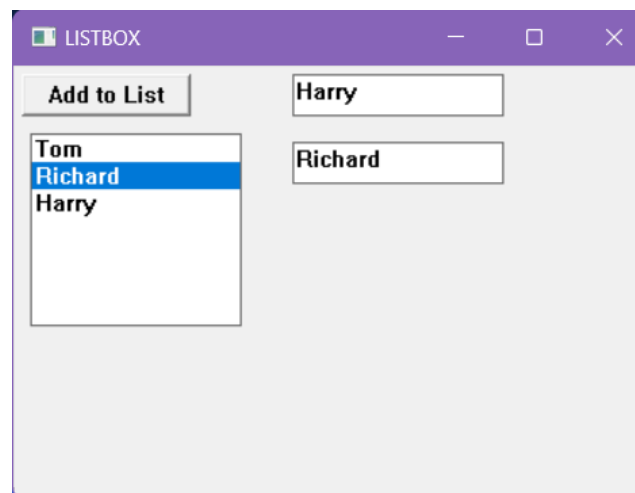
```
Print(MLE1, path);  
DisplayImage(FRM1, image);
```

In this example, the image file named "bender.jpg" in the "2-MEDIA" directory on the D: disk is loaded and shown in the frame.



2.8. List Box

We call the graphical interface structure that allows you to display items consisting of words or sentences as a selectable list, a list box. Below is a screenshot of an application that uses this widget. In this application, the string you write in the single-line text box at the top right is added to the list box as a different item every time you press the "Add to List" button. For example, as seen in the picture below, "Tom," "Richard," and "Harry" were written in the text box at the top (the box labeled Harry) and then the button was pressed each time.



After these three names are added to the list, when we hover the mouse cursor over the word "Hüseyin", which is in the second place in the list box, and double click the left button, the selected item (which in this case is Hüseyin) is written in the second text box at the bottom left. Below is the C++ code of this application.

```
#include "icb_gui.h"

int LB1, SLE1, SLE2;

void ICGUI_Create()
{
    ICG_MWTitle("LISTBOX");
    ICG_MWSize(400, 300);
}

void ListboxFunc(int index)
{
    ICBYTES addthis;
    ICG_GetListItem(LB1, index, addthis);
    SetText(SLE2, addthis);
}
```

```

void ButtonFunc()
{
    ICBYTES readthis;
    GetText(SLE1, readthis);
    ICG_AddToList(LB1, & readthis.C(1));
}

void ICGUI_main()
{
    ICG_Button(5, 5, 100, 25, "Add to List", ButtonFunc);
    SLE1= ICG_SLEditBorder(165, 5, 125, 25, "");
    SLE2 = ICG_SLEditBorder(165, 45, 125, 25, "");
    LB1 = ICG_ListBox(10, 40, 125, 125, ListboxFunc);
}

```

The first thing that catches our attention in ICGUI_main is the ICG_ListBox function that creates a list box. As with all other widget creation functions, the first four arguments of this function determine its position and dimensions. The fifth argument is the function named "ListboxFonk", which has an input in integer (int) and is defined above. This function is called when you double click on an item in the list box. When we examine the definition of ICG_ListBox, we see that it actually has another optional argument:

```

int ICG_ListBox(int x, int y, int width, int height, void (*callback)(int),
               bool sort=false);

```

If the bool argument named "sort" at the end is optionally set to "true", the list will sort the items written into the box alphabetically rather than in the order they were sent. If this argument is not given, as we did, the asset value false is used..

When we examine the button function, we see that the contents of the text box at the top are transferred to a fluid named "read" using the SL1 handle, and then the ICG_AddToList() function is called to add this fluid to the list box. Since the second argument of this function must be a pointer in char, the address of the first element of the vector in the fluid is sent.

In this code example, we encounter a box function that takes integer input for the first time:

```

void ListboxFunc(int index)

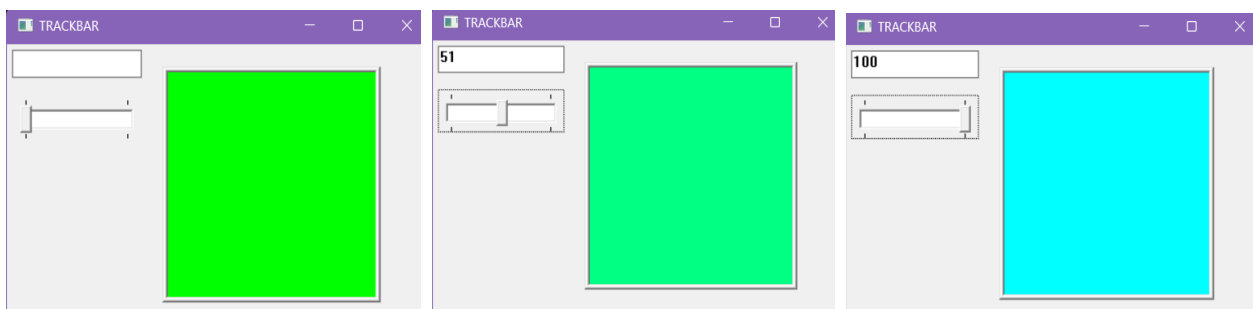
```

The entry "index" here is the number of the list item you double-clicked. This number starts from zero. In the next step of the program, we see that by giving this

index number to the `ICG_GetListItem()` function, the item in that index row is copied to a fluid named "write". In the last step, this fluid is printed in the text box accessed with the `SLE2` handle. In other words, when we double click on the "Hüseyin" item in the list box, our function gets an index number, we copy the first item to the "write" fluid with this index number, and then print the content of this fluid into the second text box.

2.9. Track Bar

We call the step box the box that looks like the adjustment buttons on electronic devices that are used by pulling left or right to change settings such as sound volume or image brightness. This box produces integers between certain values by scrolling left and right when created horizontally, or up and down when created vertically. These integers are used as parameters to other functions in the program. The image below shows an example application that changes the color of an image using the step box. When this application is first started, the leftmost window appears. The picture inside the frame is pure green. Then, as shown in the middle picture, when the level is drawn to the middle point, the green loses its purity. On the far right, when the step setting is increased to the last point, the picture turns into pure turquoise. In this application, the picture can turn into any tone between pure green and pure turquoise, depending on the value of the step setting. You can think of this like color adjustment on a television.



Now let's see the source code of this application:

```
#include "icb_gui.h"

int FRM1, SLE1;
ICBYTES image;

void ICGUI_Create()
{
    ICG_MWTitle("TRACKBAR");
    ICG_MWSize(430, 300);
}

void LevelSetFunc(int level)
{
    ICG_SetWindowText(SLE1, "");
    ICG_printf(SLE1, "%d", level);
    image = 0xff00 + level * 2.55;
    DisplayImage(FRM1, image);
}

void ICGUI_main()
{
    CreateImage(image, 200, 200, ICB_UINT);
    SLE1 = ICG_SLEditBorder(5, 5, 125, 25, "");
    ICG_TrackBarH(5, 45, 125, 40, LevelSetFunc);
    FRM1 = ICG_FrameMedium(150, 20, 200, 200);
    image = 0xff00;
    DisplayImage(FRM1, image);
}
```

The function that creates the step box in the ICGUI_main() function:

```
ICG_TrackBarH(5, 45, 125, 40, LevelSetFunc);
```

Apart from four position and size values, it also takes a function called "LevelSetFunc" as an argument. The point to consider here is the vertical height of the step box. In order for this box to be drawn fully, it must be at least 40 pixels wide.

In the ICGUI_main() function, unlike previous sample programs, an "unsigned int" matrix with dimensions of 200x200 pixels is created in the image fluid. In previous examples, we used the following command to create a 24-bit color image:

```
CreateMatrix(A, 250, 250, 3, ICB_UCHAR);
```

The command above creates a 3-channel (KYM) matrix of 250x250 size and 8 bits for each channel. The command we use now is

```
CreateImage(image, 200, 200, ICB_UINT);
```

It created a 200x200 matrix with a single channel, but where the channel itself was 32 bits. This type of image allocates 8 bits for alpha and 24 bits for RGB.

In the I-See-Bytes library, the 32-bit image matrix can be created in two different ways. You can either create a 200x200 matrix with a single channel of 32 bits, as we did, or a 200x200x4 matrix with 8 bits per channel:

```
CreateImage (A, 200, 200, 4, ICB_UCHAR);
```

Both of these produce the same type of image. However, if you want to access these matrix elements, you need to access them through different methods.

In the continuation of the main function, after the image matrix is created, we see the following statement in the fifth line:

```
image = 0xff00;
```

In this expression, each element of the image matrix is equal to the hexadecimal number 0xff00. This number turns each pixel of the image into pure green. So what does this number mean? To understand this, we must first remember that each digit written in hexadecimal corresponds to 4 bits. So we need to use two digits for 8 bits and eight digits for 32 bits. Since we are trying to write each element into a matrix with a 32-bit number, we should actually write this number like this:

```
image = 0x0000ff00;
```

However, since zeros in front of a number are meaningless, we chose the shorter version. When written in long form, each pair of digits corresponds to a color channel:

```
0x 00 00 ff 00
  A  R  G  B    (Alfa, Red, Green, Blue)
```

Since the number ff is 255, which is the maximum value a channel can have in decimal order, and the value of other color tones is zero, our application prints a pure green image on the screen as soon as it is run.

After the creation in the Main function is completed, the action takes place only in the LevelSetFunc () function of the step box. As seen in the code above, the bit integer named "stage" is input to this function. This function is called whenever the step bar is touched, that is, even when its position has not changed. The range

of the integer that is the function input is set between 0-100. However, this range can be changed with other functions.

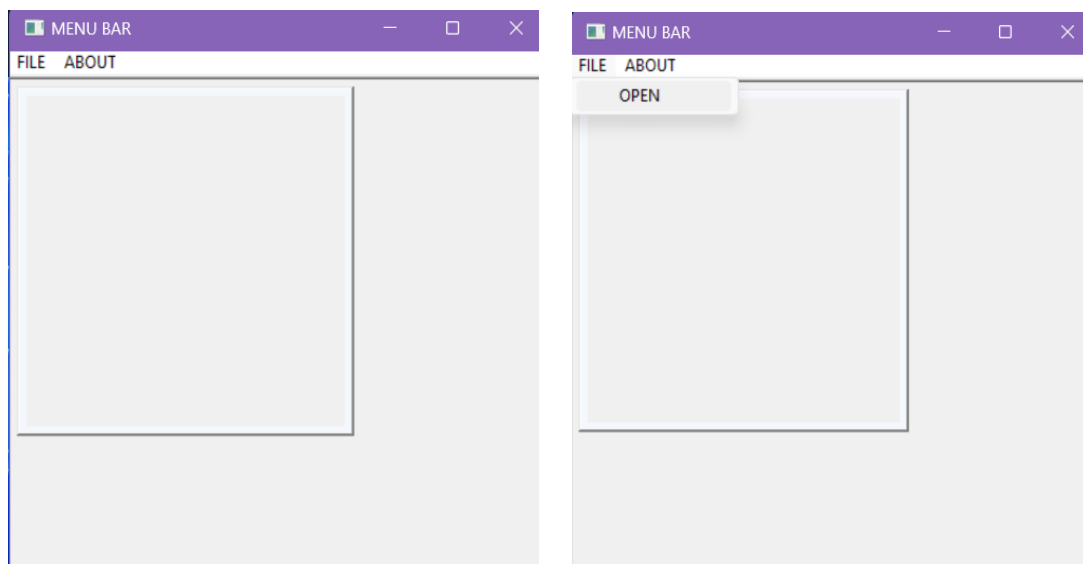
If we examine this function step by step, we notice that in the first line it deletes the inside of the text box to which the SLE1 handle is attached. Then we see that he writes the value of the “step” variable into this text box, and then sets each element of the image matrix equal to this value:

```
image = 0xff00+ level * 2.55;
```

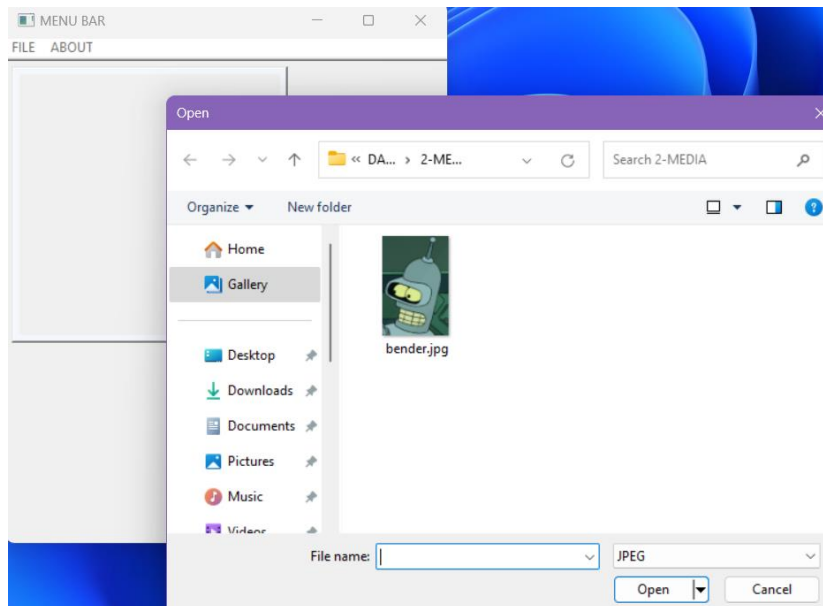
In this line, the 0xff00 value that we used previously to color the matrix green is added to a number between 0 and 255. In other words, it is added with a number between 0x0000 and 0x00ff in hexadecimal order and the entire matrix becomes this number. If we add 0xff00 and 0x00ff, there will be no overflow. So, while the green tone remains at its maximum value of 255, we strengthen the blue tone every time we increase the step bar. As a result, when both blue and green tones reach their maximum value of 255, we obtain the pure turquoise color.

2.10. Adding Menus

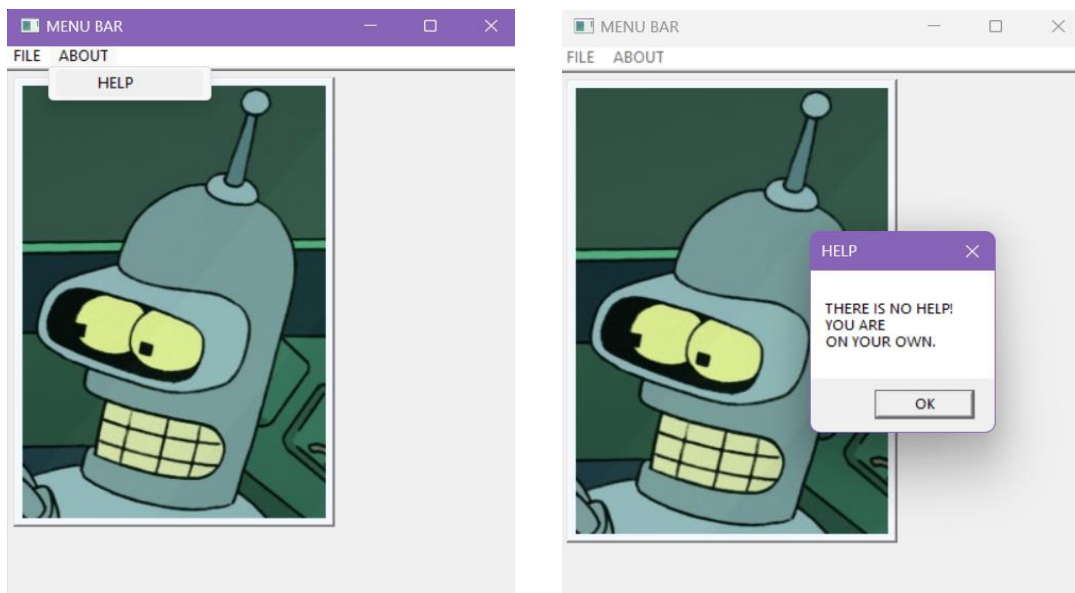
Most professional applications have a menu bar at the top of the main window. Many operations can be performed from this menu bar. Using buttons in applications with many settings or operations takes up a lot of space. Menus can be seen as a way to squeeze a large number of buttons into a narrow space. Below is an application with a menu bar. In this application, there are only two categories in the menu bar: "FILE" and "ABOUT". When we click on the File option in the menu, another option called “OPEN” appears.



When we press the Open option, the file opening menu that we learned in section 2.6 appears as shown below.



After uploading the Bender.jpg image, when we click on the "ABOUT" option in the menu bar, the option(s) under it will appear. In this example, there is only an option called "HELP". When we click on this option, the message box shown on the right below appears.



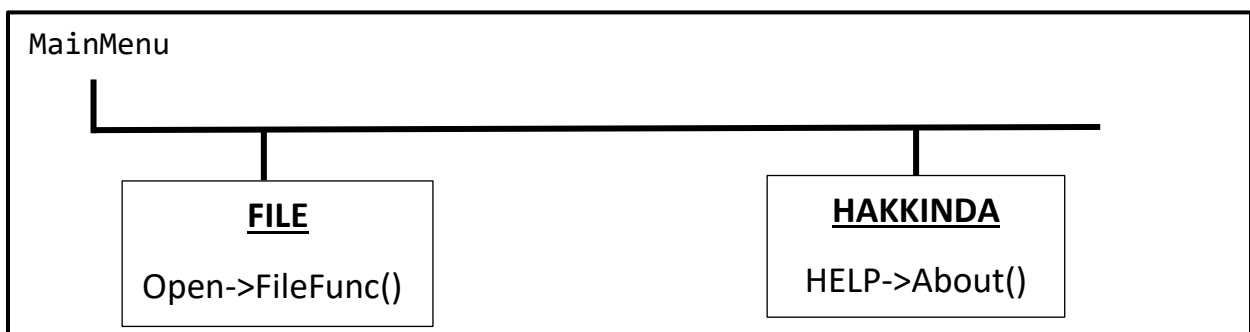
Now let's examine the source code of this application. We see that a picture frame is created in the usual way within the `ICGUI_main()` function. Immediately afterwards, we encounter a new variable type: `HMENU`. From the first letter of this variable type, we can guess that it is a handle belonging to the Windows operating system. We see that three different handles of this type were created. These are `MainMenu`, `FileMenu` and `AboutMenu`. We can guess that these three in practice

correspond to the menu bar itself, the "FILE" and "ABOUT" options in the menu bar, respectively.

```
void ICGUI_main()
{
    FRM = ICG_FramePanel(5, 5, 250, 250);
    HMENU MainMenu, FileMenu, AboutMenu;
    MainMenu = CreateMenu();
    FileMenu = CreatePopupMenu();
    AboutMenu = CreatePopupMenu();
    ICG_AppendMenuItem(FileMenu, "OPEN", FileFunc);
    ICG_AppendMenuItem(AboutMenu, "HELP", Info);
    AppendMenu(MainMenu, MF_POPUP, (UINT_PTR) FileMenu, "FILE");
    AppendMenu(MainMenu, MF_POPUP, (UINT_PTR) AboutMenu, "ABOUT");
    ICG_SetMenu(MainMenu);
}
```

Then, we connect the menu bar handle to the menu bar created with the CreateMenu() function, one of the Win32 API's own functions. As for the other two options, we first create sub-options called "OPEN" and "HELP", give the names of the functions we want to run when they are selected (ICG_AppendMenuItem()), and then connect them to the sub-menus created with Win32 functions called CreatePopupMenu(). So we prepare a tree structure as follows:

Finally, we select the MainMenu, which we prepared with the ICG_SetMenu() function, as the menu of our application. All we need to do next is write the FileSec() and Info() functions:



```
#include "icb_gui.h"

int FRM;

void ICGUI_Create()
{
    ICG_MWSize(420, 500);
    ICG_MWTitle("MENU BAR");
}

void FileFunc()
{
    ICBYTES path, image;
    ReadImage(OpenFileMenu(path, "JPEG\0*.JPG\0"), image);
    DisplayImage(FRM, image);
}

void Info()
{
    MessageBox(ICG_GetMainWindow(), "THERE IS NO HELP!\nYOU ARE\n\ ON\nYOUR OWN.", "HELP", MB_OK);
}
```

FileFunc() opens a file selection window as shown in section 2.6, loads the selected image into a matrix called "image" and then displays this image in the frame. Info() creates an incomplete message box by calling MessageBox, one of the functions of the Win32 API. If you want to learn the details of this function, you can refer to the Visual C++ documentation.

3. Matrix Usage

One of the most important structures that the ICBYTES fluid can transform into is the matrix. In addition to having almost the same features as the array structures existing in C/C++, matrices can hold many different data such as images, sounds and 3-dimensional models. Two different matrices are allowed in the student version of the ICBYTES library. These are the “plain matrix” and the “image matrix.” There is no difference between the plain matrix and the image matrix for the user. Both can be subjected to the same processes. However, the way these two matrices are created is different. For example, while the plain matrix is created by the following methods,

- `ICBYTES A = {{1,2,3},{4,5,6},{7,8,9}};`
(Only `double` or `int` type is allowed for this type of declaration.)
- `ICBYTES A;`
`CreateMatrix(A, 250, 250, 3, ICB_UCHAR);`
- `ICBYTES A;`
`A=5.4;`
- `ICBYTES A;`
`A=“you can write here”;`

image matrix can be created in only one way:

- `ICBYTES A;`
`CreateImage(A, 250, 250, 3, ICB_UCHAR);`
`CreateFlippedImage(A, 250, 250, 3, ICB_UCHAR);` //shown flipped

The only difference between these two types from the user's perspective is that the image matrix can be displayed faster on the screen. However, the image matrix consumes more memory at the expense of this speed. If the matrix will be used only for calculations or display once, the "plain matrix" type should be selected. However, if the matrix will be printed repeatedly as a picture on the screen (20-30 times per second), for example images coming from a video source or an animation, then the "picture matrix" should be selected.

3.1. Matrix Methods

Matrix methods are functions that depend on a matrix. These functions are called by placing a dot after the name of the fluid carrying the matrix. For example:

ICBYTES A = {{1,2,3},{4,5,6}}; $\rightarrow$ $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

After defining a matrix of the form,

- A.X() gives us the length of the X dimension of matrix A, which in this example is 3.
- A.Y() gives us the length of dimension Y of matrix A, which in this example is 2.
- A.Z() gives us the length of the Z (3rd) dimension of matrix A, which in this example is 1.
- A.W() gives us the length of the W (4th) dimension of matrix A, which in this example is 1.

3.2. Matrix Operations

3.2.1. Assignment

The most important matrix operation is the assignment operation. The result of the assignment will be different depending on whether the matrix is defined or not. For example:

```
ICBYTES A;  
A=5;
```

After the expression, a matrix consisting of a single integer element is created in fluid A. However,

```
ICBYTES A;  
CreateMatrix(A, 10, ICB_INT);
```

The expression creates a vector (or array) with X dimension 10 and Y dimension 1, that is, 10 elements. After this:

```
A=5;
```

Equating all 10 elements in this vector equals 5. On the other hand,

```
A=7.89;
```

If we use the above expression, it imitates the behavior of the C/C++ language and makes all elements of the vector equal to 7, which is the integer part of 7.89.

If both sides of the equation have ICBYTES fluids, the copied fluid is transferred to the copied one in exactly the same way:

```
B=A;
```

In this case, B becomes the same as A, regardless of what type of fluid it was.

3.2.2. Matrix Variable Types

Matrix variable types and functions used to access their elements are shown in the table below.

ICBYTES Code Name	Visual C++ Equivalent	Bit Length	Acess
ICB_CHAR	char	8	A.C()
ICB_UCHAR	unsigned char (Byte)	8	A.B()
ICB_SHORT	short	16	A.S()
ICB_USHORT	unsigned short	16	A.u()
ICB_INT	int (long, int32_t)	32	A.I()
ICB_UINT	unsigned int (uns. long)	32	A.U()
ICB_LONG	long long (int64_t)	64	A.L()
ICB_ULONG	unsigned long long (uint64_t)	64	A.O()
ICB_FLOAT	float	32	A.F()
ICB_DOUBLE	double	64	A.D()

3.2.3. Addition and Subtraction

To add or subtract a matrix with any number or another matrix, only “+=” and “-=” operators are defined in the ICBYTES student version. When the matrix or number types on both sides of these operators are not the same, the operation is performed according to the rules of the C/C++ language. So if two matrices are to be added,

```
A+=B;
```

is written. Matrix B is added with A and written into A. Matrices A and B must have the same dimensions. That is, if matrix A is 5x8, matrix B must also be 5x8. However, their types do not have to be the same. For example, if matrix A is ICB_INT and matrix B is ICB_FLOAT, the integer parts of the elements in B are added to the elements of A, just like in the C language. The type of A does not change. When a matrix is added or subtracted by a number, this operation is applied to all elements of the matrix. The following statement,

```
A-=7;
```

causes the number 7 to be subtracted from all elements of A.

3.2.4. Multiplication and Division

In the student version of ICBYTES, multiplication and division are defined with the `*=` and `/=` operators and only for basic variable types (char, int, float, etc.).

3.2.5. Comparison

You can check whether two matrices are equal with the `"=="` symbol. In other words, you can construct expressions of the form `if(A==B){...}`.

3.2.6. Matrix Inverse and Determinant

`"inv()"` function is used for matrix inverse:

```
bool inv(i, o)
```

The matrix to be inverted is "i" and the result is in matrix "o". This function returns true if the matrix has an inverse. Returns false if the matrix has no inverse.

For the determinant, the `"determinant(i)"` function is used. This function returns a number of type `"double"`.

3.2.7. Transpose

The `A.t(B) →` command places the transpose of matrix B into matrix A. Matrices A and B are of the same type.

3.2.8. Dot product

`C.dot(A,B) →` command puts A.B dot product inside the matrix C.

3.2.9. Cross Product

`C.cross(A,B) →` command puts $A \times B$ cross product inside the matrix C.

3.3. Other Matrix Functions

The table below lists the functions that perform non-mathematical operations on the matrix..

Function name	Function Description
<code>bool Convert2ImageMatrix(i);</code>	It turns the plain matrix into a picture matrix.
<code>bool ConvertType(m,type);</code>	Changes the variable type while preserving the matrix content. (e.g. int → float)
<code>bool IsFloating(i)</code>	Tells whether matrix i is floating point or integer.
<code>bool IsMatrix(i)</code>	It tells whether i carries a matrix or not.

<code>bool AreDimsEqual(i, j)</code>	Tells whether the dimensions of matrices i and j are the same.
<code>bool AreEqualImage(i, j)</code>	Returns true if images i and j are the same size and bit depth.
<code>bool FlipY(source, target)</code>	Inverts the source matrix vertically and writes it to the target matrix.
<code>void Free(M)</code>	Deletes matrix M and frees up the memory space it consumes.
<code>double Sum(M)</code>	Returns sum of all elements of matrix M.
<code>void SumX(A,B)</code>	Returns sum of all elements of each row of matrix M as the vector B.
<code>void SumY(A,B)</code>	Returns sum of all elements of each column of matrix M as the vector B.
<code>double Max(M)</code>	Returns the greatest element of matrix M.
<code>void MaxX(A,B)</code>	Returns greatest of all elements of each row of matrix M as the vector B.
<code>void MinX(A,B)</code>	Returns smallest of all elements of each row of matrix M as the vector B.
<code>double Min(M)</code>	Returns the smallest element of matrix M.
<code>void MaxY(A,B)</code>	Returns greatest of all elements of each column of matrix M as the vector B.
<code>void MinY(A,B)</code>	Returns smallest of all elements of each column of matrix M as the vector B.

When using these listed functions, the user only needs to create the fluid; There is no obligation to create the target matrix. If the target matrix is not large enough, the function makes the target (result) matrix large enough for the result.

3.4. Special Matrices

The most frequently needed special matrix is the identity matrix. An identity matrix is a square matrix with all diagonal elements from upper left to lower right are one. Rest of its elements are zero. In order to create one, you can use the command:

```
IdentityMatrix(matrix, size, type)
```

In order to create a matrix whose elements are increasing one by one starting from zero at the upper left corner, you can use:

```
IncreasingMatrix(matrix, x,y,z, type)
```

Statistical applications frequently need matrices whose elements are random numbers between (0,1). You can use:

Uniform distribution $\rightarrow$ `RandomUniform(mat, x,y,z)`

Gauss distribution $\rightarrow$ `RandomNormal(mat, x,y,z)`

Finally, to create magic matrices whose sum of rows or columns are always equal:

`Magic(mat,3)` $\rightarrow$ $\text{mat} = \begin{bmatrix} 2 & 9 & 4 \\ 7 & 5 & 3 \\ 6 & 1 & 8 \end{bmatrix}$

3.5. Error Diagnosis

One of the most important features of the ICBYTES library is that it has extremely intelligent mechanisms that allows you to identify programming mistakes which do not occur at compile time, but only at run time. It will allow you to find out where the error originates.

C++ language is a very powerful language. In fact, among computer languages, it is the most powerful language second to the machine language. What we mean by a powerful language is that it allows the programmer to do anything he/she wants to do. In addition to being powerful, C++ is also the fastest language after machine language. What we mean by speed is the running speed of the machine code it produces, that is, .EXE files. Due to all these, the C/C++ language has always maintained its place among the top five most used languages in the world since its creation in 1972. Because of this popularity, even Java and C# languages are built on the C++ language. It would not be an exaggeration to say that the C++ language structure is the most used language structure, as even if these three languages fall forward or behind in different years, one of them always maintains its place in the top three.

Despite its popularity, C/C++ is a difficult language to learn and use even after learning it. The main reason for this is that it is designed to make the code it produces the fastest and the freedom it gives to the user. In C++, you can access any memory area you want as long as the operating system allows it. You can even leave your program's memory area and access another program's memory area.

This was possible for DOS and Windows 98 operating systems. As of Windows 2000, these permissions have been removed. The C/C++ compiler does not check whether the pointers and array indexes you use are correct. So, let's say you wrote a code like this:

```
int i[10];  
i[-1]=5;  
i[23]=7;
```

In this code, you create an index with 10 elements and then write numbers to its negative one, -1st and 23rd elements. In C/C++ language, valid indexes in an index with 10 elements are between 0 and 9. But the C/C++ compiler compiles this code without any objection. What happens next? When you run the program, the program crashes. In other words, the program ends abruptly. Your operating system may or may not tell you why this is the case with a warning message. Even if it does, this error message often does not contain any information that will help you fix this problem.

Let's say you are a company and you wrote an inventory program for a customer. Your user has 100 shelves, and you used an index with 100 elements to keep memory of how many products are on each shelf. When the user buys a product from the shelf, he enters the shelf number into the program, and the program reduces the number of products on that shelf by one. If the shelf number entered by the user is in a variable called "shelf_no", simply let this program code be as follows:

```
int shelf[100];  
shelf[shelf_no]--;
```

Most of the time your program runs just fine. However, if the user presses the key "1" accidentally and writes 111 while trying to write the 11th shelf, your program will crash. Or at best it alters the value of another, unrelated variable in your program. Many languages other than C/C++ check this at compile time, if possible, and even while running, and give a warning message. But C/C++ doesn't do this. The reason is to create the fastest .EXE code possible. For a language to be able to perform such checks, it is forced to run boundary-checking code for every line you type that deals with arrays. So in another language "shelf[shelf_no]--;" When you type, the compiler adds a piece of code like this:

```
if(shelf_no >=0 || shelf_no <=99)  
    shelf [shelf_no]--;  
else Error Message ... counter fail safe code etc.
```

If we assume that you read and write each pixel once while processing a three-megapixel image, this if-else structure would have to be executed 6,000,000 times. You will spend 3 to 5 times the extra time you need to process the image. That's why no company uses a language other than C/C++ in projects such as video processing or computer vision that require real-time image processing applications. Obviously, we are talking about sane companies here. Not one of those companies that go bankrupt after two years.

The issue here is not just time. Suppose you built a moving robot. This robot needs to see objects around it with a camera and avoid hitting them. If you used a language that performs the above operations, you will spend more energy because you perform more operations. This means that you will use larger batteries on your robot. You will use larger and more expensive motors to transport larger batteries. This will all add up to your costs. If your rival company was not stupid like you and wrote its code in C++, it would reduce the cost to half of yours and bankrupt you after two years. That's why, in the engineering faculties of (high quality) universities, ways to reduce the cost along with the quality of engineering are taught. By the way, if this were a flying drone rather than a robot, try to imagine how many times the costs would increase.

3.4.1. How to Start Troubleshooting?

Let's assume we have a button function like the one below:

```
void buttonfunc()
{
    ICBYTES a(1);
    CreateMatrix(a, 5, 5, ICB_INT);
    a.I(0, 0) = 5;
    a.C(2, 3) = 4;
}
```

In this function, a fluid named "a" is created. However, during creation, something is done that we have not seen before. There are parentheses next to it and "one" (1) is written inside. When a fluid is created this way, it means that we assign an identification number to the fluid "a". We'll find out why in a moment.

Then, fluid "a" is transformed into a 5x5 integer matrix. Then, 5 is written into the (0,0)th element of the "a" matrix. But there is a problem here. Matrix indexes start from 1. Therefore, if we try to reach the zeroth element of a matrix, it would be like accessing the -1th element of an array. When we come to the next line, we see that an attempt is made to access the integer matrix a as if it were a char. At best, this will reach the wrong element. The program may crash when you run this code. The worst part is about all this is that crashing is one of the better

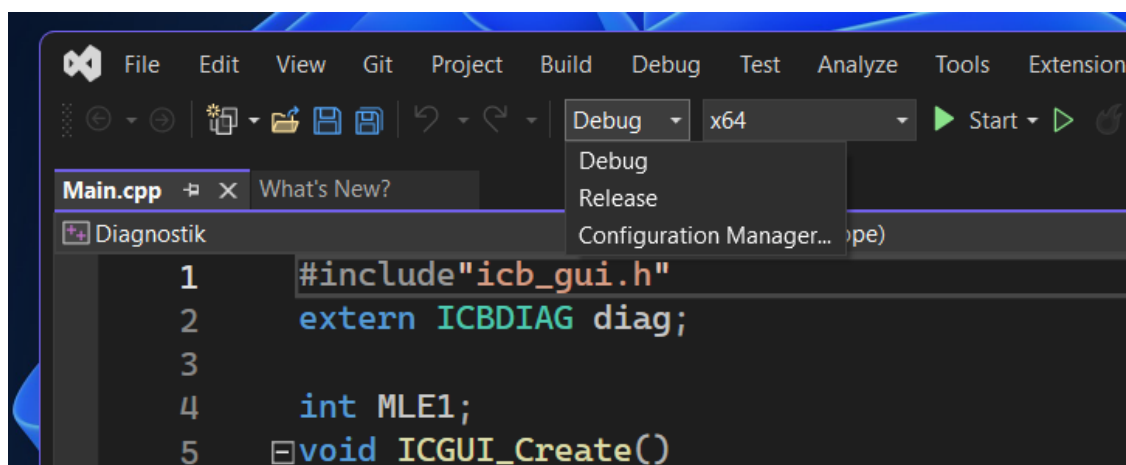
alternatives. Because if the program crashes, at least you know there is a **bug** in your program. If the program does not crash, it means it will make incorrect calculations. Imagine if this were an accounting program, your accounting calculations would be wrong. If this was the image processing function of a tomography device, it would produce tomography images showing cancer in a cancer-free patient. Both situations will make the programmer responsible. Fortunately, the ICBYTES library has extremely clever diagnostic and debugging mechanisms that uncover such errors. To activate these mechanisms, we need to make some minor changes at the beginning of our program. First, we need to define the object that writes debug messages at the top:

```
#include "icb_gui.h"
extern ICBDIAG diag;
```

In the second line of our program, we tell the compiler that we want to use a (diagnostic) object named “diag” created in the library. Now this object needs a multi-line text box to write its messages, and we provide this in the ICGUI_main() function:

```
void ICGUI_main()
{
    ICG_Button(5, 5, 120, 25, "Buton1", buttonfunc);
    MLE1 = ICG_MLEditSunken(5, 50, 250, 150, "", SCROLLBAR_V);
    diag.SetOutput(ICG_GetHWND(MLE1));
}
```

Finally, we need to compile our program in “DEBUG” mode. To do this, we select DEBUG mode from the Visual Studio menu as seen below.

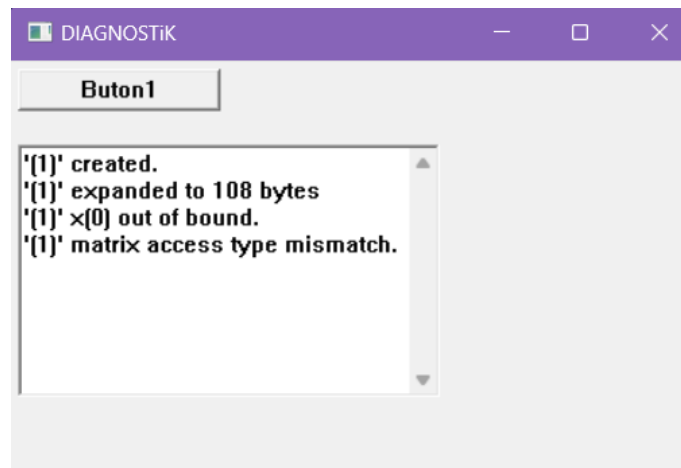


After doing all this, when we compile and run the program, the following screenshot will appear when you press the Button.

This program has mutated to inform us about every important phase of the ICBYTES variable and its misuses. on the first line:

'(1)' created.

It says that the ICBYTES fluid with the number "1" was created. Remember, we assigned fluid "a" an identification number when it was created:



ICBYTES a(1);

This number is currently used by the diagnostic system to express which fluid information is being provided to us. Now let's look at the second of these messages:

(1)' expanded to 108 bytes

It is reported that the amount of memory used by our fluid has expanded to 108 bytes. The reason for this is that we created a 5x5 integer matrix in the fluid "a". Integers use 4 bytes. Therefore, 25 integers will use 100 bytes. Our fluid must be using the extra 8 bytes itself. In the next line:

'(1)' x(0) out of bound.

It says that we are trying to reach the zeroth element of the X dimension of our matrix and this is incorrect. If you remember, in the third line of the button function "a.I(0, 0) = 5;" we wrote. It informs us that this is incorrect access. In the last line of the button function, "a.C(2, 3) = 4;" We tried to access an integer matrix as if it were characters by writing He also warns us about this:

'(1)' matrix access type mismatch.

This is how the error diagnosis and debugging feature works, which makes the ICBYES library different from other libraries and provides ease of use and reliability.

3.6. Matrix Properties

The matrices carried by ICBYTES fluid have many properties. Programmers will need to achieve these features when working with these matrices. Many functions have been added to the library to make this easier.

3.5.1. Size Inquiry

There are three different methods to query Matrix Dimensions. Since matrices in the ICBYTES library can have up to four dimensions, the first two of them return the dimensions of the matrix (X,Y,Z,W) as an array or another matrix.:

```
long long* SizeinArray(ICBYTES& i);
ICBYTES & size(ICBYTES& i);
```

The first of these returns the magnitude of those four dimensions in a long long[4] index. The second one returns a 4-element vector of type long long in the ICBYTES fluid. We can better understand the use of these functions by writing a program like the one below.

```
void PrintMatrix()
{
    ICBYTES A = { {1,2,3},{4,5,6},{7,8,9} };
    DisplayMatrix(MLE, A);
    ICBYTES B = size(A);
    DisplayMatrix(MLE, B);
    long long* t = SizeinArray(A);
    ICG_printf(MLE, "A matrix dimensions:(%d , %d)", t[0], t[1]);
}
```

MLE here is the handle for a multi-line edit box. The output that this function will print on the screen will be as follows:

```
(INT 3 x 3)
1 2 3
4 5 6
7 8 9
```

```
(INT64 4 x 1)
3 3 1 1
```

```
A matrix dimensions:(3 , 3)
```

The point to note here is that when we create a 3x3 matrix, the size of the other dimensions is one, not zero.

The third function we use to query the dimension is the `GetMatrixDims()` function. This function returns one of the integers defined below:

<code>ICB_HAS_X</code>	1	(has only X dimension)
<code>ICB_HAS_Y</code>	2	
<code>ICB_HAS_XY</code>	3	(has both X and Y dimensions)
<code>ICB_HAS_Z</code>	4	
<code>ICB_HAS_XZ</code>	5	
<code>ICB_HAS_YZ</code>	6	
<code>ICB_HAS_XYZ</code>	7	(has X and Y and Z dimensions)
<code>ICB_HAS_W</code>	8	
<code>ICB_HAS_XW</code>	9	(has X and W dimension)

...

<code>ICB_HAS_XYZW</code>	15	(has X and Y and Z dimensions)
---------------------------	----	--------------------------------

Apart from these, other methods that perform matrix query are given in Appendix-2.

3.7. Accessing Matrix of Unknown Type

We previously mentioned that one of the worst features of OpenCV library is the difficulty in accessing matrix elements. Unfortunately, this problem is magnified due to poor documentation. If you do not know the type of matrix that a function returns to you, it is recommended that you first convert that matrix to a double matrix when you want to change that matrix or print the values to the screen. This causes difficulty for the programmer and slowdown for the program. It is impossible to write to a matrix of unknown type. However, in the ICBYTES library, it is very easy to read the elements of a matrix whose type you do not know. All you have to do is open parentheses next to the matrix and enter the index of the element you want to reach. So, let's say you have a matrix called A, whose type you do not know, and you want to assign the (15,4) element to a variable or print it:

```
double t=A(15,4);
```

Writing the above code is enough. There are two things you should pay attention to here:

- This way you can just read the element of the matrix. **YOU CANNOT WRITE INTO IT.**
- Regardless of the type of matrix, the returned value is always in `double`.

Whether the type of the matrix is int or char, you will receive the value in double, so if this type of variable is not useful to you, you can typecast:

```
int t=(int)A(15,4);
```

If you want to write into the matrix element, then you should use the `Set()` function. For example, let's say we want to set the (15,4) element of A to -1.5:

```
Set(15,4,A,-1.5);
```

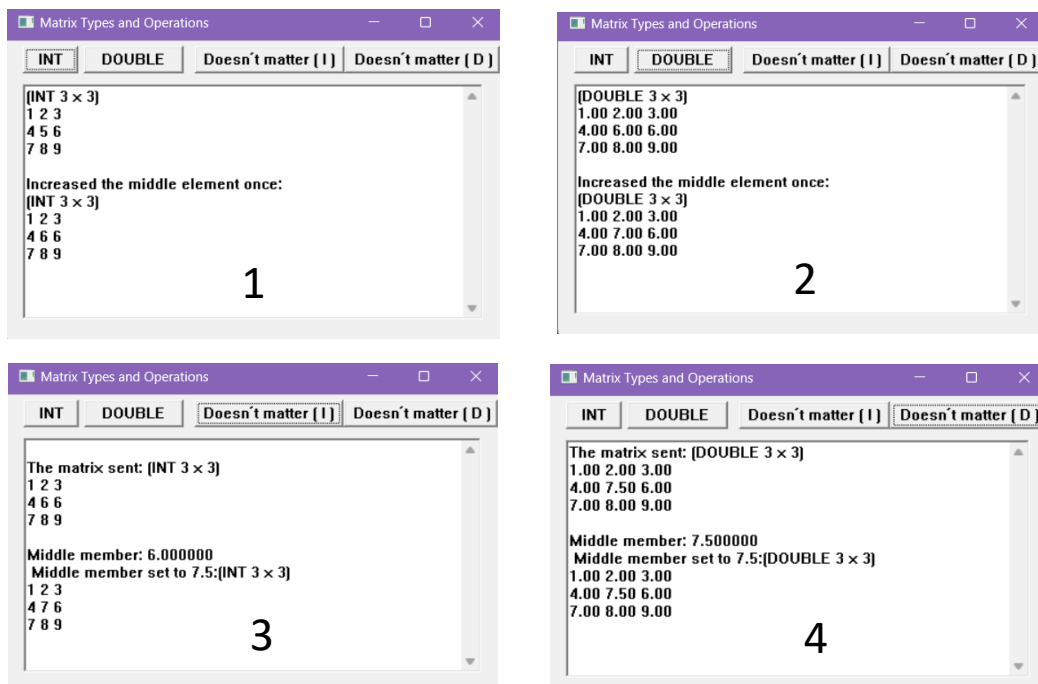
The command performs this operation. The last entry of the Set() function is the value we want to write. This value can be any type. So, instead of -1.5, we can write an integer (for example, 5). All types defined in C++ (char, short, float...) can be used as the final input. The points we need to pay attention to here are:

- If the type of matrix A is integer and you write double/float to it, only the integer part of the number will be written.
- If the type of the matrix is unsigned or is not large enough to hold the value you type, your number is written to the element of the matrix in accordance with the C++ variable assignment rules. That is, if the type of matrix A is unsigned, the Set() command writes 4.294.967.295 to the (15,4) element of A. For this reason, caution should be exercised.

Below are screenshots of an example program that uses these features. When we press the buttons of this program in order, the following happens:

1. The (**int**) matrix created in Main is printed on the screen, then the element in the middle is accessed with the I.I(2,2) method and incremented by one:
`static ICBYTES I = { {1,2,3},{4,5,6},{7,8,9} };`
2. The (**double**) matrix created in Main is printed on the screen, then the element in the middle is accessed with the D.D(2,2) method and incremented by one:
`static ICBYTES D = { {1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0} };`
3. If created inside the main():
`static ICBYTES I = { {1,2,3},{4,5,6},{7,8,9} };`

It is sent to the doesnt_matter() function created in Main, and the element in the middle is accessed with the U(2,2) method and the value is printed on the



screen as 6.000000, even though the value is actually 6 (as it was increased by one in Step 1); Then, instead of this value, an attempt is made to write 7.5 with the Set(2,2,U,7.5) method, but when the matrix is printed on the screen, the middle element is seen to be 7..

4. If created inside the main():

```
static ICBYTES D = { {1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0} };
```

It is sent to the (doesn't matter) function created in Main, and the element in the middle is accessed with the U(2,2) method, and although the value is actually (double) 6.00 (since it was increased by one in Step 2), it is printed on the screen as 6.000000; Then, instead of this value, 7.5 is successfully written with the Set(2,2,U,7.5) method..

Below is the source code of this program:

```
#include "icb_gui.h"

int MLE;

void ICGUI_Create()
{
    ICG_MWTitle("Matrix Types and Operations");
    ICG_MWSize(470, 320);
}

void Int_proc(void *icb)
{
    ICBYTES* pi = (ICBYTES*)icb;
    DisplayMatrix(MLE, *pi);
    ICG_printf(MLE, "Increased the middle element once:\n");
    pi->I(2, 2)++;
    DisplayMatrix(MLE, *pi);
}

void Double_proc(void* icb)
{
    ICBYTES* pi = (ICBYTES*)icb;
    DisplayMatrix(MLE, *pi);
    ICG_printf(MLE, "Increased the middle element once:\n");
    pi->D(2, 2)++;
    DisplayMatrix(MLE, *pi);
}

ICG_printf(MLE, " Middle member set to 7.5:");
DisplayMatrix(MLE, U);
}
```

```

void DoesntMatter(void* icb)
{
    ICBYTES& U = *(ICBYTES*)icb;
    ICG_printf(MLE, "The matrix sent: ");
    DisplayMatrix(MLE, U);
    ICG_printf(MLE, "Middle member: %f\n", U(2,2));
    Set(2, 2, U, 7.5);
    ICG_printf(MLE, " Middle member set to 7.5:");
    DisplayMatrix(MLE, U);
}

void ICGUI_main()
{
    //all must be integer
    static ICBYTES I = { {1,2,3},{4,5,6},{7,8,9} };
    ICG_Button(15, 5, 50, 25, "INT", Int_proc,(void*)&I);
    //all must be double
    static ICBYTES D = { {1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0} };
    ICG_Button(70, 5, 90, 25, "DOUBLE", Double_proc,(void*)&D);
    ICG_Button(170,5,135, 25, "Doesn't matter ( I )",
DoesntMatter,(void*)&I);
    ICG_Button(305,5,140,25, " Doesn't matter ( D )",
DoesntMatter,(void*)&D);
    MLE = ICG_MLEditSunken(15, 40, 415, 215, "", SCROLLBAR_V);
}

```

Finally, you can also access a matrix of unknown type by converting it to a type of your choice. For example, to convert to integer:

```
ConvertType(M, ICB_INT);
```

3.8. Character Access

Regardless of its type, if you want to access the memory area of a matrix as bytes, all you have to do is use square brackets:

```
A[5]='C';
```

The command writes the letter 'C', that is, the value 67, in the 6th byte of the A matrix. If your matrix is already a vector of characters (i.e. char or unsigned char), you can use it as a character index like this. Here are the points you should pay attention to:

- C++ index format is used, not a matrix. That is, index values start at zero and go up to one minus the size of the matrix in bytes.
- Since the y coordinate is opposite to the matrix coordinates on PC compatible machines, if you are using a matrix instead of a vector, it starts accessing from the bottom line. Since there is only one line in the vector, this does not cause a problem. You can use it just like C++ arrays.

4. Graphical Interface Styles

In the previous sections, we introduced the basic graphics features of the I-See-Bytes (ICBYTES) library. Although many different applications can be written with these features, we need much different and more diverse widgets and styles to create professional-looking graphical interfaces that can perform complex tasks. In this section, we will take a deeper look at I-See-GUI (ICGUI), which is the sub-library of the I-See-Bytes library containing graphical interface functions.

4.1. Picture Button

In this example, we will see how to create a button box that contains an image instead of text. In the code example below,

```
#include "icb_gui.h"

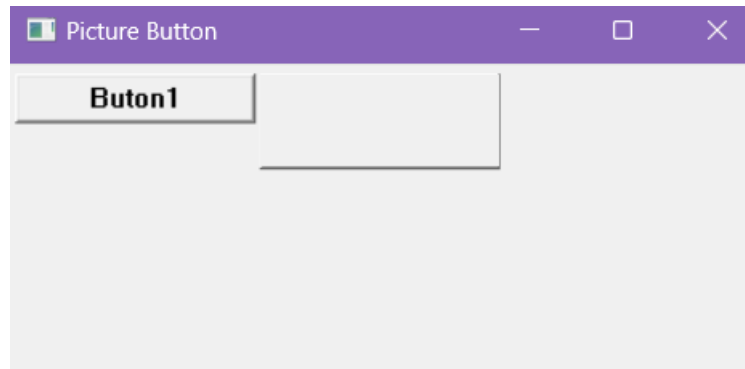
int BTN1;
ICBYTES image;

void buttonfunc1();
void buttonfunc2(){}

void ICGUI_Create()
{
    ICG_MWSize(400, 200);
    ICG_MWTitle("Picture Button");
}

void ICGUI_main()
{
    ICG_Button(5, 5, 120, 25, "Buton1", buttonfunc1);
    BTN1 = ICG_BitmapButton(127, 5, 120, 48, buttonfunc2);
    ReadImage("Buton.bmp", image);
}
```

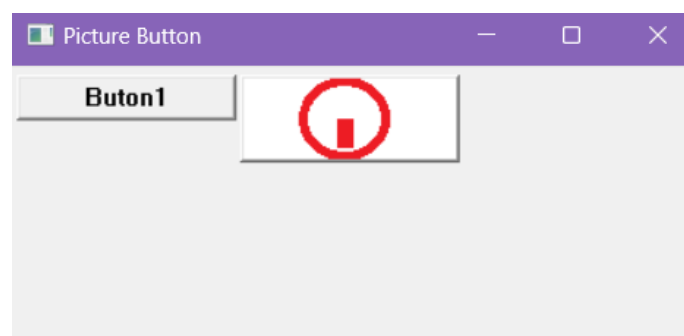
We see a new type of Button creation function in ICG_Main. This function, whose name is "ICG_BitmapButton()", creates a button with dimensions of 120x48. Unlike other button functions, this function does not take a character index. As a result, the button created by this function is a button without text as shown below.



Another command that draws our attention in the ICG_Main function is "ReadImage()." It loads a file called "Button.bmp", which we created with the paint program in Windows, to be loaded as a matrix into the fluid named "image". Here, only the name of the bitmap file is given without giving its file path. This means that the Button.bmp file must be located in the same folder as the .EXE created by this project. If the project is executed through Visual Studio, the Button.bmp file must be in the same folder as the project file.

When we look at the button1 on the left, we see that it is a button we are familiar with, and when we examine its function, we see that it calls the "SetButtonBitmap()" function, that sets the image of our second button using the image fluid we loaded into the Button.bmp file, which is currently empty. When button1 is pressed, the image of our application changes to the following:

```
void buttonfunc1()
{
    SetButtonBitmap(BTN1, image);
}
```



In a normal application, instead of pressing a button, you want the image to be placed on the button as soon as the application opens. To make this happen, simply change the ICG_Main function as follows.

```
void ICGUI_main()
{
    int BTN1;
    static ICBYTES image;
    ICG_Button(5, 5, 120, 25, "Buton1", buttonfunc1);
    BTN1 = ICG_BitmapButton(127, 5, 120, 48, buttonfunc2);
    ICB_ReadImage("Buton.bmp", image);
    SetButtonBitmap(BTN1, resim);
}
```

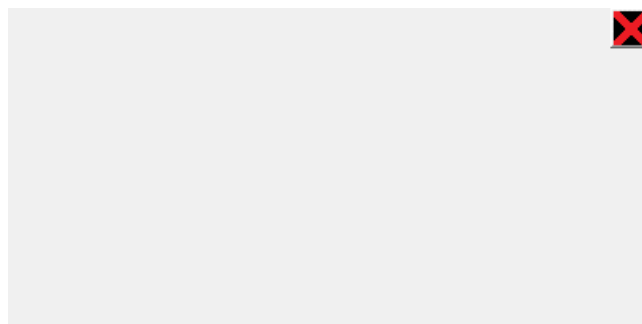
The applications where we would most want to use these types of image buttons are those with windows without a title bar. For example, the code below creates a tiny black button with a red cross on it that lets you exit the program, eliminating the title bar. You can choose these types of windows when you want to use the entire screen.

```
#include "icb_gui.h"

void buttonfunc1(){exit(0);}

void ICGUI_Create()
{
    ICG_MW_RemoveTitleBar();
    ICG_MWSize(400, 200);
}

void ICGUI_main()
{
    int BTN1;
    static ICBYTES image;
    BTN1 = ICG_BitmapButton(373, 0, 25, 25, buttonfunc1);
    ICB_ReadImage("exit.bmp", image);
    SetButtonBitmap(BTN1, image);
}
```



4.2. Multiline Edit Window Types

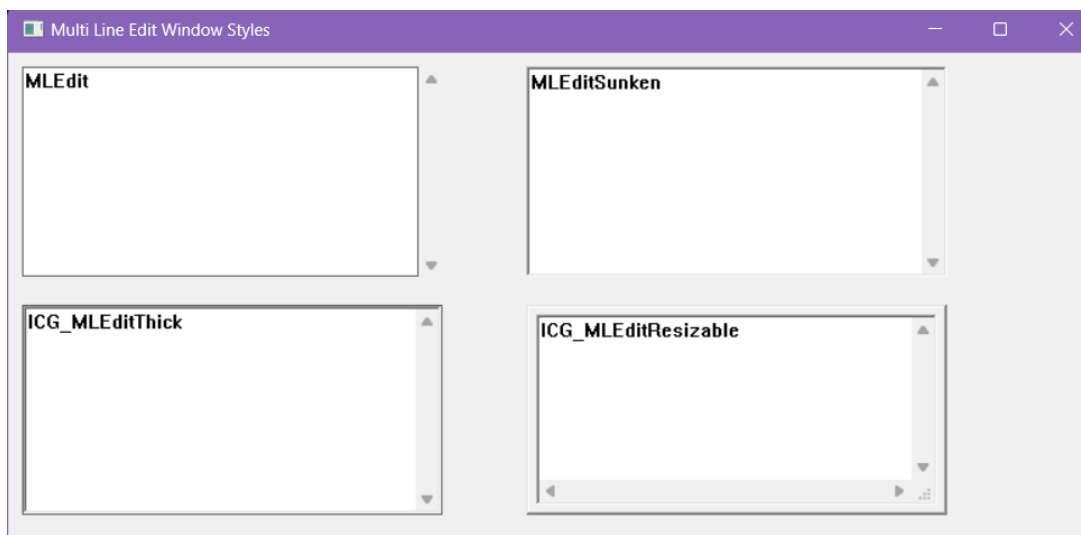
There are four types of multi-line text boxes (Multi Line (ML) Edit Widget) in the I-See-GUI student version. The code example below shows how to create these.

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWSize(800, 400);
    ICG_MWTitle("Multi Line Edit Window Styles");
}

void ICGUI_main()
{
    ICG_MLEdit(10, 10, 300, 150, "MLEdit", 1);
    ICG_MLEditSunken(370, 10, 300, 150, "MLEditSunken",
    SCROLLBAR_V);
    ICG_MLEditThick(10, 180, 300, 150, "ICG_MLEditThick",
    SCROLLBAR_V);
    ICG_MLEditResizable(370, 180, 300, 150, "ICG_MLEditResizable",
    SCROLLBAR_HV);
}
```

Below is a screenshot of the application window created by this code. The simplest version, MLEdit(), is shown in the upper left corner. We have often used MLEditSunken() in previous examples as it is generally the most preferred style. MLEditThick() has a slightly thicker frame. MLEditResizable() is a style whose size can be changed by holding the edges with the mouse cursor. When creating this style, it will be useful to create it with the "SCROLLBAR_HV" option, that is, with both horizontal and vertical scroll columns.



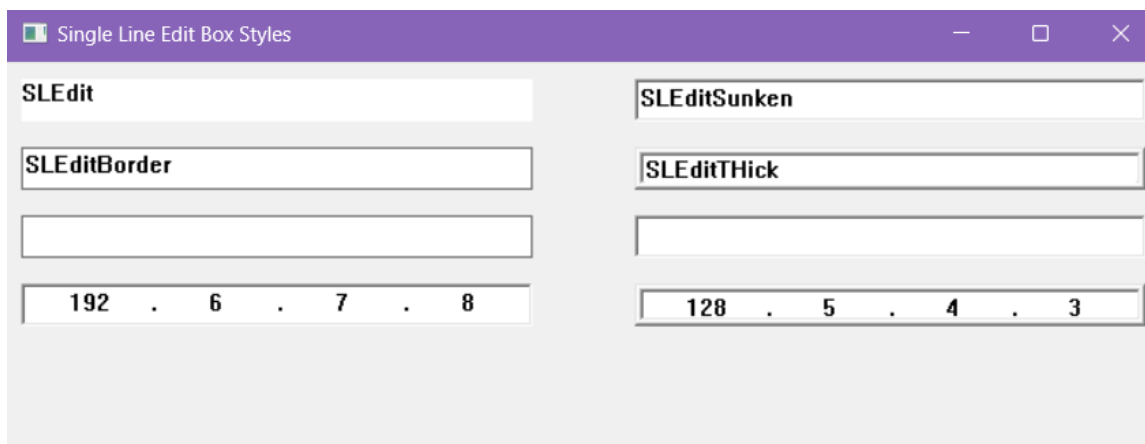
4.3. Single Line Edit Window Types

There are eight types of single-line text box styles, as seen in the code example below and the screenshot of the application. The first four of these are plain text boxes, the next two are password boxes, and the last two are internet address boxes.

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWSize(700, 300);
    ICG_MWTitle("Single Line Edit Box Styles");
}

void ICGUI_main()
{
    ICG_SLEdit(10, 10, 300, 25, "SLEdit");
    ICG_SLEditSunken(370, 10, 300, 25, "SLEditSunken");
    ICG_SLEditBorder(10, 50, 300, 25, "SLEditBorder");
    ICG_SLEditThick(370, 50, 300, 25, "SLEditThick");
    ICG_SLPasswordB(10, 90, 300, 25);
    ICG_SLPasswordSunken(370, 90, 300, 25);
    ICG_IPAddressSunken(10, 130, 300, 25, 0xc0060708);
    ICG_IPAddressThick(370, 130, 300, 25, 0x80050403);
}
```



One thing to note here is that when a thick frame type is used, the tail parts of some letters such as "g" are not visible due to the frame width growing inward. The height of all the boxes here is selected as 25 pixels. While the tail of the letter g can be easily seen in the box at the top right, it is not visible due to the thick frame of the box below it.

Another point to note is that you need to use an "[unsigned int](#)" type variable, not a character string, to write addresses to internet address boxes:

```
ICG_IPAddressSunken(10, 130, 300, 25, 0xc0060708);  
ICG_IPAddressThick(370, 130, 300, 25, 0x80050403);
```

The final parameters of these functions are written using hexadecimal because they are relatively easier to read. In C++, numbers starting with 0x are written in hexadecimal. The decimal equivalent of 0xc0 is 128.

4.4. Picture Frame Styles

There are six types of image frame styles in the I-See-GUI library. Here's an example of the code that creates these styles and an image of the resulting window. There is also the `ICG_Frameless()` method, which cannot be shown here because it has no frame.

```
#include "icb_gui.h"  
  
void ICGUI_Create()  
{  
    ICG_MWSize(730, 300);  
    ICG_MWTitle("Frame Styles");  
}  
  
void ICGUI_main()  
{  
    ICG_FramePanel(10, 10, 100, 100);  
    ICG_FrameThin(120, 10, 100, 100);  
    ICG_FrameMedium(240, 10, 100, 100);  
    ICG_FrameThick(360, 10, 100, 100);  
    ICG_FrameSunken(480, 10, 100, 100);  
    ICG_FrameDeep(600, 10, 100, 100);  
}
```



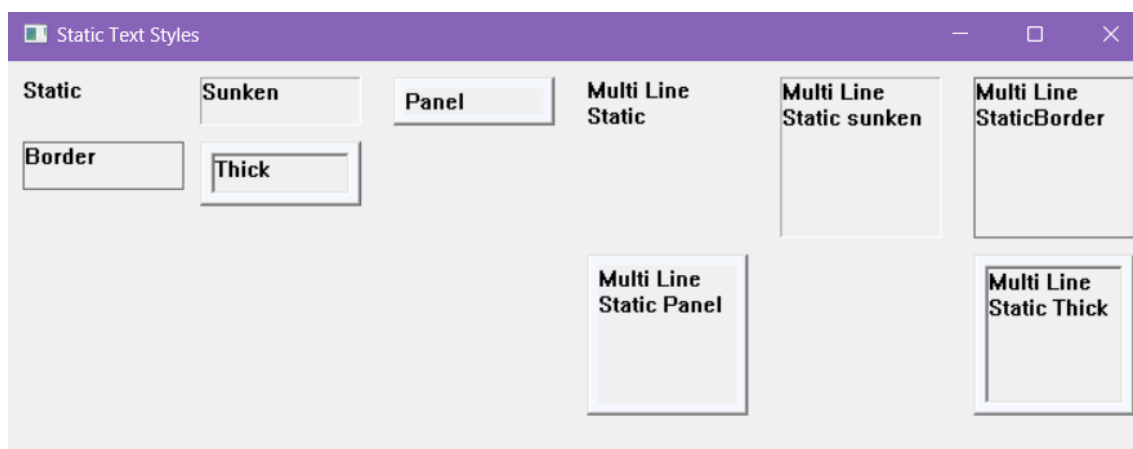
4.5. Static Text Styles

The code that creates different static text styles in the ICGUI library and the output of this code are shown below.

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWSize(730, 300);
    ICG_MWTitle("Static Text Styles");
}

void ICGUI_main()
{
    ICG_Static(10, 10, 100, 30, "Static");
    ICG_StaticBorder(10, 50, 100, 30, "Border");
    ICG_StaticSunken(120, 10, 100, 30, "Sunken");
    ICG_StaticThick(120, 50, 100, 40, "Thick");
    ICG_StaticPanel(240, 10, 100, 30, "Panel");
    ICG_MLStatic(360, 10, 100, 100, "Multi Line Static");
    ICG_MLStaticPanel(360, 120, 100, 100, "Multi Line Static
Panel");
    ICG_MLStaticBorder(600, 10, 100, 100, "Multi Line
StaticBorder");
    ICG_MLStaticSunken(480, 10, 100, 100, "Multi Line Static
sunken");
    ICG_MLStaticThick(600, 120, 100, 100, "Multi Line Static
Thick");
}
```



4.6. Track Bar

Using the trackbar, it is possible to decrease or increase a value using the Mouse.

The function that creates a horizontal trackbar:

```
LEVEL=ICG_TrackBarH(X, Y, Width, Height, MyLevel);
```

The function that creates a vertical trackbar:

```
LEVEL=ICG_TrackBarV(X, Y, Width, Height, MyLevel);
```

The X and Y arguments of this function are the upper left coordinates of the widget. Even if the level handle doesn't move when clicked on it, it shall call the MyLevel(int) function anyway. The (int) argument to this function indicates the position of the lever at that moment. For example, if we use a function like the one shown below:

```
void MyLevel(int k)
{
    ICG_printf(MLE, "%d ", k);
}
```

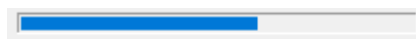
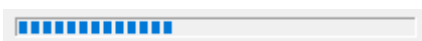
The multiline edit window will display the current position of the lever indicated by a number between 0 and 100. If you wish to access the level information without using that function, you can use the one below:

```
int a=ICG_GetTrackBarPos(LEVEL);
```

4.7. Progress Bar

The following function creates a progress bar which is used to indicate the progress of a process. The last argument defines the type of the bar. The default value (if you don't write anything) is zero which creates a discrete bar made up of little squares (shown on the left). If you set that parameter to one, then it creates the continuous progress bar that is shown on the right.

```
GEL=ICG_ProgressH(x, y, width, height, int type=0);
```



In order to set the step size use: `ICG_SetProgressStep(GEL, 1);`

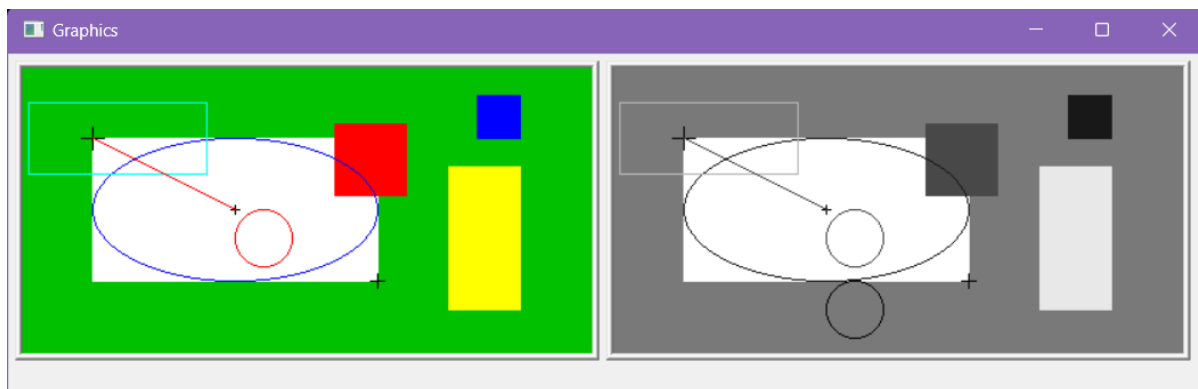
In order to set the min and max range: `ICG_SetProgressRange(GEL, 0, 1000);`

To set the progress level:

```
ICG_SetProgressPos(GEL, 750); //75 progress level.
```

5. Shape Drawing

Some basic capabilities for graphic drawing have been added to the student version of the I-See-Bytes library. Thanks to these abilities, students can learn subjects such as user interface or game programming, which are areas that involve computer graphics, more quickly. The screenshot of the app below shows some of these features.



This example will show you how to draw basic graphic elements: straight line, rectangle, circle and ellipse. While a 32-bit color image is shown on the left side of the application, its black and white version is shown on the right side. Four different functions were used to draw these shapes. The first argument of these functions is of type ICBYTES fluid. The next two arguments are the coordinates of the upper left corner of the shape. Even for the ellipse, these two arguments must be the upper left corner of the rectangle into which it can be drawn. For the line, these are the starting coordinates. These coordinates start from one, not zero, because they are written into a matrix. The functions that draw the white rectangle and the blue ellipse drawn inside it are shown below:

```
FillRect(color, 50, 50, 200, 100, 0xffffffff);
```

```
Ellipse(color, 50, 50, 100, 50, 0xff);
```

Here, the function called FillRect (Filled Rectangle) draws on the fluid carrying the matrix in ICBYTES, whose name is "color." It draws a white rectangle

(0xffffffff: K:255 Y:255 M:255) with upper left corner coordinates x:50 and y:50 and a horizontal width of 200 pixels and a vertical length of 100 pixels. The Ellipse() function draws an ellipse whose upper left corner coordinates are x:50 and y:50, whose horizontal radius is Rx:100 and vertical radius is Ry:50 pixels, and whose color is 0xff (Blue:255). Since the white rectangle is drawn first, the ellipse is drawn on top of it. Otherwise, the rectangle will be drawn over the ellipse and prevent it from being visible. All filled squares and rectangles in the picture were drawn with the FillRect() function, and all ellipses and circles were drawn with the Ellipse() function. To draw a circle, all that is required is for the Rx and Ry parameters to have the same value.

In the picture, the red line and the turquoise hollow rectangle, starting from the upper left corner of the rectangle and ending at its middle point, are respectively:

```
Line(color, 50, 50, 150, 100, 0xff0000);
```

```
Rect(color, 5, 25, 125, 50, 0xffff);
```

It is drawn with its functions. The Line() function needs the coordinates of the start and end points of the line. Rect() takes the same arguments as FillRect().

The C++ code of the application whose screenshot is given above is shown below..

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWTitle("Graphics");
    ICG_MWSize(860, 300);
}

void ICGUI_main()
{
    int FRM1, FRM2;
    static ICBYTES color, grayscale;
    CreateImage(color, 400, 200, ICB_UINT);
    FRM1 = ICG_FrameMedium(5, 5, 400, 200);
    FRM2 = ICG_FrameMedium(420, 5, 400, 200);
    color = 0xc000;
    FillRect(color, 50, 50, 200, 100, 0xffffffff);
    FillRect(color, 220, 40, 50, 50, 0xff0000);
```

```

FillRect(color, 300, 70, 50, 100, 0xffff00);
FillRect(color, 320, 20, 30, 30, 0xff);
Ellipse(color, 50, 50, 100, 50, 0xff);
Ellipse(color, 150, 100, 20, 20, 0xff0000);
Rect(color, 5, 25, 125, 50, 0xffff);
MarkPlus(color, 50, 50, 15, 0);
MarkPlus(color, 150, 100, 5, 0);
MarkPlus(color, 250, 150, 10, 0);
Line(color, 50, 50, 150, 100, 0xff0000);
Luma(color, grayscale);
Ellipse(grayscale, 150, 150, 20, 20, 0);
DisplayImage(FRM1, color);
DisplayImage(FRM2, grayscale);
}

```

In the screenshot, apart from the geometric shapes, we see plus (+) signs of different sizes at both ends of the line and in the lower right corner of the white rectangle. A function called `MarkPlus()` was used to create this mark. Apart from the `ICBYTES` fluid, the coordinates of the center of the sign, the size and color tone of the crosshair are given as input to this function, respectively. The `Luma()` function was used to create the grayscale image. When you pay attention, you will see that the only difference between the color image and the grayscale image is a second black circle drawn just below the white rectangle. The purpose of drawing this circle is to show that the drawing functions described here also work on images with 8-bit color resolution.

5.1. Button Icon

In Section 4.1, we drew a button icon using the Paint program that comes with the Windows operating system and then pasted it onto a Bitmap button. In this part, we will perform the same task with shape drawing functions. In the application window screenshot further down below, you will see a button with a slightly more professional design.



The command sequence that creates this icon is shown below:

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWTitle("Button Icon");
    ICG_MWSize(420, 150);
}

void buttonfunc(){}

void ICGUI_main()
{
    int BTN1;
    static ICBYTES buttonimage;
    CreateImage(buttonimage, 120, 50, ICB_UINT);
    buttonimage = 0xff0000;
    BTN1 = ICG_BitmapButton(140, 15, 120, 50, buttonfunc);
    Ellipse(buttonimage, 45, 8, 17, 17, 0xffffffff);
    Ellipse(buttonimage, 46, 9, 16, 16, 0xffffffff);
    Ellipse(buttonimage, 47, 10, 15, 15, 0xffffffff);
    Ellipse(buttonimage, 48, 11, 14, 14, 0xffffffff);
    Line(buttonimage, 61, 1, 61, 25, 0xffffffff);
    Line(buttonimage, 62, 1, 62, 25, 0xffffffff);
    Line(buttonimage, 63, 1, 63, 25, 0xffffffff);
    for(int y=1;y<= buttonimage.Y();y++)
        for (int x = 1; x <= buttonimage.X(); x++)
        {
            buttonimage.U(x, y)-= 0x030000 * y;
        }
    SetButtonBitmap(BTN1, buttonimage);
}
```

In this program, a 32-bit image is created and all pixels are made red:

```
static ICBYTES buttonimage;
CreateImage(buttonimage, 120, 50, ICB_UINT);
buttonimage = 0xff0000;
```

Then, four white circles are drawn inside each other to create a white circle four pixels thick. Then three white lines are drawn side by side from the top of this circle to the middle. In this way, the frequently used on/off icon is created. Then, in order to give the button a sense of depth, 3 is subtracted from the red tone of the pixels in the first row of the matrix, 6 from the second row, 9 from the third row, and so on, and 150 is subtracted from the last row. This causes the red color to gradually darken, creating a shadow effect.

In this example, we drew four white circles inside each other to create a white circle four pixels thick. Unfortunately, this resulted in gaps between the circles. A better method would be to use `FillEllipse()` to first create a white circle, and then inside it create a red circle with a radius four pixels smaller:

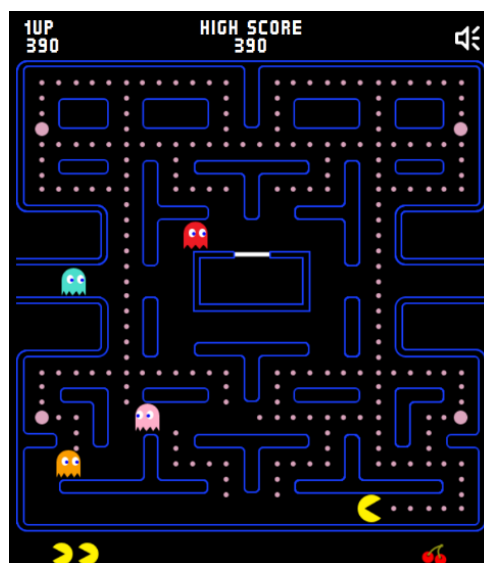
```
FillEllipse(buttonimage, 45, 8, 17, 17, 0xffffffff);  
FillEllipse(buttonimage, 49, 12, 13, 13, 0xff0000);
```

As seen below, there are no gaps in the circle created with this method. This shows us that there is more than one method of drawing shapes and some of these methods give better results than others.



5.2. Complex Graphics

Let's say you want to write a game program and for this you need to draw many geometric shapes on the screen. For example, let's say you want to write Pacman, the famous game of the 80s shown below.



To create a maze like in this game, you need to draw many geometric shapes, and calling a function for each of these shapes will increase the amount of code you will write to hundreds of lines. However, in the code example given below, all you need to do is to have the coordinates of these geometric shapes:

```
#include "icb_gui.h"

ICBYTES labirent = { {270, 50, 20, 80},{210, 110, 140, 20},{270, 170, 20,80},
{210, 230, 140, 20},{270, 410, 20, 80}, {210, 470, 140, 20},{150, 170, 80, 20},
{330, 170, 80, 20},{50, 470, 60, 20},{50, 530, 60, 40},{150, 530, 80, 40},
{150, 230, 20, 80},{150, 350, 20, 140},{150, 410, 80, 20},{450, 470, 60, 20},
{450, 530, 60, 40},{330, 530, 80, 40 },{390, 230, 20, 80},{390, 350, 20, 140},
{330, 410, 80, 20},{50, 50, 180, 20},{150, 50, 20, 80},{330, 50, 180, 20},
{390, 50, 20, 80},{90, 110, 20, 80},{50, 170, 60, 20},{450, 110, 20, 80},
{450, 170, 60, 20} };

ICBYTES cage = { {210, 290, 350, 290},{350, 290, 350, 370},{350,370,300,370},
{300, 370,300,360},{300,360,340,360},{340,360,340,300},{340,300,220,300},
{220,300,220,360},{220,360,260,360},{260,360,260,370},{260,370,210,370},
{210,370,210,290} };

ICBYTES cepheic = { {10,10,550,10},{550,10,550,110},{550,110,510,110},
{510,110,510,130},{510,130,550,130},{550,130,550,230},{550,230,450,230},
{450,230,450,310},{450,310,550,310},{550,310,550,350},{550,350,450,350},
{450,350,450,430},{450,430,550,430},{550,430,550,610},{550,610,290,610},
{290,610,290,530},{290,530,270,530},{270,530,270,610},{270,610,10,610},
{10,610,10,430},{10,430,110,430},{110,430,110,350},{110,350,10,350},
{10,350,10,310},{10,310,110,310},{110,310,110,230},{110,230,10,230},
{10,230,10,130},{10,130,50,130},{50,130,50,110},{50,110,10,110},
{10,110,10,10} };

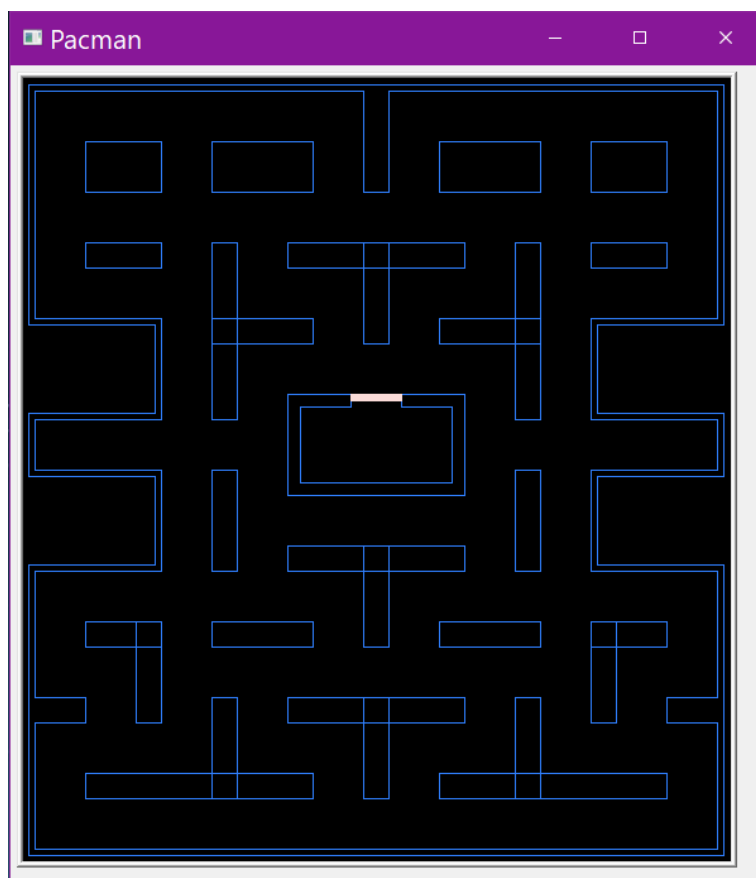
ICBYTES cephedis = { {5,5,555,5},{555,5,555,235},{555,235,455,235},
{455,235,455,305},{455,305,555,305},{555,305,555,355},{555,355,455,355},
{455,355,455,425},{455,425,555,425},{555,425,555,615},{555,615,5,615},
{5,615,5,425},{5,425,105,425},{105,425,105,355},{105,355,5,355},
{5,355,5,305},{5,305,105,305},{105,305,105,235},{105,235,5,235},
{5,235,5,5} };

void ICGUI_Create()
{
    ICG_MWTitle("Pacman");
    ICG_MWSize(620, 700);
}

void ICGUI_main()
{
    int FRM1;
    static ICBYTES field, screen;
    int color = 0x2e27ff;
    CreateImage(field, 560, 620, ICB_UINT);
    FRM1 = ICG_FrameMedium(5, 5, 400, 200);
    field = 0;
    Line(field, cage, 0x3080ff);
    Line(field, cepheic, 0x3080ff);
    Line(field, cephedis, 0x3080ff);
}
```

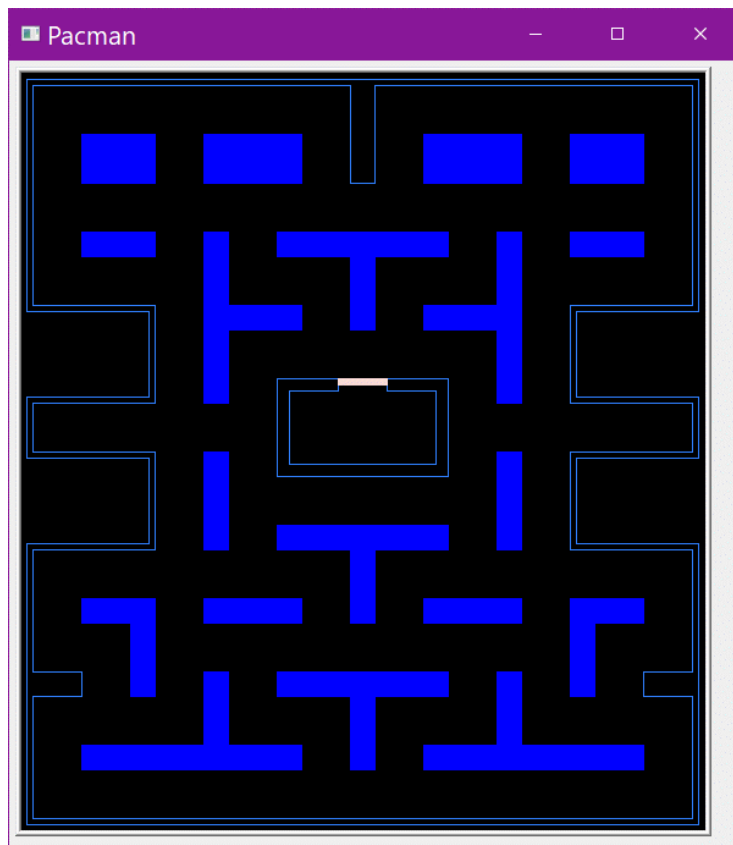
```
FillRect(field, 260, 365, 40, 5, 0xFADBD8);  
Rect(field, labirent, 0x3080ff);  
FlipV(field, screen);  
DisplayImage(FRM1, screen);  
}
```

This code, consisting of a total of 50 lines, can draw the labyrinth of the Pacman game as seen below:



In this example, the reason why our number of function calls decreased is because functions such as `Line()` and `Rect()` also accept matrices containing these coordinates as input instead of pixel coordinates. These functions can draw numerous geometric shapes using a matrix of any size with integer elements and at least four elements in the X dimension ($m \times n$; $m > 3$). Designing a game becomes very easy when a graphic design program that can calculate these coordinates is used.

Moreover, the appearance of the maze can be easily changed with a short touch. For example, when we use `FillRect()` instead of the `Rect()` function to create the maze and change its color to full blue (`0xff`), we get the following image.



5.3. Image Marking

Many applications require graphical drawings. We can give financial programs as an example. Financial data can be drawn on the screen as various types of graphs to observe trends. Other areas where data may need to be plotted are basic fields such as physics and mathematics. Here, too, measured or calculated data can be reflected on the screen as curves. There are methods in the “I Only See Bytes” library to mark important points on these graphs and write the values on the image.

The first thing that comes to mind is the circle drawing function. Using this function, small circles can be drawn to mark the reading points of the graph:

```
bool Circle(ICBYTES& i, int x, int y, int r, int color);
```

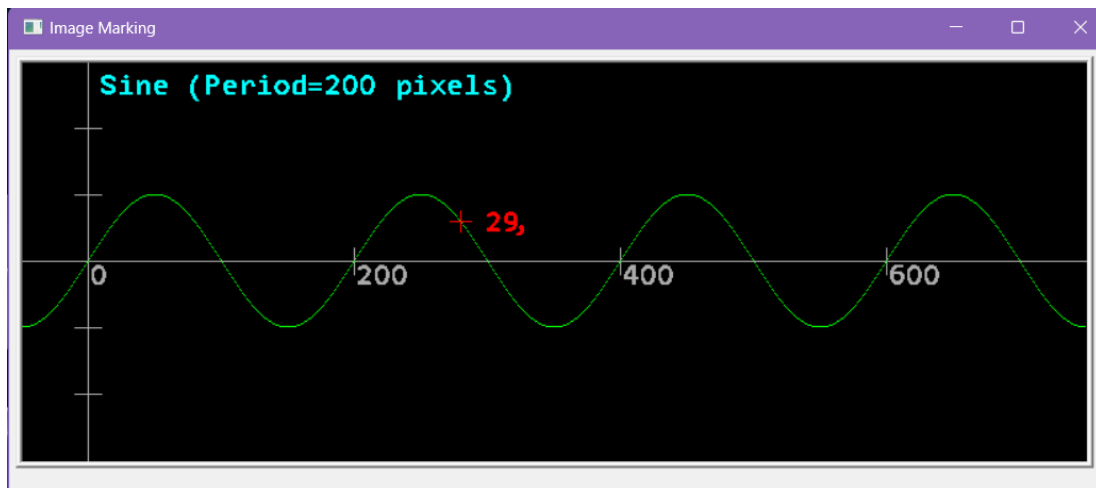
In this function, x and y denote the center of the circle, and r denotes its radius. Apart from this, some of the functions used only for image marking are as follows:

```
void MarkPlus(ICBYTES& i, int x, int y, int size, int color);
void MarkVert(ICBYTES& i, int x, int y, int size, int color);
void MarkHorz(ICBYTES& i, int x, int y, int size, int color);
```

While the topmost of these functions draws a plus-shaped shape with center coordinate (x,y) and dimensions (size) variable, `MarkVert()` draws a vertical line and `MarkHorz()` draws a horizontal line. Among the marking functions, the most important one is undoubtedly the function below, which allows you to write on the image.

```
void Impress12x20(ICBYTES& i, int x, int y, const char* txt,
                 unsigned color);
```

This function uses a 12x20 pixel-tall, fixed-width font and writes everything you write in the "txt" character string onto the image on a single line, starting from the point determined by the (x,y) coordinates of the upper right corner. Below is a screenshot of an application written using these functions.



In this application, the graph of a sine signal with a peak of 50 pixels and a full period of 200 pixels is drawn. The value of the horizontal coordinate at the point where $x = 330$ is marked using the (+) symbol, and the value of the function at that point (29.40) is printed on the image. Also, at the top of the image, the characteristics of the signal are printed in turquoise. The code for this application is given on the next page.

```

#include"icb_gui.h"
#include<math.h>

void ICGUI_Create()
{
    ICG_MWTitle("Image Marking");
    ICG_MWSize(850, 400);
}

void ICGUI_main()
{
    int FRM1;
    static ICBYTES image;
    FRM1 = ICG_FrameMedium(5, 5, 400, 200);
    CreateImage(image, 800, 300, ICB_INT);
    Impress12x20(image, 60, 10, "Sine (Period=200 pixels)", 0x00ffff);
    Line(image, 50, 1, 50, 300, 0xaaaaaa);
    Line(image, 1, 150, 800, 150, 0xaaaaaa);
    MarkVert(image, 250, 150, 20, 0xaaaaaa);
    MarkVert(image, 450, 150, 20, 0xaaaaaa);
    MarkVert(image, 650, 150, 20, 0xaaaaaa);
    MarkHorz(image, 50, 100, 20, 0xaaaaaa);
    MarkHorz(image, 50, 50, 20, 0xaaaaaa);
    MarkHorz(image, 50, 200, 20, 0xaaaaaa);
    MarkHorz(image, 50, 250, 20, 0xaaaaaa);
    Impress12x20(image, 53, 153, "0", 0xaaaaaa);
    Impress12x20(image, 253, 153, "200", 0xaaaaaa);
    Impress12x20(image, 453, 153, "400", 0xaaaaaa);
    Impress12x20(image, 653, 153, "600", 0xaaaaaa);
    double f;
    int y, decimal, mark;
    char decimals[8], fractions[8];
    for (int x = 1; x < 800; x++)
    {
        f = 50.0 * sin(3.1415 * (double)(x-50) * 0.01);
        y = 150.0 - f;
        image.I(x, y) = 0x00ff00;
        if (x == 330)
        {
            MarkPlus(image, x, y, 15, 0xff0000);
            _ecvt_s(decimals, 32, f, 4, &decimal, &mark);
            _ecvt_s(fractions, 32, f, 4, &decimal, &mark);
            decimals[decimal] = 0;
            Impress12x20(image, x+20, y-7, decimals, 0xff0000);
            Impress12x20(image, x + (decimal + 1)*13, y - 7, ",", 0xff0000);
            Impress12x20(image, x+(decimal + 2)*13, y - 7,
                        &decimals[decimal], 0xff0000);
        }
    }
    DisplayImage(FRM1, image);
}

```

When we examine this code, the first thing that catches our eye is the `<math.h>` library added to our project other than "icb_gui.h". This library has been added to call the sine function `sin()`. Then, an image matrix of 800x300 dimensions is created in a fluid called "image" within the `ICGUI_main()` function. Immediately afterwards, we see that the text "Sine (Period = 200 pixels)" has been printed in turquoise color on the left of this image using the `Impress12x20()` function.

The x-axis and y-axis are drawn by calling the `Line()` function twice, one after the other, and then vertical notches are made at the 200, 400, and 600 points of the x-axis with the `MarkVert()` method. The same is done on the y-axis using the `MarkHorz()` function, thus drawing the x and y axes. Then, using the `Impress12x20()` function, 0, 200, 400 and 600 are written to the bottom right of the notches on the x-axis, starting from the origin.

After all this, in order to plot the sine signal, y value is entered in the for loop increasing from x = 1 to 800:

```
f = 50.0 * sin(3.1415 * (double)(x-50) * 0.01);
```

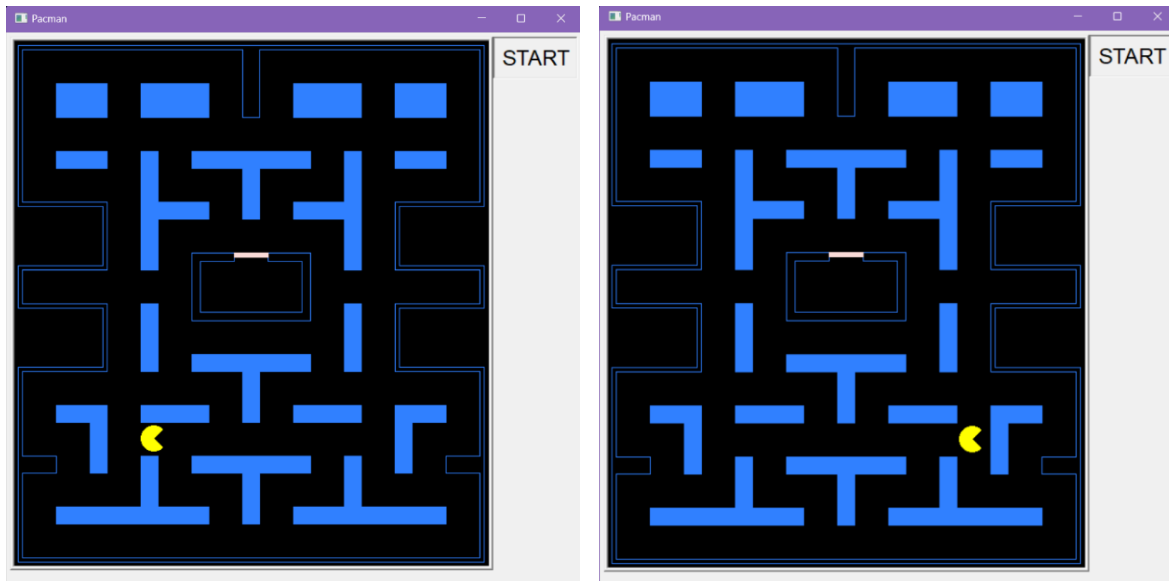
A sine function with an amplitude of 50, a period of 200 pixels and shifted to the left by 50 pixels is calculated and plotted. When the for loop index x reaches 330, an "if" condition marks the value of the sine function at that point with a red plus sign and writes the value next to it. The `_ecvt_s()` function is used to convert double floating point numbers to a character string.

5.4. Simple Animation

Game programming can actually be summarized as interactive animation programming. After a fixed background is drawn, different moving objects that the player and the computer manage need to be drawn. In game programming jargon, these objects are called sprites or Moving Object Blocks (MOBs).

However, today, short-term animations have become an indispensable part of the user interface. For example, flashing a button that prompts you to click on it or changing the shape of a setting box can be considered in the category of simple and short-term animations. Let's continue with the Pacman game as an example of how this is done. Below is a screenshot of the start and end moments

of an animation that moves HNB, the protagonist of the Pacman game, whose labyrinth we drew in section 5.2, horizontally.



The circular Pacman object shown here starts at the central point (165,470) on the image and slides horizontally 300 pixels in approximately 2.5 seconds. The button function that performs this animation is given below..

```
int FRM1;

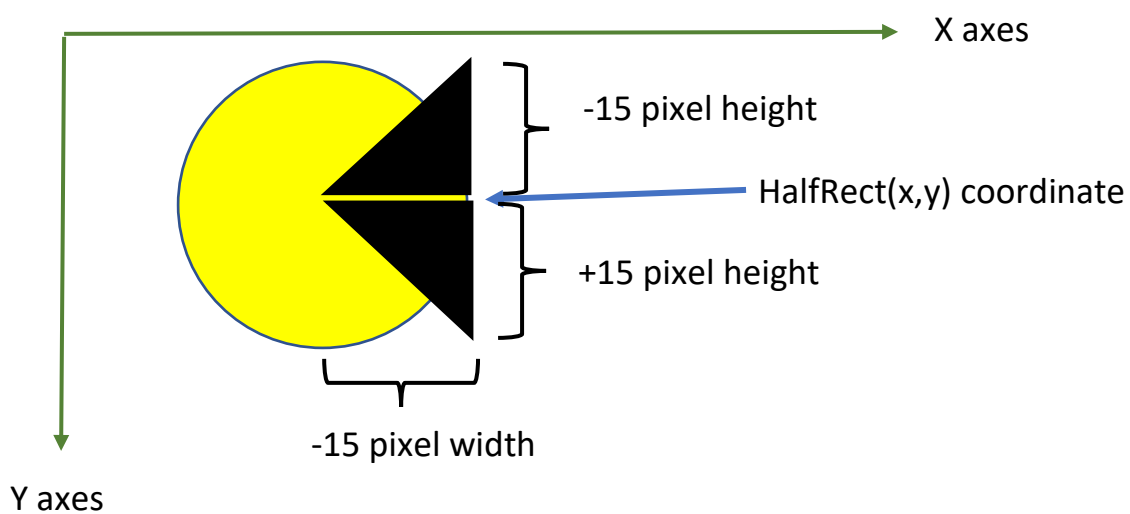
void Pacman(void *i)
{
    ICBYTES& screen = *(ICBYTES*)i;
    for (int x = 150; x < 450; x+=2)
    {
        FillCircle(screen, x - 15, 470, 15, 0xffff00);
        HalfRect(screen, x, 470, -15, -15, 0);
        HalfRect(screen, x, 470, -15, 15, 0);
        DisplayImage(FRM1, screen);
        Sleep(10);
        FillRect(screen, x - 30, 455, 30, 30, 0);
    }
}
```

We encounter three new methods in this function. The first of these, `FillCircle()`, draws a filled yellow circle with a center (x,y) and a radius of 15, as the name suggests. Then, to create Pacman's mouth, black right triangles are drawn with the method called `HalfRect()` and a part of the yellow circle is deleted. How this happens is shown in detail below.

The first two inputs of the method called `HalfRect()` are the coordinates of the perpendicular corner of the created triangle. The next two inputs are the

width and length of the created triangle. However, if these are negative, the triangle is drawn in the opposite direction.

After drawing Pacman, the program is frozen for 10 milliseconds with the `Sleep(10)` command. 10 milliseconds is 1/100 of a second. Then, a black, solid rectangle is drawn over it and Pacman is completely deleted from the screen. However, immediately afterwards, Pacman is redrawn two pixels to the right, and our eye perceives this as Pacman's movement. When these operations are repeated 150 times, our eyes see them as an animation of Pacman sliding to the right.



Another thing that is different in this code is that an `ICBYTES` fluid is sent as a parameter to the button function. The button function parameter can only be a void type pointer. Therefore, by type shifting, the pointer named "i" has been converted into a reference in `ICBYTES`:

```
ICBYTES& screen = *(ICBYTES*)i;
```

The content of the `ICGUI_main()` function is given below:

```
void ICGUI_main()
{
    static ICBYTES field, screen;
    int color = 0x2e27ff;
    CreateImage(field, 560, 620, ICB_UINT);
    ICG_Button(570, 5, 100, 25, "START", Pacman, &screen);
}
```

```

FRM1 = ICG_FrameMedium(5, 5, 400, 200);
field = 0;
Line(saha, zindan, 0x3080ff);
Line(saha, cepheic, 0x3080ff);
Line(saha, cephedis, 0x3080ff);
FillRect(field, 260, 365, 40, 5, 0xFADBD8);
FillRect(field, labirent, 0x3080ff);
FlipV(field, screen);
DisplayImage(FRM1, screen);
}

```

5.5. More Complex Figures

It was enough to use circles and half rectangles to draw the Pacman. But many games require more complex figures. There are other functions in the I-See-Bytes to help you draw more complex objects. The Paste() function is one of those. For example, By using the LoadImage() function, you can load the image below into a small matrix and then paste that matrix onto another larger one..



On the other hand, you can also use SetPixels() function to create graphic objects. The code below shows how to create a simple icon of the Turkish Jet Fighter, Kaan.

```

ICBYTES m;
int color = 0x2e27ff;
CreateImage(m, 210, 100, ICB_UINT);
int FRM = ICG_FrameMedium(5, 5, 160, 110);

ICBYTES kaan = { {10,1},{10,2},{9,3},{10,3},{11,3},{9,4},{10,4},{11,4},{8,5},
{9,5},{10,5},{11,5},{12,5},{8,6},{9,6},{10,6},{11,6},{12,6},{8,7},{9,7},{10,7},{11,7},
{12,7},{7,8},{8,8},{12,8},{13,8},{7,9},{8,9},{12,9},{13,9},{7,10},{8,10},{12,10},
{13,10},{7,11},{8,11},{12,11},{13,11},{7,12},{8,12},{9,12},{11,12},{12,12},{13,12},
{6,13},{7,13},{8,13},{9,13},{10,13},{11,13},{12,13},{13,13},{14,13},{5,14},{6,14},
{7,14},{8,14},{9,14},{10,14},{11,14},{12,14},{13,14},{14,14},{15,14},{4,15},{5,15},
{6,15},{7,15},{8,15},{9,15},{10,15},{11,15},{12,15},{13,15},{14,15},{15,15},
{16,15},{3,16},{4,16},{5,16},{6,16},{7,16},{8,16},{9,16},{10,16},{11,16},{12,16},{13,16},
{14,16},{15,16},{16,16},{17,16},{2,17},{3,17},{4,17},{5,17},{6,17},{7,17},{8,17},
{9,17},{10,17},{11,17},{12,17},{13,17},{14,17},{15,17},{16,17},{17,17},{18,17},
{1,18},{2,18},{3,18},{4,18},{5,18},{6,18},{7,18},{8,18},{9,18},{11,18},{12,18},
{13,18},{14,18},{15,18},{16,18},{17,18},{18,18},{19,18},{1,19},{2,19},{3,19},{4,19},
{5,19},{6,19},{7,19},{8,19},{10,19},{12,19},{13,19},{14,19},{15,19},{16,19},{17,19},
{18,19},{19,19},{1,20},{2,20},{3,20},{4,20},{5,20},{6,20},{7,20},{8,20},{10,20},
{12,20},{13,20},{14,20},{15,20},{16,20},{17,20},{18,20},{19,20},{4,21},{5,21},{6,21},
{7,21},{8,21},{9,21},{10,21},{11,21},{12,21},{13,21},{14,21},{15,21},{16,21},
{7,22},{8,22},{9,22},{10,22},{11,22},{12,22},{13,22},{6,23},{7,23},{9,23},{10,23},

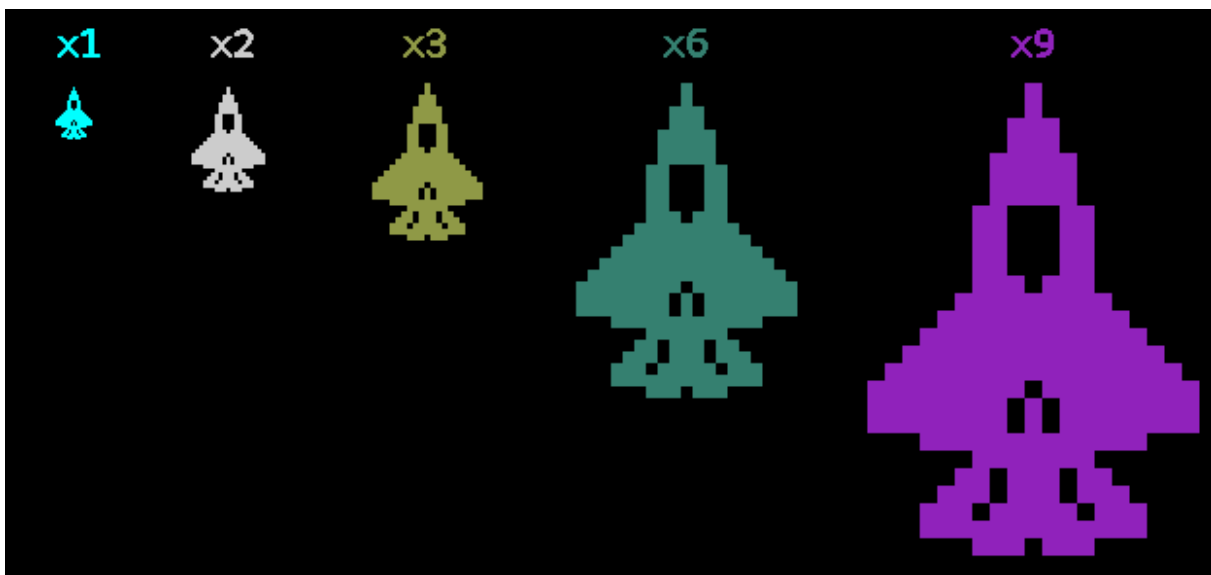
```

```

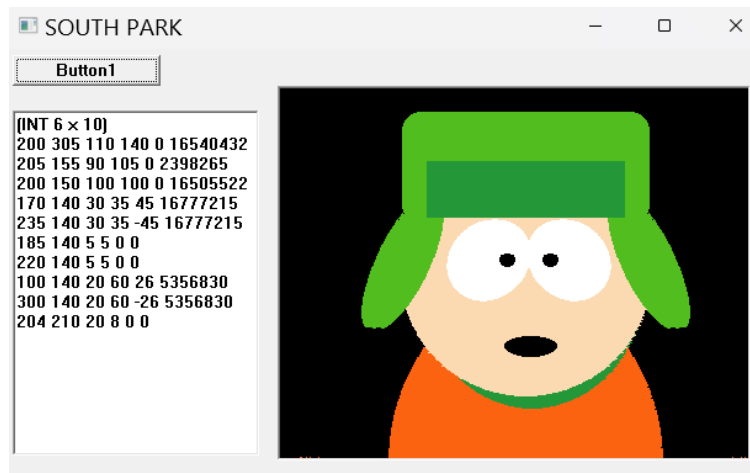
{11,23},{13,23},{14,23},{5,24},{6,24},{7,24},{9,24},{10,24},{11,24},{13,24},{14,24},
{15,24},{4,25},{5,25},{6,25},{8,25},{9,25},{10,25},{11,25},{12,25},{14,25},{15,2},
5},{16,25},{4,26},{5,26},{6,26},{7,26},{8,26},{9,26},{10,26},{11,26},{12,26},{13,2},
6},{14,26},{15,26},{16,26},{7,27},{8,27},{9,27},{11,27},{12,27},{13,27}};
m = 0;
SetPixels(kaan, 65, 15, 0x8f9946, m);//x3
ICBYTES p, q;
CreateImage(p, 19, 27, ICB_UINT);
SetPixels(kaan, 1, 1, 0x9022bb, p);//x9
MagnifyX3(p, q);
Paste(q, 150, 15, m);//x9
SetPixelsX2(kaan, 100, 15, 0x358070, m);//x6
MagnifyX3(m, panel);
SetPixels(kaan, 30, 45, 0xffff, panel);
Impress12x20(panel, 30, 15, "x1", 0xffff);
SetPixelsX2(kaan, 100, 45, 0xcccccc, panel);
Impress12x20(panel, 109, 15, "x2", 0xcccccc);
Impress12x20(panel, 208, 15, "x3", 0x8f9946);
Impress12x20(panel, 342, 15, "x6", 0x358070);
Impress12x20(panel, 521, 15, "x9", 0x9022bb);
DisplayImage(FRM, panel);

```

In this example, a matrix of size $2 \times N$ is created inside the fluid named “kaan.” This matrix contains the coordinates of the pixels of the shape that is going to be drawn. Later on, you can draw it in actual size by using `SetPixels()` or `SetPixelsX2()` to draw it twice the size. On the other hand, `MagnifyX3()` function can magnify an image matrix three times. By combining these functions in different orders, one can draw an object in many magnification factors.



In the example below, you see a program that draws Kyle Broflovski from the adult cartoon South Park. This program is only 28 lines long.



During the drawing of this picture, we used new functions that we have never mentioned before. The first of these is the `FillRoundRect()` function. This function is used to draw the rounded rectangle of the child's hat.

In order to draw rest of the elliptical shapes we used `FillTiltedEllipse()` function. This command draws a filled and tilted ellipse. This function needs 5 arguments. The first two are the coordinates of the center of the ellipse (x,y), the horizontal and vertical radiuses of the ellipse (rx,ry) and the tilt angle (integer degrees) and finally the color of the ellipse. Below is the code of this program:

```
#include"icb_gui.h"
int MLE, FRM;
ICBYTES img;
void ICGUI_Create()
{
    ICG_MWTitle("SOUTH PARK");
    ICG_MWSize(640, 400);
}

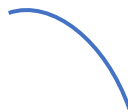
void butonfunc()
{
    CreateImage(img, 380, 300, ICB_UINT);
    ICBYTES elpscor={{200,305,110,140,0,0xfc6310},{205,155,90,105,0,0x249839},
    {200,150,100, 100, 0, 0xfbdab2}, { 170, 140, 30, 35, 45, 0xffffffff },
    {235,140,30,35,-45,0xffffffff},{185,140,5,5,0,0x0},{220,140,5,5,0,0x0},
    {100,140,20,60,26,0x51bd1e},{300,140,20,60,-26,0x51bd1e},{204,210,20,8,0,0x0}};
    FillRoundRect(img, 100, 20, 200, 90, 15, 0x51bd1e);
    FillTiltedEllipse(img, elpscor);
    FillRect(img, 120, 50, 160, 10, 0x51bd1e);
    FillRect(img, 120, 60, 160, 45, 0x249839);
    DisplayImage(FRM, img);
    DisplayMatrix(MLE, elpscor);
}
```

```
void ICGUI_main()
{
    ICG_Button(5, 5, 120, 25, "Button1", butonfunc);
    MLE = ICG_MLEditSunken(5, 50, 200, 280, "", 0);
    FRM = ICG_FrameSunken(220, 30, 380, 300);
}
```

Finally, we shall talk about a function that draws elliptical arcs. This function draws the peripheral of a tilted ellipse or a portion of it:

```
void TiltedEllipseArc(ICBYTES& img, int x, int y, int rx, int ry, int angle, int color, int arc_strt = 0, int arc_end = 360);
```

For example the shape shown below is a 90 degree portion of a tilted ellipse. The arguments shown above are the center of the ellipse (x,y), the horizontal and vertical radiuses (rx,ry), the angle of the tilt in degrees, the color, and the degrees of the start and end of the arc (arc_strt, arc_end). If these last values are not used (not entered) then their default values are 0 and 360 which draws an entire tilted ellipse.



6. Device Access

One of the greatest conveniences of the I-See-Bytes library is device access. While image processing libraries such as OpenCV do not allow access to the microphone or sound card output, on the other hand, special libraries written for audio processing do not allow access to devices such as cameras and LIDAR. The professional version of the I-See-Bytes library has the ability to access IP cameras over the Ethernet, LIDARs and even robot arm control circuits. However, in the student version, these capabilities are limited to sound card input/output and access to USB cameras (including IP cameras with Direct-X/MFC drivers). In this section, we will learn how to write applications that include device access.

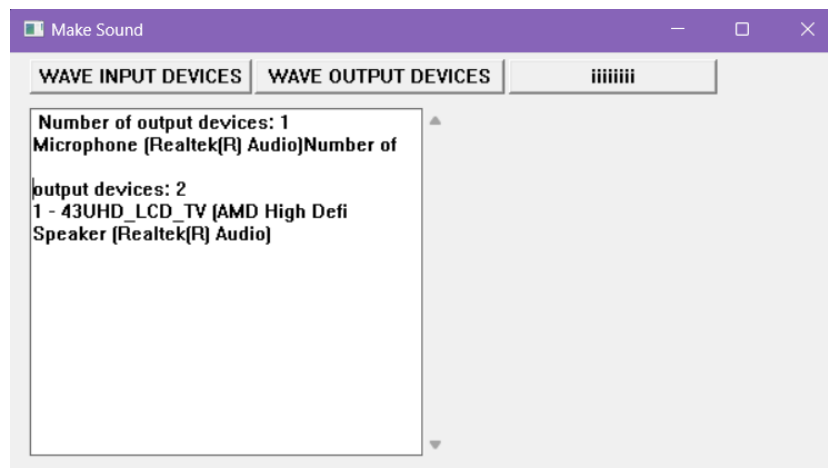
6. 1. Sound Output

In the first chapter, we mentioned that there are two basic data types in the I-See-Bytes library and the first of them is ICBYTES, which we use to store data and gives the library its name, and the other one is ICDEVICE, which we will use to access devices. In this section we will learn how to use the ICDEVICE fluid.

The first step in device access is to query the number and properties of devices connected to the computer. However, unfortunately, many libraries cannot provide this feature. For example, the OpenCV library cannot even tell you the number of cameras installed on your computer. However, the I-See-Bytes library has very advanced system polling functions compared to many other libraries. For example, the following commands not only tell you how many audio output devices you have, but also their brands and features:

```
ICBYTES output;  
int number=WaveOutputDevices(output);
```

What is meant by “wave” here is a sound wave signal. However, since waves beyond the hearing threshold of the human ear can also be created, they are only called waves. This function expects a fluid in ICBYTES as input. Let's say, if there are two devices with audio output features connected to your computer's hardware, then it creates a 32x2 matrix in terms of characters and writes their brands and models on each line. In the application whose screenshot is given below, the number and names of the audio input and output devices connected to the computer it is currently running are shown.



When the first button from the left is pressed, the number and names of the wave input devices are written on the screen, and when the middle button is pressed, the number and names of the wave output devices are written. It is understood from this output that this computer's own speaker is connected to a Realtek sound chip. Again, the computer is most likely connected to a 43-inch UHD screen or television via HDMI cable and can also access its sound system. More importantly, the 43-inch display is currently set to be the default audio interface. The function called when the “WAVE OUTPUT DEVICES” button in the middle is pressed is shown below:

```
void OutputDevices()
{
    ICBYTES output;
    ICG_printf(MLE, "Number of output devices: %d\n", WaveOutputDevices(output));
    Print(MLE, output);
    ICG_printf(MLE, "\n");
}
```

Here, “MLE” is the handle of the multi-line text box in which the device information is printed in the picture above. The entire code of this application is shown below.

```

#include "icb_gui.h"

int MLE;
unsigned char ii[67] = {139,131,127,108,108,96,67,84,54,31,40,18,9,11,5,11,12,
30,37,54,74,91,103,127,139,147,164,176,194,202,218,230,230,244,250,248,237,
226,225,201,182,171,139,117,98,74,56,37,26,24,9,9,11,16,24,37,43,58,75,84,84,
103,108, 122,124,138,141 };

void ICGUI_Create()
{
    ICG_MWTitle("Make Sound");
    ICG_MWSize(620, 350);
}

void InputDevices()
{
    ICBYTES input;
    ICG_printf(MLE, " Number of input devices: %d\n", WaveInputDevices(input));
    Print(MLE, input);
    ICG_printf(MLE, "\n");
}

void OutputDevices()
{
    ICBYTES output;
    ICG_printf(MLE, "Number of output devices: %d\n", WaveOutputDevices(output));
    Print(MLE, output);
    ICG_printf(MLE, "\n");
}

void iiii()
{
    ICDEVICE d;
    ICBYTES i;
    if (CreateSound(i, 1, 3000, ICB_UCHAR, 8000) == 0)
    {
        for (int y = 0; y < 3000; y++) i.B(1, y) = ii[y % 61];
    }
    if (!CreateCompatibleDevice(d, ICB_WAVEOUT, 0, i))
        ICG_printf(MLE, "Can't access sound card!\n");
    if (!WaveOut(d, i))
        ICG_printf(MLE, "Device cannot play this sound!\n");
    Sleep(400);
    WaveOut(d, i);
    CloseDevice(d);
}

void ICGUI_main()
{
    ICG_Button(15, 5, 160, 25, "WAVE INPUT DEVICES", InputDevices);
    ICG_Button(177, 5, 180, 25, " WAVE OUTPUT DEVICES ", OutputDevices);
    ICG_Button(360, 5, 150, 25, "iiiiiii", iiii);
    MLE = ICG_MLEdit(15, 40, 300, 250, "", SCROLLBAR_V);
}

```

As can be seen from the source code, this application's main purpose, apart from querying the audio input/output devices connected to the computer, is to

produce the "iiii" sound. In order to do this, the wave information containing the digital data of this sound is written into the "unsigned char ii[67]" array at the beginning of the code. Then, when the button that makes the sound is pressed, the function called `iiiiii()` calls a function that we have encountered for the first time:

```
CreateSound(i, 1, 3000, ICB_UCHAR, 8000)
```

This function actually creates a matrix of size 1x3000 in bytes. However, since this matrix (vector) carries audio data, it creates some special data structures along with the matrix, just as the `CreateImage()` function to create a matrix that carries the image. Thus, when the user wants to listen to this sound, the process of playing the sound is faster because these special data structures are readily available. Additionally, the `CreateSound()` function expects an extra input, unlike the `CreateMatrix()` or `CreateImage()` functions. This is the last number "8000". This number sets the speed at which the audio data will be played. In other words, it indicates that the sound card will use 8000 samples of the data in this matrix every second. Since our matrix is 3000 long, this means that this matrix can only produce sound for $3/8$ seconds = 375 milliseconds.

If you pay attention, the top array "ii[67]" is only 67 elements long. However, fortunately, the sound it produces is periodic, that is, a repeating sound. Therefore, if we write this index consisting of 67 elements repeatedly into our matrix (actually, it would be more accurate to call it our vector) consisting of 3000 elements, we can extend the duration of the sound "iiii". This is what the following for loop does.

Then we encounter a new function again:

```
ICDEVICE d;  
ICBYTES i;  
CreateCompatibleDevice(d, ICB_WAVEOUT, 0, i)
```

This function tells us that we want to reach the wave output device number zero and play the sound wave with numerical data in the "i" matrix on this device. What we need to remember here is that the i matrix contains mono, that is, single channel (if it were dual channel, it would be stereo), data in bytes and 8000 of these data must be played per second. Therefore, we need hardware that can do these. If the hardware is capable of playing this data, a data structure that provides access to this hardware is created in the "d" fluid and the hardware becomes ready for use. In this case, the `CreateCompatibleDevice()` function returns true. If there is a positive return, all we need to do from now on is to call

the WaveOut() function as many times as we want, send the "i" matrix to it and play the sound in the matrix. When we are done, we can delete the device access data created in the "d" fluid with the CloseDevice(d) command.

If we wanted to produce stereo sound, we would have to create our matrix with dimensions 2x3000 instead of 1x3000:

```
CreateSound(i, 2, 3000, ICB_UCHAR, 8000);
```

Then, we would have to write into i.B(2,y) along with i.B(1,y) inside the for loop.

6.2. Reading Sound Files

In the Windows operating system, the audio recording format called WAVE or WAV, which is the abbreviation of the words "Waveform Audio Format" developed jointly by IBM and Microsoft, is used. This format is one of the two most used formats, along with the AIFF format, especially for uncompressed audio files. All sound notifications made by the Windows operating system at startup, when you make an error or for any other reason are kept in ".wav" format files. You can access these files from the C:\Windows\Media file. To load audio files in this format into the matrix, all you have to do is call the ReadWave() function:

```
ICBYTES A;  
ReadWave(A, "C:/Windows/Media/tada.wav");
```

If this function reads the file successfully, you can use another version of the WaveOut() function for convenience to play the sound matrix in fluid A:

```
WaveOut(A, 0);
```

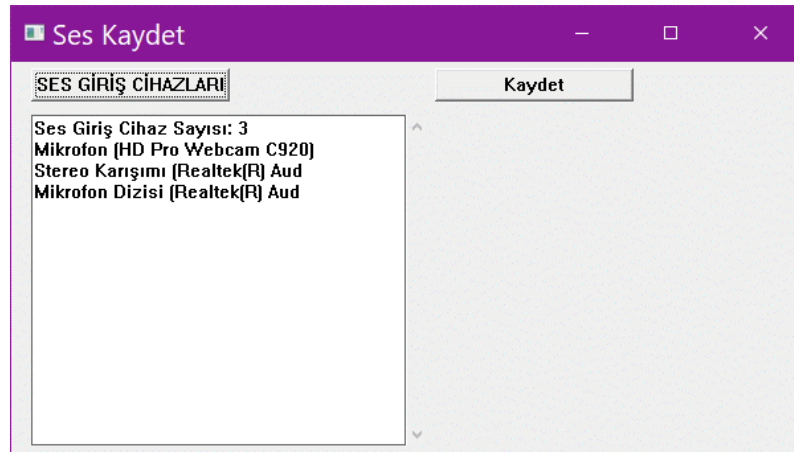
This function ensures that the audio data carried by matrix A in a single line is played by the fixed audio output device numbered zero. By the way, before playing the "tada.wav" file, if you execute the following command:

```
A*=6;
```

You will hear the volume increase considerably. Here, we increased the amplitude of the sound by multiplying the sound data by 6. If we multiplied by 15, you would hear distortion in the sound because we would have reached numbers that were too large (overflow) for the variable type used by the matrix.

6. 3. Sound Recording

If we want to record sound from a microphone or another signal recording input into a matrix or vector, we need to repeat steps very similar to those we did in the previous section. First of all, we need to determine the device with which we will record sound or signal. For this, we must list the audio/signal input devices connected to our computer:



Unlike last time, this time we see three input devices connected to our computer. This time, we see that our zero numbered signal input device is the microphone of an HD PRO Webcam C920 model webcam from Logitech. The number one input device is a Stereo Mixer, and in the last row, number two, we see that the microphone input of our computer's sound card. The function shown below, which is called when we click the "Record" button, takes a recording from the webcam's microphone at a speed of 8000 samples per second for two seconds and then plays it back to us:

```
void Record()
{
    ICDEVICE d;
    ICBYTES i;
    CreateSound(i, 1, 16000, ICB_UCHAR, 8000);
    if(CreateCompatibleDevice(d, ICB_WAVEIN, 0, i))
        ICG_printf(MLE, "Recording Started\n");
    WaveIn(d, i);
    ICG_printf(MLE, " Recording Complete\n");
    WaveOut(i, 0);
}
```

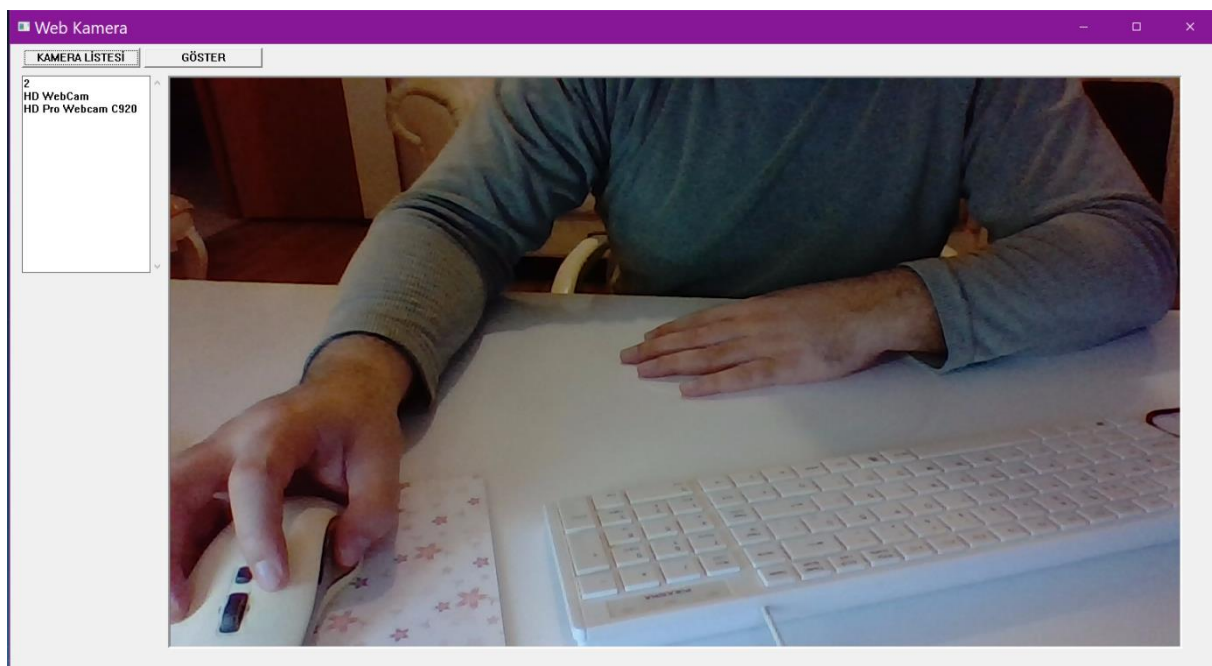
As explained in the previous section, we first created a vector using the CreateSound() function. Unlike last time, this time we created our vector with a length of 16000, not 3000. This is because we want to get a recording that will last two seconds. If we want the microphone to take 8000 digital samples per

second, we need a vector long enough to hold 16000 numbers for two seconds. Then we used the `CreateCompatibleDevice()` function again, but this time we gave the second parameter as `"ICB_WAVEIN"`. This parameter told the function that we wanted to create an input device. If you noticed in the previous section, this parameter was `"ICB_WAVEOUT"`.

Then, we placed the audio recording into the `"i"` vector by calling the `WaveIn()` function. Now there is nothing left to do but listen to the sound we recorded with the `WaveOut()` function. In this section, we learned how easy it is to access audio input and output with the I-See-Bytes library.

6. 4. Camera Access

The application shown below shows the number of web cameras connected to our computer, model names and the image taken from the camera.



According to the text box, two cameras are connected to this computer. Of these, the fixture numbered zero is called "HD Webcam", which has a general name. Such names are often used to describe webcams that come on a notebook or tablet computer. From this description, we can guess that the computer being used is a laptop. We understand that the second camera is the HD PRO Webcam C920 model from Logitech. The source code of this application is shown below.

```

#include"icb_gui.h"

int MLE,FRM;

void ICGUI_Create()
{
    ICG_MWTitle("Web Kamera");
    ICG_MWSize(1550, 850);
}

void Kameralar()
{
    ICBYTES kamera_listesi;
    ICG_printf(MLE,"%d\n",GetVideoSourceList(kamera_listesi));
    Print(MLE, kamera_listesi);
}

void Goster()
{
    ICDEVICE d;
    ICBYTES i;
    CreateDXCam(d, 0);
    for (int x = 0; x < 100; x++)
    {
        CaptureImage(d, i);
        DisplayImage(FRM, i);
        Sleep(30);
    }
    CloseDevice(d);
}

void ICGUI_main()
{
    ICG_Button(15, 5, 150, 25, "KAMERA LİSTESİ", Kameralar);
    ICG_Button(170, 5, 150, 25, "GÖSTER", Goster);
    MLE = ICG_MLEdit(15, 40, 180, 250, "", SCROLLBAR_V);
    FRM = ICG_FrameDeep(200, 40, 1024, 1024);
}

```

In this application, we see that the function called GetVideoSourceList() is called when the "CAMERA LIST" button is pressed. This function takes an ICBYTES fluid as input. The function places a matrix of 32xcamera characters into this fluid and writes the name of one of the cameras connected to the system on each line of this matrix. Additionally, the function itself returns an integer, which tells us the number of cameras. Then, when we print the ICBYTES fluid (camera_list) that

we gave as input to the function, we see the model of the two cameras currently connected to the computer.

When we press the other button, SHOW, we see that the image from the camera is reflected on the screen for 3 seconds. When we examine the function that performs this, we encounter a new function used to create a device:

```
ICDEVICE d;  
ICBYTES i;  
CreateDXCam(d, 0);
```

This function creates an ICDEVICE type device structure that allows us to connect camera number zero. The term DXCam in the name of this function comes from the abbreviation of "DirectX Camera". What this means is:

If a camera is connected to your computer through any hardware channel, these routes may be of different standards such as USB, Ethernet, FireWire etc.; If this camera is introduced to the operating system via DirectX drivers, you can access it with the CreateDXCam() function.

So, in addition to USB cameras, you can also connect to network cameras if they have DirectX drivers. The student version of the I-See-Bytes library only provides access to cameras via DirectX. If you want to access IP, that is, network cameras via RTSP protocol, you should use higher versions of this library (for example, the Professional version).

Following the Show function, another command appears for the first time:

```
CaptureImage(d, i);
```

As the name suggests, this function captures an image from the "d" device, which in this case is the camera, into the "i" fluid. This picture is the image that the camera is currently looking at. Then this image is printed on the screen. When this process is repeated 100 times every 30 milliseconds, a 3-second live image is projected on the screen.

Microsoft has stopped using the Direct-X library. Although this library is still supported, it has recently started using a different library called "Microsoft Foundation Classes" (MFC). The I-See-Bytes library supports both libraries. If you want to access the camera using this library:

```
ICDEVICE mfcam;  
if (CreateMFCam(mfcam, 0) < 0) return;
```

gives you access to the first camera. If you want to see what resolutions the camera supports:

```
ICBYTES modes;  
if (ListCamModesTxt(mfcam, modes))  
    Print(MLE, modes);
```

You can use the code. For example, for the camera on the laptop mentioned above, the following list will be displayed:

```
1280x720 30fps NV12  
1280x720 30fps MJPG  
640x480 30fps NV12  
640x480 30fps MJPG  
640x360 30fps NV12  
640x360 30fps MJPG  
1280x720 10fps YUY2  
640x480 30fps YUY2  
640x360 30fps YUY2
```

This list specifies the horizontal and vertical resolution (for example 1280x720), frame rate per second (fps) and image format. If desired, the user can select one of these resolutions when initiating the device. For example:

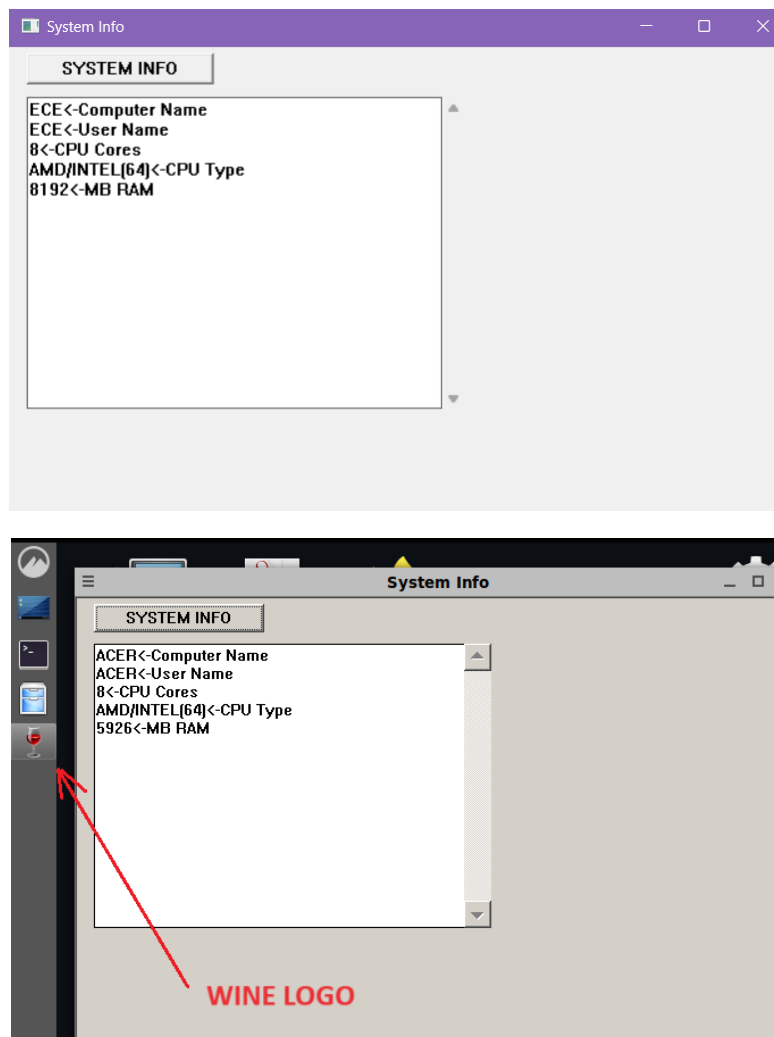
```
CreateMFCam(mfcam, 0, 640, 360);
```

With this command, you can specify that you want video in this specific resolution. If the user does not enter these parameters, the I-See-Bytes library will automatically select a suitable resolution from the top of the list. This is usually the highest resolution the camera can provide.

6. 5. System Information Query

Some applications need to know about the hardware of the computer they are running on. For example, the type of processor in the computer or how many cores it has. The application shown below lists the following information about the system in order:

- Computer Name
- User name
- Number of processor cores
- Processor Type
- RAM (amount of memory)



The image shown at the top was obtained by running this application on the **Windows operating system**. The one below was obtained by running the same executable, that is, the compiled code, on **64-bit Debian Linux**. In the Linux toolbar on the left, the icon of our program is shown with a wine glass with the WINE logo. As we mentioned before, the binary code of the programs you compile using the I-See-Bytes library, that is, the .EXE file, often runs smoothly on both Windows and Linux computers with WINE installed.

WINE is not an emulator, simulator or “virtual machine” (www.winehq.org). It's a compatibility layer. In other words, it is a layer that translates Windows system calls into Linux system calls. Therefore, it runs incredibly faster than Java or Python. You will observe that your programs run at almost the same speed as they run in the Windows operating system. The only difference is that when the program loads at the first run, it may take 4-5 seconds for it to appear on the screen. WINE compatibility layer is written for every Linux distribution (Ubuntu, Red Hat, Pardus, etc.). It is very easy to install. Many Linux variants even come with WINE installed as a default. For Linux distros where this is not offered, it is

much easier to install it manually. This means that the platform independence feature that Java boasts about is also offered by I-See-Bytes. Moreover, by producing programs that run at least 10 times faster than Java.

WINE program is now also produced for the Android operating system. However, this will require a version of the I-See-Bytes library compiled for the ARM processor. Such a version is not yet supported but is possible in the future.

The code of this application, which provides information about the system, is given below.

```
#include "icb_gui.h"

int MLE;

void ICGUI_Create()
{
    ICG_MWTitle("System Info");
    ICG_MWSize(650, 450);
}

void Info()
{
    ICBYTES info;
    SystemInfo(info);
    Print(MLE, info);
}

void ICGUI_main()
{
    ICG_Button(15, 5, 150, 25, "SYSTEM INFO", Info);
    MLE = ICG_MLEdit(15, 40, 350, 250, "", SCROLLBAR_V);
}
```

6. 6. Querying Peripheral Devices

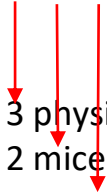
In the previous section, we learned how to query the properties of the circuits on the computer itself, that is, the motherboard. In this section, we will query the devices connected to the motherboard, called peripheral devices. The commands we will use for this are shown below:

```
ICBYTES info;
PeripheralInfo(info);
DisplayMatrix(MLE, info);
```

Sample output of these commands would be as shown below:

(SHORT 13 x 1)

3 2 2 3 1 0 0 9 0 2 1 0 0



3 physical disks (not logical)

2 mice

2 screens (monitors)

3 keyboards (including virtual keyboard)

1 web camera

0 scanners

0 printers

9 usb port multipliers and devices

0 sensors

2 batteries

1 security (TPM) device

If this query tells us that there is a battery connected to our device, it means that we are currently using a laptop or a tablet. The fact that there are two screens indicates that there is another monitor connected to the laptop other than its own screen.

6.7. File Access Device

The I-See-Bytes library also allows files on disk to be accessed as devices. The user is free to use the file open/read/write commands in C++ if he/she wishes. However, when these files are opened as devices, very convenient methods are provided to be used with the ICBYTES fluid. There are three types of file open commands:

```
ICDEVICE filedevice;
```

```
CreateFileReader(filedevice, "Myfile.bin");//To read only an existing file
```

```
CreateExistingFileWriter(filedevice, "Mine.bin");//To read/write existing file
```

```
CreateFileDevice(filedevice, " Myfile.bin");//If file exists, read /write access.
```

```
// If it doesn't, it creates the file
```

If successful, they return true as `bool`. All of these files should be closed with `CloseDevice()` when they are done. After the file is successfully opened,

```
long long GetFileLength(filedevice);
```

You can find out the length of the file with the command.

```
HANDLE GetFileHandle(filedevice); //Returns Windos OS handle
int C_RunTimeHandle(filedevice); // Returns C Runtime handle
```

Read Commands:

Our first command works in the same way as the various types of read commands in C++:

```
unsigned ReadIntoMatrix(filedevice, ICBYTES& mat);
```

This command starts reading from the beginning after the file is opened and goes down towards the bottom of the file as it reads. How much it reads depends on the size of your matrix. The number it returns is the amount of data that has been successfully read in bytes. An example of this is shown in the “Video Playback” section of Chapter 7. The matrix read with this command keeps the address from which the data was read in the file. You can access this data with the `GetAddress(ICBYTES& mat)` command. If you want to read a specific address, you can write this address to your matrix with the `SetAddress()` command and ensure that the reading starts from that address. Since the data in the matrix is inverted, use matrices with Y and Z dimensions of 1 and X dimensions of >1 to access the data in the software in order.

Our second command loads the entire ICBYTES fluid at the desired address:

```
long long ReadICBYTES(filedevice, ICBYTES& i, long long from_address);
```

function. The last argument of this function is the byte address from which the file will be read. You do not need to create the ICBYTES fluid as a matrix. The fluid information must already be present at the written point. In other words, if a matrix is written to this point, the type and dimensions of the matrix are also written there. The command creates the matrix itself using that data and fills it in. The address it returns is the point where the ICBYTES fluid data ends.

Write Commands:

Here is the command that writes the type of fluid that the `ReadICBYTES()` command we mentioned earlier can read to an address in the file:

```
long long WriteICBYTES(filedevice, ICBYTES& i, long long to_address)
```

If you write any matrix you create to an address in the file, all the information about the matrix, namely its dimensions, whether it is int or float, and everything else, is written there. Thus, another program reading that address in that file can create the same matrix without knowing its type and size. An example of this will

be shown in the next section, “A Simple Database.” The number returned by this command is the address where the data of the written ICBYTES stream ends. You can write a second stream to that point using this address.

If you want to always write just the contents of the matrix at the end of the file:

```
unsigned AppendMatrixToFile(filedevice, ICBYTES& mat);
```

You can use the function. It returns the amount of data you wrote in bytes.

We have previously mentioned the ReadIntoBuffer() function. Its equivalent command is shown below:

```
int WriteBuffer(filedevice, ICBYTES& mat);
```

Writes the contents of a matrix to a file. Used with the ReadIntoBuffer() function to modify a section of a file. For example, you can use these two commands one after the other to encrypt the file piece by piece.

6. 8. A Simple Database

In this example, we will build a simple database. The database we will build will not be professional quality because the purpose of this book is not to teach database science. However, it shows that the functions in the I-See-Bytes library can be used to develop simple database applications that will meet the needs of a small or medium-sized company. The first functions we will use for this purpose are for English keys,

```
unsigned long long* KeyMap(ICBYTES& inp, int maps);
```

and for Turkish keys:

```
unsigned long long* KeyMapTR(ICBYTES &inp, int maps);
```

These functions convert every 12 characters in the “inp” vector (array) into 64-bit positive integers. For example, let's say the name of the author of this book (“Ibrahim Cem Baykal”) is in the “inp” matrix. In this case,

```
unsigned long long* map=KeyMapTR(inp,2);
```

function will send us two 64-bit positive integers:

```
map[0]= 13163227624530739465
```

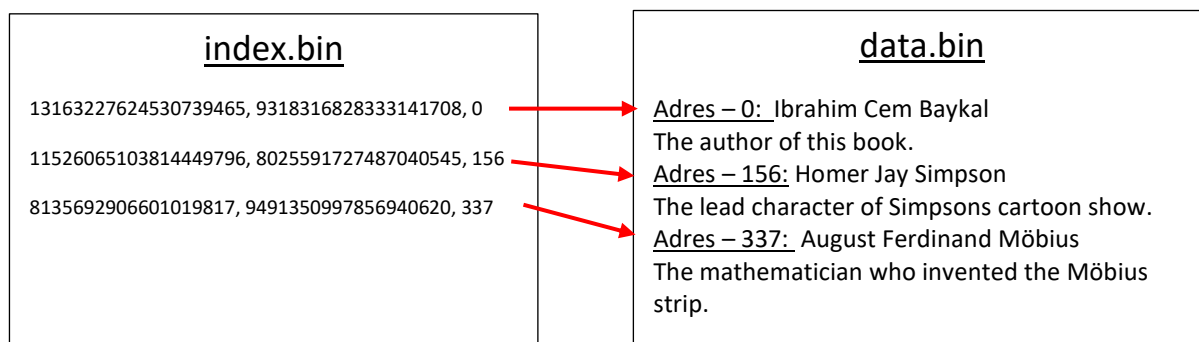
```
map[1]= 9318316828333141708
```

These numbers are obtained by compressing each of the 12 letters of the input in a special way. The input ("Ibrahim Cem Baykal") consists of 18 letters, including spaces. The first 24 letters of a database key are usually long enough to query that key. The KeyMapTR() function produces the same number in the input key lowercase/uppercase letter combinations. In other words, if you had sent ("ıBrahim ceM bAYKAL"), you would have obtained the same numbers in map[0] and map[1]. Remember that there is no uppercase "ı" in English. Also, "ÇĞÜÖ" and lowercase "ı" do not exist in English. For this reason, if you are going to use Turkish keys (alphabet) in your database, you can use the KeyMapTR() function, and if you are going to use English alphabet strictly, you can use the KeyMap() function. Also, the numbers produced grow in alphabetical order. In other words, if your key starts with "a", it will produce smaller numbers than a key that starts with "z". You can use this to sort your keys.

We will use two different files for our database. The first one will be the index file and will hold the map[] numbers we mentioned above and the locations of the actual data in the data file. The data file will hold both the key and the data itself that is called with that key. When we enter the data into our application and save it:



The index file and data file are organized as shown below:



The entire code of this database application, which is provided as a sample project with the I-See-Bytes library, will not be given here again. Only the important parts will be reviewed. For example, when the program first starts, the following function is called:

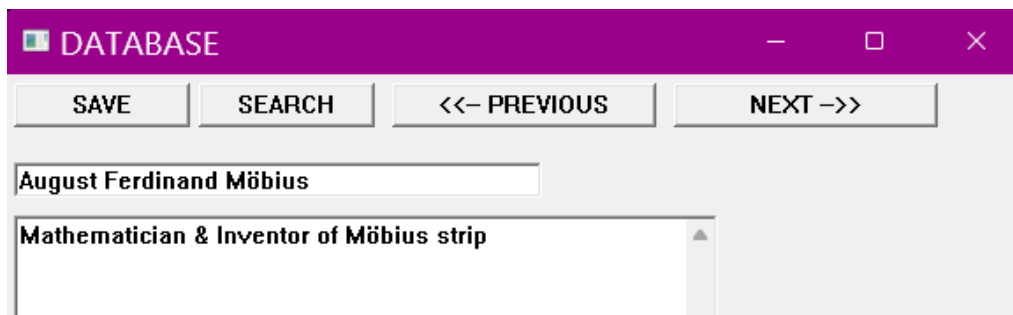
```
void AtStart(DATA_BASE& v)
{
    CreateMatrix(v.index, 3, 1, ICB_ULONGLONG);
    v.index = 0;
    CreateFileDevice(v.index_file, "index.bin");
    CreateFileDevice(v.data_file, "data.bin");
    //read index
    ReadICBYTES(v.index_file, v.index, 0);
}
```

As can be clearly seen, a 3x1 matrix named "index" is created in a sequence of DATABASE, then if the index.bin and data.bin files already exist, they are opened, if not, they are created. Then the ICBYTES fluid (if any) inside the index file is loaded. If index.bin is newly created, the 3x1 index matrix that we created at the top and whose elements are reset will not change.

Now let's assume that the user has opened the program and the filled index matrix as above is loaded. When the user will perform a search, he will type the key in the single-line text box and press the "SEARCH" button. When this button is pressed, the following function is called:

```
void Search(void *p)
{
    DATA_BASE& vt = *((DATA_BASE*)p);
    GetText(SLE, vt.key);
    unsigned long long* map = KeyMapTR(vt.key, 2);
    unsigned whichone= SearchIndex(vt.index, map);
    if (whichone!= 0xffffffff)
    {
        long long newadd=ReadICBYTES(vt.data_file,vt.key,vt.index.O(3,whichone));
        ReadICBYTES(vt.data_file, vt.info, newadd);
        ICG_SetWindowText(SLE, &vt.key.C(1));
        ICG_SetWindowText(MLE, &vt.info.C(1));
    }
    else
    {
        ICG_printf(MLE, "NOT FOUND!\n");
    }
}
```

The key placed in the “key” matrix with the `GetText()` command is passed through the `KeyMapTR()` function and converted into numbers that we will search in the index matrix, these numbers are searched with the `SearchIndex()` function and the row of the index matrix is found. When found, the third element of that row (`vt.index(3, which one)`) gives us the address of that record in the data file. We first read the key itself, then the data, place them in the key and information matrices and then display them on the screen. Thus, the query is successfully completed. For example, when we write “AUGUST FERDINAND MÖBIUS” in the box above and press the “SEARCH” button, the following screen image will appear. As can be seen, the text in the first box is also returned to its previous state.



7. Advanced GUI Techniques

It is possible to write many different types of programs, even professional ones, using the features described so far. For example, if it is known how it works, an inventory recording program can be written by combining it with an accounting program or a database. However, there are some types of software where conventional programming techniques are not sufficient. The reason for this is that every event that occurs in the Windows operating system is transmitted to your program with a message. In other words, every time the user presses a key on the keyboard or every time the mouse is moved, a message is sent to the program you wrote. Your program works by constantly processing and interpreting these messages. If you take a break from this message processing marathon even for a moment, you will see your program freezes. So where are these messages processed?

Unfortunately, processing these messages requires very complex and long programs. Writing programs that can process these messages, each containing different purposes and parameters, requires a very good knowledge of the WIN32 API, the lowest programming interface of the Windows operating system. One of the biggest benefits of the I-See-Bytes library is that it handles these complex operations itself and provides the programmer with a clear, simple and logical set of functions, allowing her to evaluate the messages as she wishes.

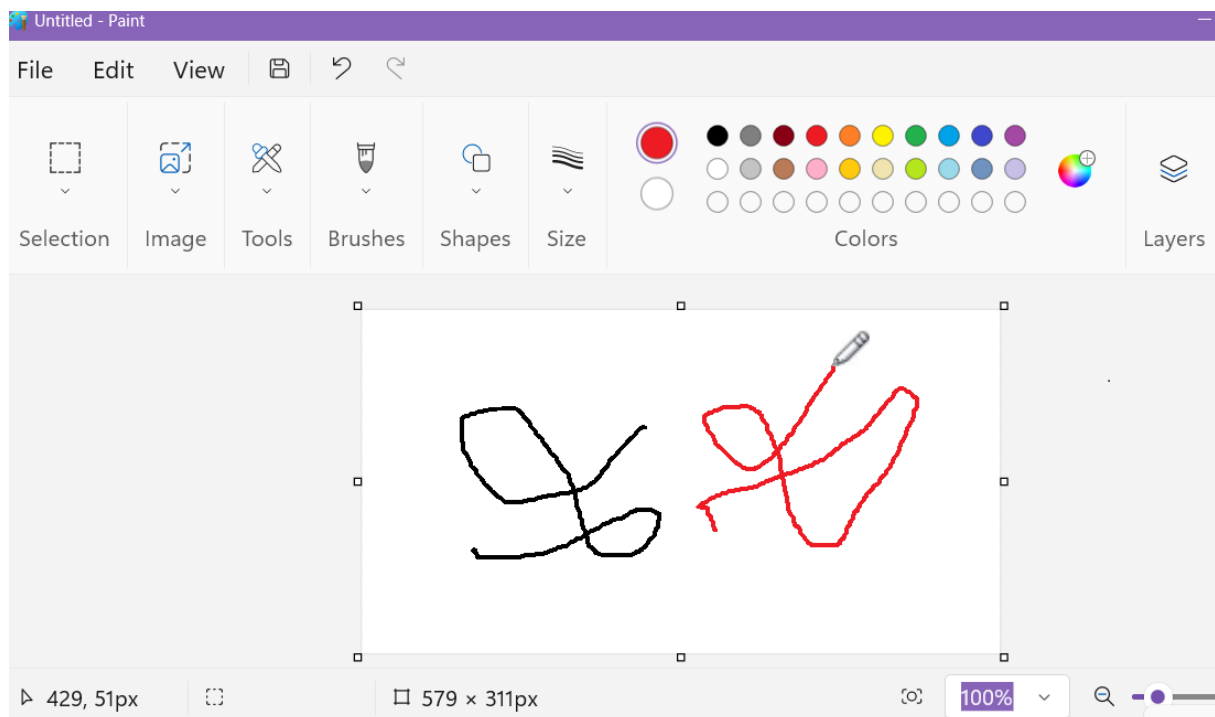
The biggest difficulty with this message structure in the Windows operating system occurs when we write a long-running program. For example, when a game program starts, it constantly draws animations on the screen. If you write an animation program that takes 5-10 minutes using the techniques we have learned so far, you will see that when this program runs, it does not receive any external reaction until it is finished. For example, you cannot move the window, minimize it, or even close it. While your code is constantly drawing something on the screen, it cannot process the messages sent by the operating system.

Game programs and video programs come first among the programs that create animations or draw images continuously. But it is not limited to these. Animations are now used in scientific software, financial programs that perform mathematical analysis, and many other places. In order to provide users with richer experiences, today's applications prefer to constantly reflect many visual and audio elements on the screen.

In this section, we will try to reach the technical maturity required to write such applications..

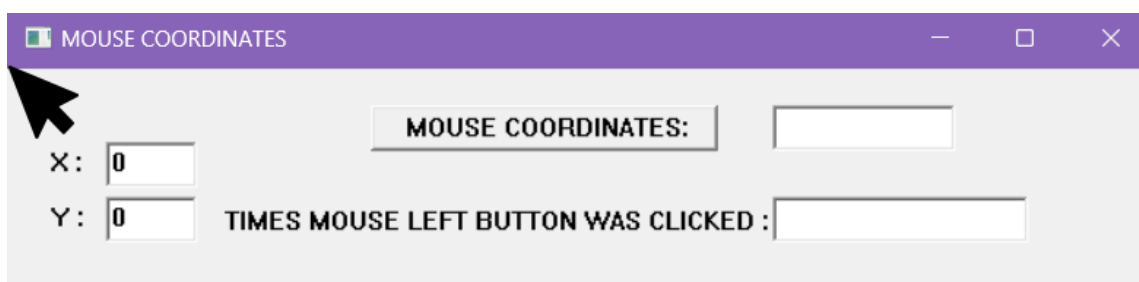
7.1. Accessing Mouse Coordinates

In most of the applications we are used to, the mouse is controlled by the operating system and there is no need for the programmer to process the incoming data. The program only becomes aware of this when the mouse hovers over a widget and interacts with that widget. However, in some special applications, the program must know the location of the mouse and activate special functions accordingly. The most well-known example of this is the Paint program in the Windows operating system. If we wanted to write such an application, we would need to constantly know the coordinates of the mouse cursor.

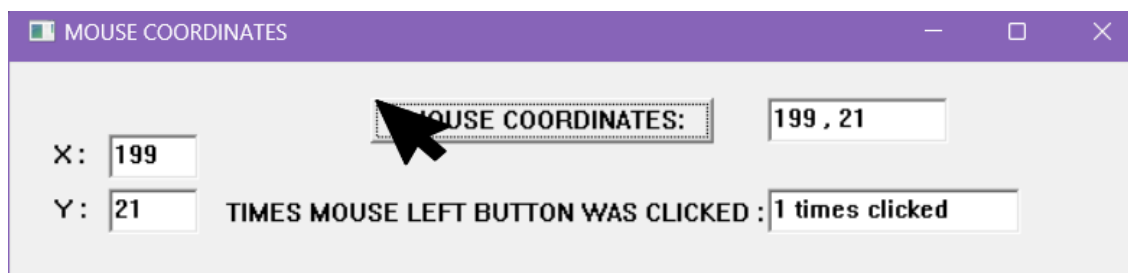


There are mechanisms in the I-See-Bytes library that allow us to access this data.

In the application whose screenshots are given below, we used functions that are called automatically when the mouse moves and also, we used functions that give us the coordinates of the mouse when we query. The first type of functions are called as soon as the mouse cursor enters the window borders of our application and are called again every time the mouse cursor moves. This way, our program can react as soon as the mouse cursor moves. The X and Y coordinates values at the far left of the window change with every mouse movement. In this example, the tip of the mouse cursor is moved to the upper left corner of our window. Therefore, the values of X and Y are zero.



In the example below, the tip of the mouse cursor was placed on the upper left corner of the button labeled "MOUSE COORDINATES", the left mouse button was clicked and this button was pressed. As we will see shortly, the coordinates of the upper left corner of this button are selected as (199,20) in our source code. This shows us that the coordinate information provided by Windows is not 100% consistent. The following button function is called when the button is clicked:



```
void MouseCoor()
{
    char ch[16];
    sprintf_s(ch, 16, "%d , %d", ICG_GetMouseX(), ICG_GetMouseY());
    ICG_SetWindowText(SLEF, ch);
}
```

As seen here, the `ICG_GetMouseX()` and `ICG_GetMouseY()` functions can return the X and Y coordinates of the mouse whenever we want.

We learned which functions to call to instantly query mouse coordinates. So how are the X: and Y: coordinates, which we see on the left side of the screenshots above, changing automatically when the mouse is moved, created? To find out, we have to go through the entire code.

```
#include "icb_gui.h"
#include <stdio.h>

int SLEX, SLEY, SLEF, SLESAY;

void ICGUI_Create()
{
    ICG_MWTitle("MOUSE COORDINATES");
    ICG_MWSize(650, 170);
}

void WhenMouseMoves(int x, int y)
{
    char ch[8];
    _itoa_s(x, ch, 8, 10);
    ICG_SetWindowText(SLEX, ch);
    _itoa_s(y, ch, 8, 10);
    ICG_SetWindowText(SLEY, ch);
}

void MouseCoor ()
{
    char ch[16];
    sprintf_s(ch, 16, "%d , %d", ICG_GetMouseX(), ICG_GetMouseY());
    ICG_SetWindowText(SLEF, ch);
}

void WhenMouseLeftClicked()
{
    static int count= 0;
    count++;
    char ch[32];
    sprintf_s(ch, 32, "%d many times was clicked", count);
    ICG_SetWindowText(SLESAY, ch);
}

void ICGUI_main()
{
    ICG_Static(25, 43, 20, 25, "X :");
    ICG_Static(25, 73, 20, 25, "Y :");
    SLEX = ICG_SLEditSunken(55, 40, 50, 25, "");
    SLEY = ICG_SLEditSunken(55, 70, 50, 25, "");
    ICG_Button(200, 20, 190, 25, " MOUSE COORDINATES:", MouseCoor);
    SLEF = ICG_SLEditSunken(420, 20, 100, 25, "");
    ICG_SetOnMouseMove(WhenMouseMoves);
}
```

```
ICG_Static(120, 75, 320, 25, "TIMES MOUSE LEFT BUTTON WAS CLICKED :");
SLESAY = ICG_SLEditSunken(420, 70, 140, 25, "");
ICG_ICG_SetOnMouseDown(WhenMouseLeftClicked);
}
```

When we examine this code, we see that the function called `WhenMouseMoves()`, which is must be executed everytime the mouse moves within the `ICG_Main` function, is declared to the library as follows:

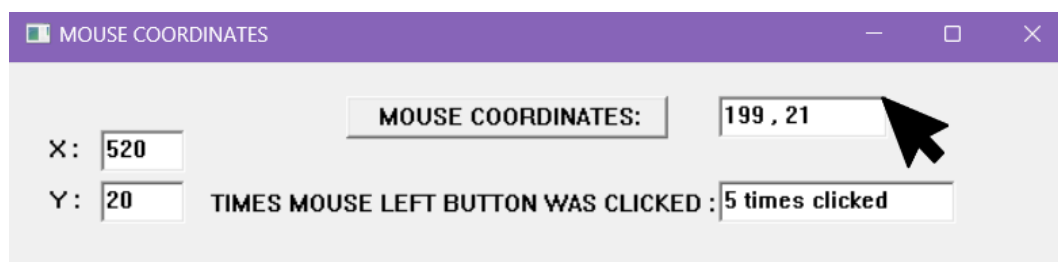
```
ICG_SetOnMouseMove(WhenMouseMoves);
```

After this function is called, the function named `WhenMouseMoves(int x, int y)` will be called automatically whenever the mouse position changes. The inputs of this function are the position coordinates of the mouse cursor. Inside the function, we see that all that is done is to print these coordinates in the text boxes.

Likewise, if there is a function that we want to be called every time the left mouse button is clicked, we specify it in the library using the `ICG_SetOnMouseLClick()` function.:

```
ICG_SetOnMouseLClick(WhenMouseLeftClicked);
```

When we examine the code of the `MouseLeftClick()` function, we see that all this code does is count the number of times the mouse has been clicked and print it in the text box attached to the `SLESAY` handle. Below is the screenshot where, after clicking the left mouse button five times, the cursor moves to the upper right corner of the text box attached to the `SLEF` handle. Considering that the upper left corner of this text box was created at the coordinates (320,20) and its width is 120 pixels, we can calculate that the upper right corner should also be at the coordinates (420,20). This is almost the same as the value we read in the X: and Y: boxes.

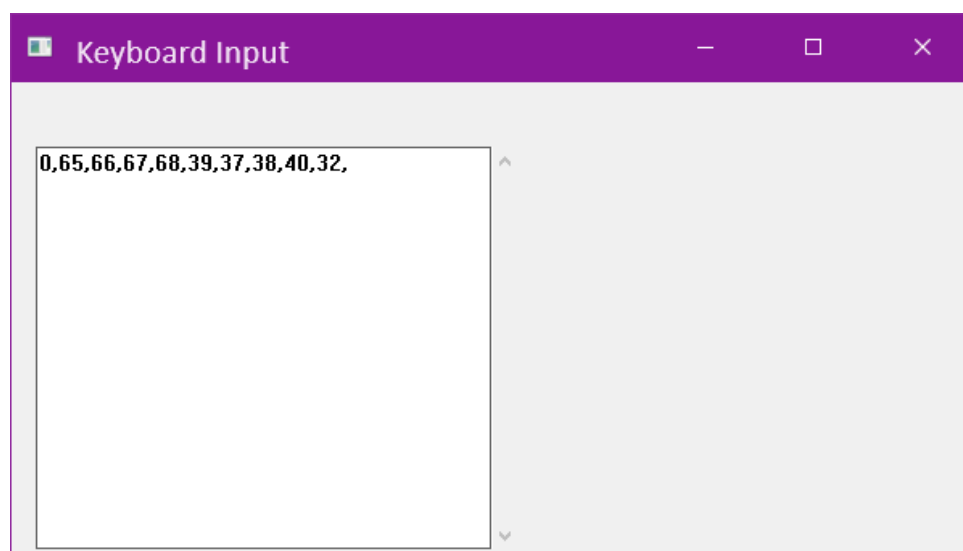


7.2. Keyboard Input

One of the last things programmers have to worry about is trying to figure out what keys are being pressed on the keyboard. The operating system already does this for us. When we create an edit box, writing letters into it is one of the basic tasks performed by the operating system. All the programmer has to do is copy the words written in the edit box into a memory area. What if there is no edit box? Or what if input is made from the keyboard while the edit box is not selected?

This type of programming requirement is most common when programming games. Even though there is no text box on the screen in a game program, the user continues to press the keyboard keys to control the game and shoot left and right. In this section, we will see how to process this information coming from the keyboard.

When you first run the application below, you will see that "0" is written in the text box. Then, if you press the A, B, C, D right arrow, left arrow, up arrow, down arrow and space keys on the keyboard, respectively, you will get the same image as below. The point we need to draw attention to here is that each key on the keyboard will always send the same number. That is, uppercase letter A and lowercase letter A will return the same number (65). In other words, pressing the "Caps Lock" key does not change anything. However, we cannot say the same for the numeric pad (if exists). The numeric pad, the calculator-like keys on the far right of most keyboards, returns different values depending on whether the "Num Lock" key is on or off. Therefore, if you are going to use these keys in your game program, you should take precautions to ensure that this key is on or off.



Below is the source code of this application. The only command that stands out in this code is the `ICG_SetOnKeyPressed()` command. This function defines which function will be called when the user presses any key on the keyboard. The name of this function is chosen as `Keyboard Press(int k)` in this example. The parameter (`k`) that the `ICBYTES` library adds to this function is the numerical code of the key pressed on the keyboard. All this function does is print these numbers into the text box using the `MLE` handle.

```
#include "icb_gui.h"

int MLE;

void ICGUI_Create()
{
    ICG_MWTitle("Keyboard Input");
    ICG_MWSize(620, 350);
}

void WhenKeyPressed(int k)
{
    ICG_printf(MLE, "%d,", k);
}

void ICGUI_main()
{
    MLE = ICG_MLEdit(15, 40, 300, 250, "", SCROLLBAR_V);
    ICG_SetOnKeyPressed(WhenKeyPressed);
}
```

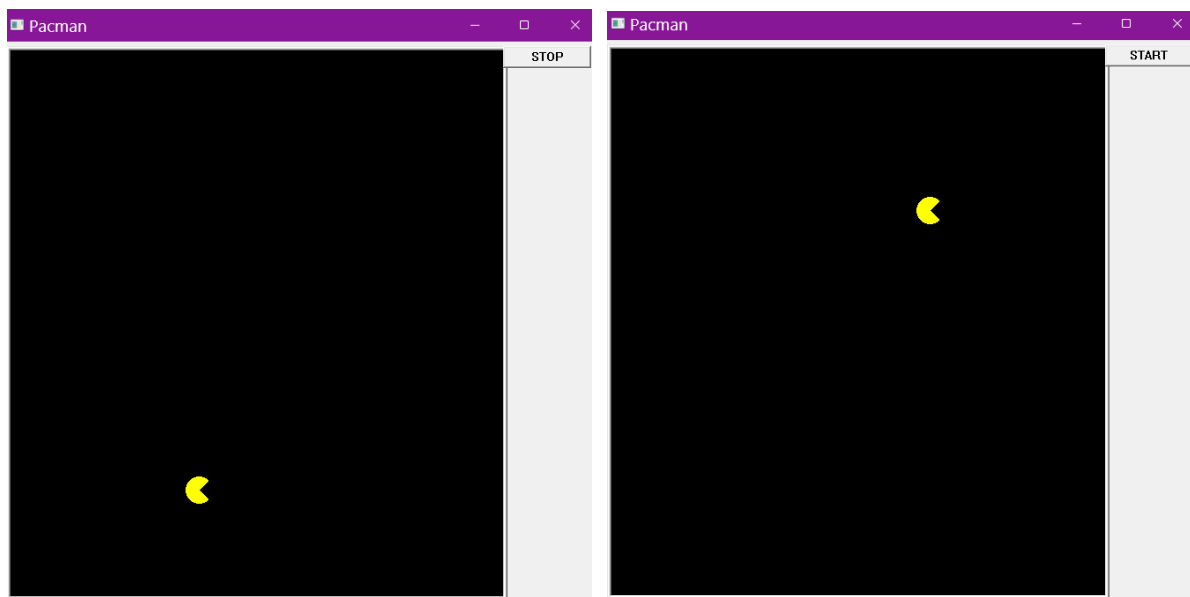
7.3. Continue to Pacman

In the fifth chapter of this book, we learned how to draw the graphics of the Pacman game. We also made a short animation of Pacman by repeatedly drawing, erasing and sliding it. In this topic, in addition to making the animation continuous, we will learn how to control Pacman's movements from the keyboard.

As we mentioned before in this section, in order for the application to run without freezing in the Windows operating system, the messages sent by the operating system must be constantly processed. For this reason, we cannot set up loops that take too long because our program cannot process messages while these loops are running. So, in such a situation, how do we run animations that

last minutes or even hours? The answer is with concurrent programming techniques. The purpose of this book is not to teach the reader concurrent programming techniques. However, we will content ourselves with giving a simple example of how to combine the concurrent programming technologies in the ICBYTES library and the Win32 API. Although it is simple, readers who wish can program the entire Pacman game by improving this example.

If we can turn the animation that makes Pacman slide and draw into a function that runs parallel to our main program, we can easily create our animation while our main program processes the messages coming from Windows. We call programs in which parallel functions run simultaneously in multithreaded programs. Each function that works parallel to each other is called a thread. Windows operating system uses a function called `CreateThread()` to create parallel functions. You can start running this function in parallel by giving the name of the function you want to run in parallel as a parameter to this function called `CreateThread()`. When the "START" button is pressed in our program, we will start our animation using the `CreateThread()` function. Below are screenshots of our application.



In this application, after pressing the "START" button, the blonde Pacman is moved to the right and up using the arrow keys. If a maze had been drawn in the background, an animation of Pacman wandering around the maze could have been obtained.

The source code of this application is shown below.

```
#include "icb_gui.h"

void ICGUI_Create()
{
    ICG_MWTitle("Pacman");
    ICG_MWSize(700, 700);
}

int thread_on = 0, keyboardinput = 0;
int FRM1, BTN;

void* Pacman(PVOID lpParam)
{
    ICBYTES field;
    CreateImage(field, 560, 620, ICB_UINT);
    int x = 230, y = 500;
    while (thread_on)
    {
        FillCircle(field, x - 15, y, 15, 0xffff00);
        HalfRect(field, x, y, -15, -15, 0);
        HalfRect(field, x, y, -15, 15, 0);
        DisplayImage(FRM1, field);
        Sleep(10);
        FillRect(field, x - 30, y - 15, 30, 30, 0);
        if (keyboardinput == 39) //if right arrow pressed
            x++;
        else if (keyboardinput == 37) //if left arrow pressed
            x--;
        else if (keyboardinput == 38) //if down arrow pressed
            y--;
        else if (keyboardinput == 40) //if up arrow pressed
            y++;
    }
    return(NULL);
}

void WhenKeyboardPressed(int k)
{
    keyboardinput = k;
}
```

```

void Start()
{
    DWORD dw;
    if (thread_on == 0)
    {
        thread_on = 1;
        ICG_SetWindowText(BTN, "STOP");
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Pacman,
NULL, 0, &dw);
    }
    else
    {
        thread_on = 0;
        ICG_SetWindowText(BTN, "START");
    }
    SetFocus(ICG_GetMainWindow());
}

void ICGUI_main()
{
    BTN = ICG_Button(570, 5, 100, 25, "START", Start);
    FRM1 = ICG_FrameMedium(5, 5, 560, 620);
    ICG_SetOnKeyPressed(WhenKeyboardPressed);
}

```

Since most of this code contains examples that we have seen before, we will only focus on the new parts. In the Start() function, it is first checked whether the global string_open variable is zero or one, its value is reversed and the text of the "START" button is converted to "END". If it changes from zero to one, the function called Pacman(), which we want to run in parallel, is started using the CreateThred() command.

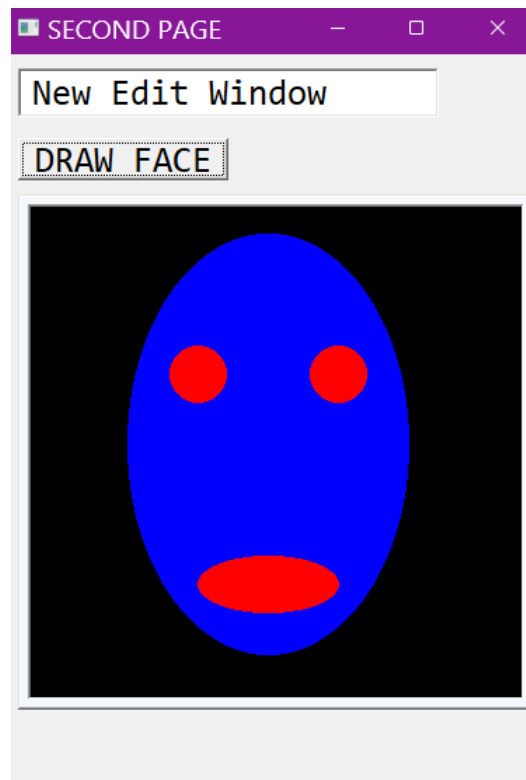
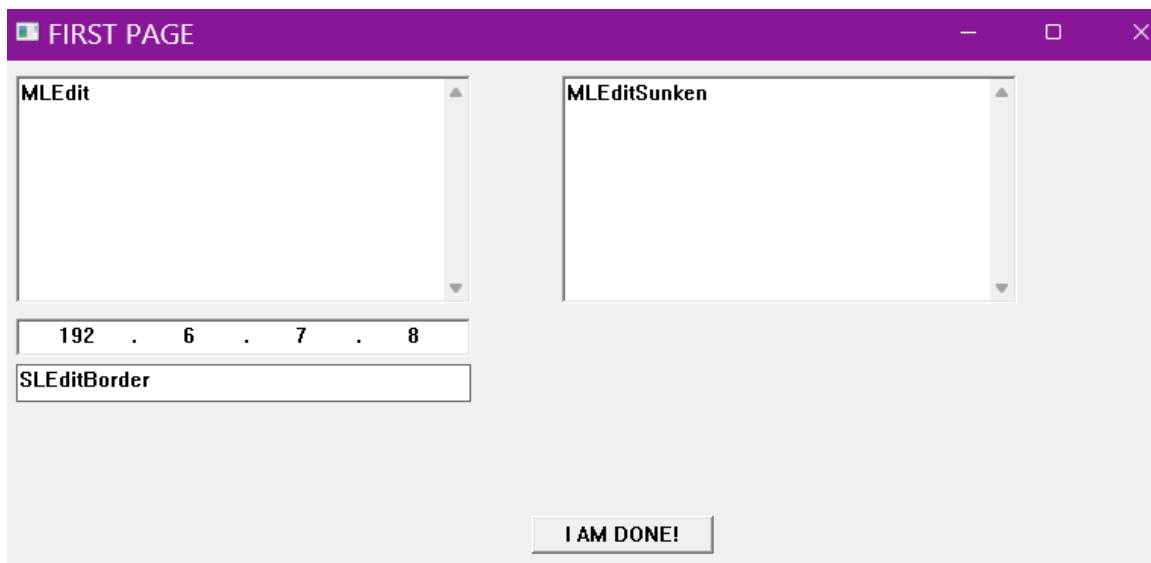
When we focus on the Pacman() function, where the entire animation takes place, we see that this function first creates an image called "field" on which it can draw. Then, it is seen that the coordinates of the point where Pacman will first appear are determined as $x = 230$ and $y = 500$.

The animation part of the work is done in an endless while() loop. This while loop will continue as long as the global string_open variable is equal to one. It can be seen that (x,y) coordinates are used in the loop and Pacman is drawn first and deleted after 10 milliseconds. This part is no different from what is explained in the fifth part. The only difference is that instead of constantly increasing the x coordinate, the x and y coordinates are increased or decreased according to the last arrow key entered from the keyboard. Since the value read from the

keyboard is constantly written to the global variable called `keyboardinput`, it can be easily read from the `Pacman()` function.

7.4. Window Morphing

When working with complex applications, just one window may not be enough. After completing operations in one window, it may be necessary to switch to a second window. In such cases, you need to transform the main window of the application. Below are two windows of such application shown.



There are multiple methods for switching to different windows in Windows. The simplest of those methods will be explained here. In this method, when we are done with the existing widgets in the first window, we destroy all the widgets in the first window by pressing a button and moving to a new window, and then new widgets are recreated in the second window. The source code of this application is shown below.

```
#include"icb_gui.h"
void ICGUI_Create()
{
    ICG_MWSize(800, 400);
    ICG_MWTitle("FIRST PAGE");
}

int MLE1, MLE2, SLE1, SIP1, BTN1; //handles of the first page
int SLE2, FRM1, BTN2; //handles of the second page

void DrawFace()
{
    ICBYTES resim;
    CreateImage(resim, 350, 350, ICB_UINT);
    FillEllipse(resim, 70, 20, 100, 150, 0xff);
    FillEllipse(resim, 200, 100, 20, 20, 0xff0000);
    FillEllipse(resim, 100, 100, 20, 20, 0xff0000);
    FillEllipse(resim, 120, 250, 50, 20, 0xff0000);
    DisplayImage(FRM1, resim);
}

void NewWindow()
{
    ICG_DestroyWidget(MLE1);
    ICG_DestroyWidget(MLE2);
    ICG_DestroyWidget(SIP1);
    ICG_DestroyWidget(SLE1);
    ICG_DestroyWidget(BTN1);
    ICG_MWSize(400, 600);
    ICG_MWTitle("SECOND PAGE");
    ICG_SetFont(30, 0, "Consolas");
    SLE2 = ICG_SLEditSunken(10, 10, 300, 35, "New Edit Window");
    BTN1 = ICG_Button(10, 60, 150, 30, "DRAW FACE", DrawFace);
    FRM1 = ICG_FrameThick(10, 100, 350, 350);
}

void ICGUI_main()
{
    MLE1 = ICG_MLEditSunken(10, 10, 300, 150, "MLEdit", 1);
    MLE2 = ICG_MLEditSunken(370, 10, 300, 150, "MLEditSunken", SCROLLBAR_V);
    SIP1 = ICG_IPAddressSunken(10, 170, 300, 25, 0xc0060708);
    SLE1 = ICG_SLEditBorder(10, 200, 300, 25, "SLEditBorder");
    BTN1 = ICG_Button(350, 300, 120, 25, "I AM DONE!", NewWindow);
}
```

As seen in this code, the design of the first page of the application is made in the usual way within the `ICG_main()` function. What's different is what happens inside the function called `NewScreen()` called by the "I AM DONE" button.

In this function, we encounter a new command called `ICG_DestroyWidget()`. This command is used to destroy the widgets we created in `ICG_main`. In other words, it destroys text editing boxes and buttons by using their handles. In this way, it allows us to create new boxes on our window and completely change the appearance of our application.

After removing the boxes, we can continue on our way by using the `ICG_MWSize()` commands that change the size of the application window, which we are used to using in the `ICGUI_Create()` function, or the `ICG_MWTitle()` commands that change the name of the application, or even by changing the font. On the second page, a different picture frame is created, and when the button is pressed, the `DrawPicture()` function, which draws a picture in this frame, is called.

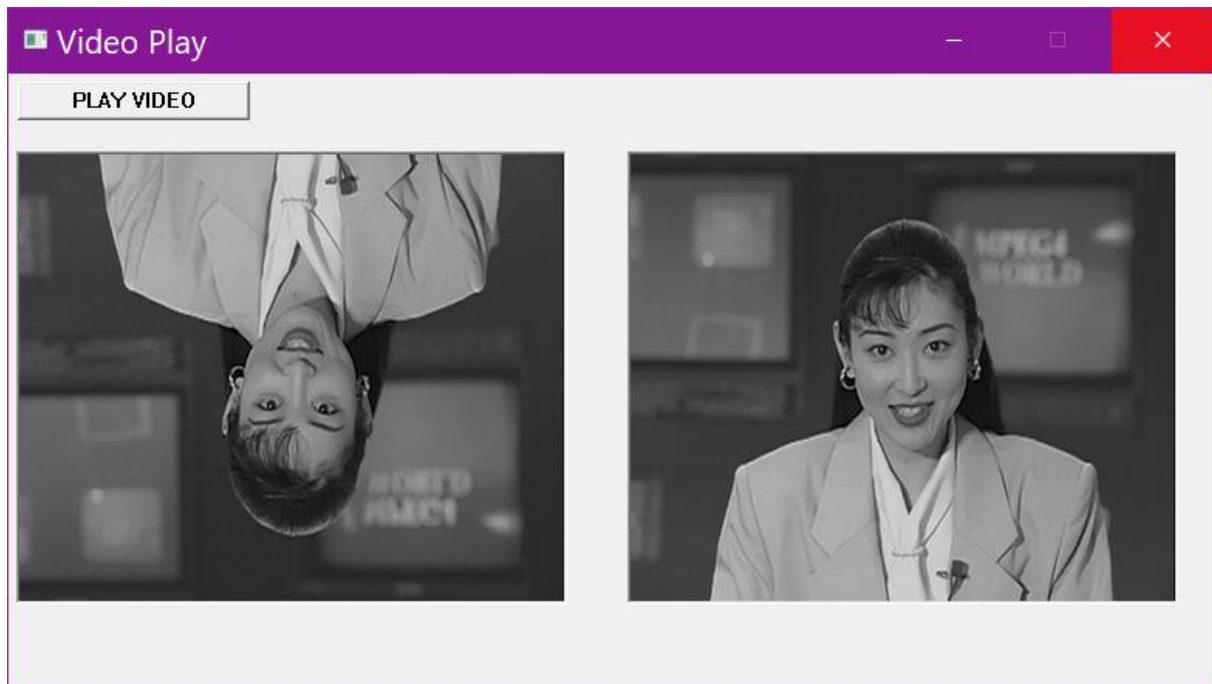
7.5. Playing Video

In introductory video processing courses, students often work with uncompressed video formats. Such uncompressed video formats are used as reference data in many scientific studies. The most well-known of these formats is the YUV (.yuv) format. Video examples frequently used in such articles can be downloaded from the following website.

<https://media.xiph.org/video/derf>

The videos and images on this website have been used in numerous scientific articles. It is very easy to open YUV format videos with the I-See-Bytes library. To do this, first download one of these videos (for example, `akiyo_cif.yuv` file) and copy it to the working folder of your project. Then define this video file as a device using the `CreateFileReader()` function. Then, you need to read each frame in the video file into the image matrix one by one and display it on the screen.

Below is a screenshot of our application that performs these operations.



In YUV format, the brightness information of each image is written first. CIF format is used to name images with dimensions of 352x288 pixels. After Y, which is the brightness information, come the U and V layers, which are the color information. However, each of the color layers is one quarter of the brightness information in this format, that is, 176x144 pixels in size. Each image in the video must appear on the screen every 29.97 milliseconds. We will only show brightness information in this application because almost all of the basic image processing algorithms are applied only on brightness (grayscale) images. Below is the code for this application.

```
#include "icb_gui.h"

int FRM1, FRM2;

void ICGUI_Create()
{
    ICG_MWTitle("Video Play");
    ICG_MWSize(800, 450);
}

void Show()
{
    ICDEVICE file;
    ICBYTES Y, U, V, InvertedY;
    CreateImage(Y, 352, 288, ICB_CHAR);
    CreateMatrix(U, 176, 144, ICB_CHAR);
    CreateMatrix(V, 176, 144, ICB_CHAR);
```

```

if (!CreateFileReader(file, "akiyo_cif.yuv"))
    return; //file could not be opened therefore we quit
while (ReadIntoMatrix(file, Y) == (352 * 288))
{
    ReadIntoMatrix(file, U);
    ReadIntoMatrix(file, V);
    DisplayImage(FRM1, Y);
    FlipV(Y, InvertedY);
    DisplayImage(FRM2, InvertedY);
    Sleep(30);
}
CloseDevice(file);
}

void ICGUI_main()
{
    ICG_Button(5, 5, 150, 25, "PLAY VIDEO", Show);
    FRM1 = ICG_FrameSunken(5, 50, 360, 300);
    FRM2 = ICG_FrameSunken(400, 50, 360, 300);
}

```

There is no need to explain this entire source code. All important operations take place within the Show() function, which is called when the “VIDEO PLAY” button is pressed. In this function, a device (ICDEVICE) whose name is a file is defined. This device is used to access the "akiyo_cif.yuv" file on the hard disk. The function that does this,

```
CreateFileReader(file, "akiyo_cif.yuv")
```

It connects the “file” device to this file with the command. Then, a matrix is created for each layer of color information: Y, U, V. These layers are located in the video file as follows:



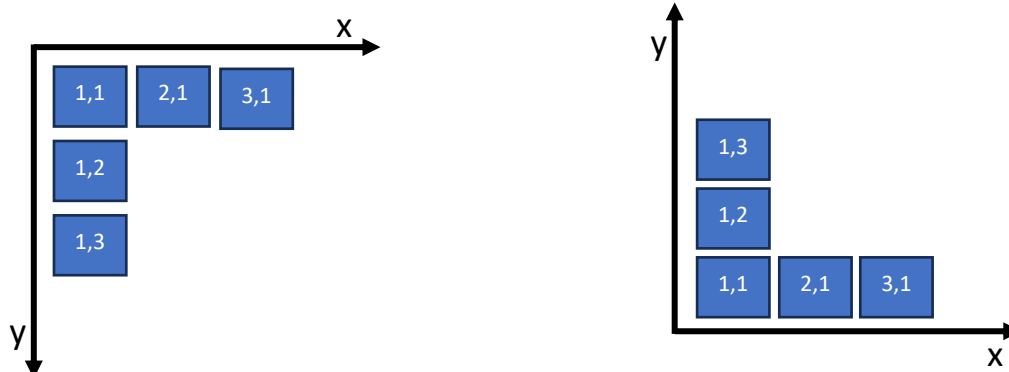
We use the ReadIntoMatrix(file, Y) function to read layer Y. This function returns the number of bytes it read. Therefore, if the Y layer reads less than 352x288 bytes, it means we have reached the end of the video file. After reading

layer Y, when we show it in picture frame-1, we see that the picture is printed upside down. The reason for this is that the (y) dimension, which is the height dimension on the graphics card, increases from bottom to top. However, in a matrix, this dimension increases from top to bottom. Therefore, when we turn this image upside down using the FlipV() function and print it on the screen in the second frame, the image is corrected. After printing the Y layer on the screen, we need to wait 30 milliseconds and read the second frame. However, before reading the second Y layer, we need to read the U and V layers. That's why the while loop is written as follows:

```
while (ReadIntoMatrix(file, Y) == (352 * 288))
{
    ReadIntoMatrix(file, U);
    ReadIntoMatrix(file, V);
    DisplayImage(FRM1, Y);
}
```

That is, the U and V layers are read immediately after the Y layer is read, but those layers are not shown on the screen.

The disadvantage of this method is the processing power required to flip the image matrix. In the ICBYTES library, the coordinate system of a normal image matrix is top to bottom as shown below on the left. However, when the pixel coordinate system was first created, the coordinate system from analytic geometry was used instead of the matrix coordinate system. This is why many image sources have the pixels upside down as shown on the right.



To create an image matrix that can address this situation, there is a command in the ICBYTES library that can create an inverted image:

```
CreateFlippedImage(...)
```

The images you create using this function are displayed inverted using the commands given to the graphics card, **without using extra processor power.**

However, do not forget that the coordinate system is different.

Using this command, we can display Akiyo correctly without flipping her, by making a small change in the “Show()” function as shown below in our previous program.

```
void Show()
{
    ICDEVICE file;
    ICBYTES Y, U, V, InvertedY;
    CreateFlippedImage (Y, 352, 288, ICB_CHAR);
    CreateMatrix(U, 176, 144, ICB_CHAR);
    CreateMatrix(V, 176, 144, ICB_CHAR);
    if (!CreateFileReader(file, "akiyo_cif.yuv"))
        return; //file could not be opened therefore we quit
    while (ReadIntoMatrix(file, Y) == (352 * 288))
    {
        ReadIntoMatrix(file, U);
        ReadIntoMatrix(file, V);
        DisplayImage(FRM1, Y);
        //FlipV(Y, InvertedY);
        //DisplayImage(FRM2, InvertedY);
        Sleep(30);
    }
    CloseDevice(file);
}
```

QUESTION: While the Y layer was created as an image matrix, the U and V matrices were created as a plain matrix. Why?

EXERCISE: Create a color image in RGB type from the Y, U and V layers you have and display it on the screen.

7.5.1 Fast Image Processing

Most image processing algorithms require multiple accesses to pixels. Considering that there are millions of pixels in a single image, accessing a single pixel must be very fast. The methods proposed so far for pixel access in the I-See-Bytes library are very slow. For example, in the example above, to access a pixel at coordinate (x,y) in the Y matrix:

Y.B(x,y)

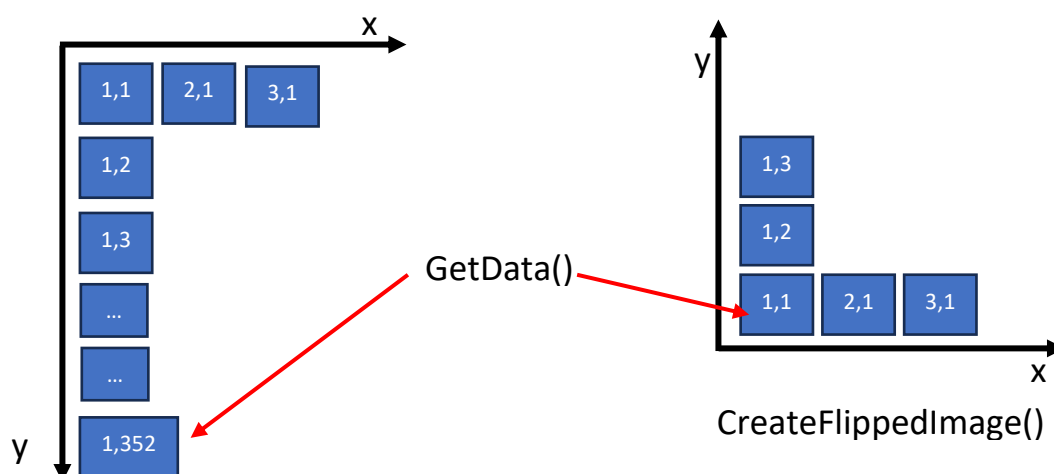
expression is enough. Although this line is a very elegant, well-expressed command that makes it easier for the user to write code, considering the limitations of the C++ language, it is actually an inefficient method. A pixel is actually a memory location described by an address in the computer. The Y.B(x,y)

command above is actually a function that calculates the address of the pixel at the (x,y) coordinate. Therefore, every time you use this command a function is called and since the function call process causes a lot of extra processing for your computer, while normally 3 operations are needed to reach a pixel, here much more may be needed (especially in Debug mode).

A faster way to do this is to access the address of the matrix directly. You can use the following command to do this. Let's say we want to make the value of the pixel at the coordinate (x-1,y-1) of the previous Y matrix 128:

```
unsigned char *c= (unsigned char *c)GetData(Y);  
c[352*y+x]=128;
```

Let's not forget that in C++ the address starts from zero. However, in matrices the index starts from (1,1). In other words, the address (1,1) for the matrix is equal to (0,0) for C++. The number 352 used here is the horizontal dimension of the Y matrix, that is, the width of the image. Another important point is that the address returned by the GetData() function is for the inverted image. In other words, it is the address of the lower left corner of the matrix, as shown below.



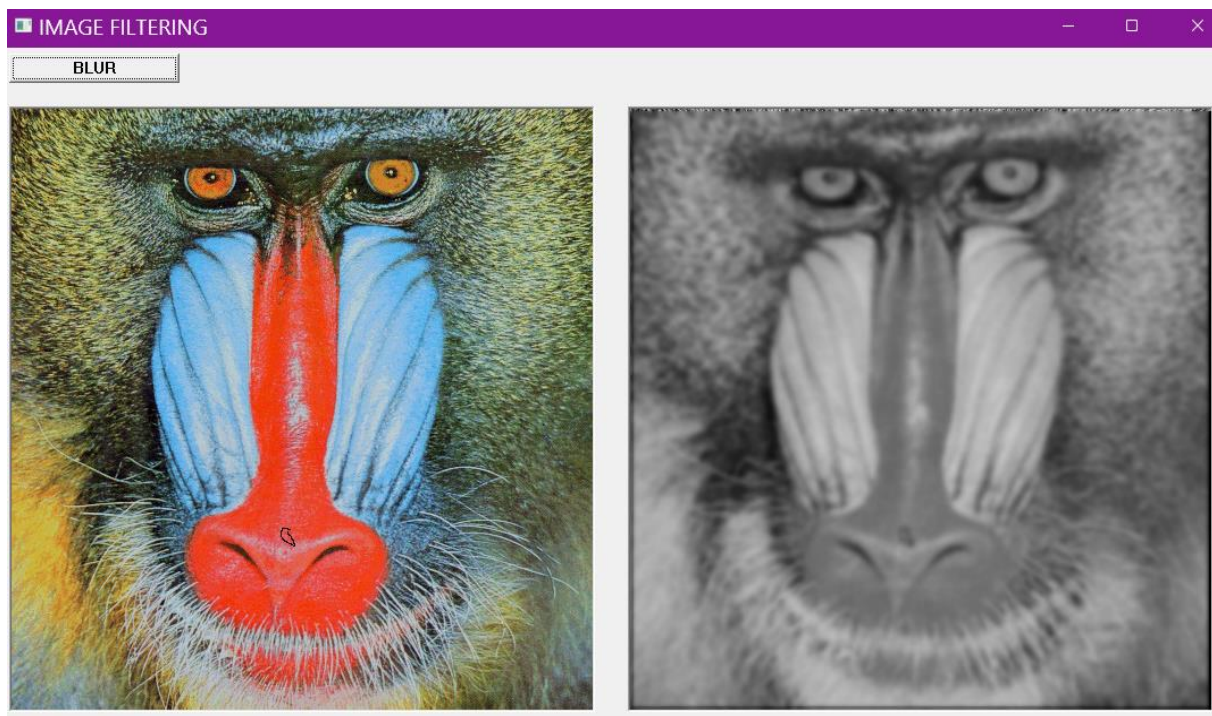
!!! IMPORTANT WARNING !!!

If you make a wrong calculation and try to reach outside the matrix while trying to reach the pixels in this way, your program may crash or read the wrong pixel values. **DO NOT ACCESS THE PIXELS THIS WAY UNLESS YOU ARE AN EXPERT C++ PROGRAMMER.**

7.6. Image Filtering

One of the first topics covered in introductory image processing courses is image filtering because this process is the first step of many other algorithms. For example, the Canny edge detection algorithm requires that the image be first processed with a low-pass filter. Low-pass filtering is also used in privacy protection applications because it reduces noise in the image and blurs it. For example, a hospital software may need to blur a patient's body image when displaying it.

Another application is image enhancement. While low-pass filters purify the picture from noise and distortions, high-pass filters increase the clarity (sharpness) of the picture. Since the subject of this book is not image processing, we will not go into further detail. Below is an application that blurs an image using a low-pass filter.



Above you see the “Mandrill.jpg” image, which is frequently used in image processing articles and lectures. You can easily find this picture on the internet. If you pay attention to the monkey's fur in the grayscale photo on the right, you can see that it is clearly blurred compared to the one on the left. Below is the code for this application.

```

#include "icb_gui.h"

int FRM1, FRM2;

void ICGUI_Create()
{
    ICG_MWTitle("IMAGE FILTERING");
    ICG_MWSize(1100, 650);
}

void Filter()
{
    ICBYTES RGB, grey, filtered;
    ICBYTES filter_coef = { 0.1,0.2,0.4,0.2,0.1 };
    ReadImage("mandril.jpg", RGB);
    DisplayImage(FRM1, RGB);
    Luma(RGB, grey);
    for (int x = 0; x < 5; x++)
    {
        FilterH(grey, filtered, filter_coef, ICB_UCHAR);
        FilterV(filtered, grey, filter_coef, ICB_UCHAR);
    }
    DisplayImage(FRM2, grey);
}

void ICGUI_main()
{
    ICG_Button(5, 5, 150, 25, "BLUR", Filtrele);
    FRM1 = ICG_FrameSunken(5, 50, 360, 300);
    FRM2 = ICG_FrameSunken(550, 50, 360, 300);
}

```

It seems that everything happens within the function called `Filter()`. First, the coefficients of our low-pass filter are placed into the vector named "filter_coef". The image file named Mandril is then read and displayed in the left frame attached to the FRM1 handle. Since filtering can only be done on the brightness component of the image, the image was converted to grayscale using the `Luma()` function. It was then filtered horizontally and vertically five times in a for loop. The `FilterH()` function used here filters the image in the "grey" fluid horizontally using the "filter" vector and writes it to the "filtered" fluid:

```
FilterH(grey, filtered, filter_coef, ICB_UCHAR);
```

It wants the result to be "unsigned char", that is, in bytes. Then, the `FilterV()` function filters the resulting image vertically and throws the result back into the fluid named "gray". Thus, when we return to the beginning of the cycle, the

image filtered horizontally and vertically is filtered again. The image filtered five times in this way is shown in the frame on the left attached to the FRM2 handle.

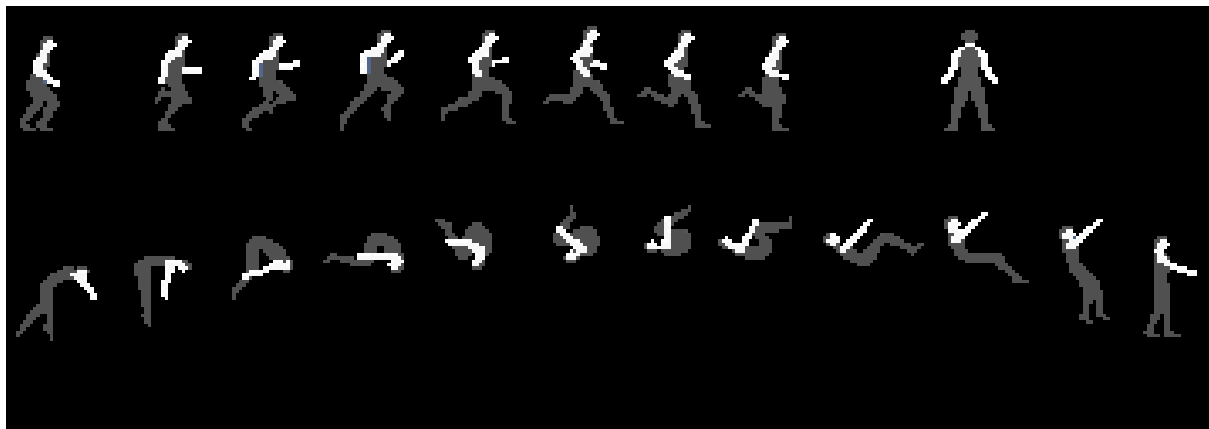
In this example, a low-pass filter with simple parameters is used. Those who are curious can learn what the parameters of a high-pass filter should be and write an application that makes the picture clearer than it is. All they need to do is change the parameters shown in the line below accordingly:

```
ICBYTES filter_coef = { 0.1,0.2,0.4,0.2,0.1 };
```

7.7. Advanced Animation Techniques

Graphic animations are used in many applications. One of the places where these are used most is games. Moving objects in games are seen from different angles when they change direction, or they perform enlargement and shrinkage animations when they move closer or further away. In this section, we will explain how to write the Commodore 64 computer game called "Impossible Mission", which became very famous in the 1980s.

In this game, you control an agent who tries to find the key pieces of the code that will stop the bomb that will blow up the world in the underground shelter of a mad scientist. Animated movie frames of this agent are shown below.



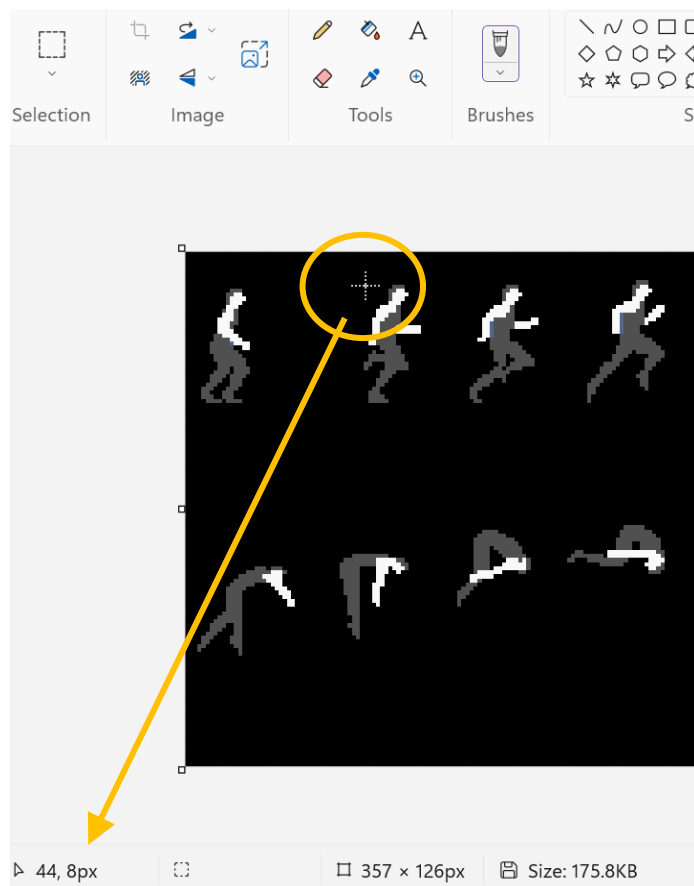
Pictures containing such game hero animated movie frames are called "sprite sheets". Such sprite sheets are abundantly published on the Internet by game programmers or arcade game fans. What we need to do is to load each animation frame as a separate image matrix. If we can copy each frame from the picture above one by one and then paste them sequentially, we can get an animation. For this purpose, there are functions such as Copy(), Paste() and PasteNone0() (paste non-zero places) in the ICBYTES library.

The function below shows us a code example that loads the frames required for our agent's running animation into 7 separate ICBYTES matrices.

```
ICBYTES AgentRun[7];

void LoadAgentRun()
{
    ICBYTES coordinat{ {45, 9, 13,30},{71,9,16,30},{100,8,20,30},
{130,8,23,30},{160,7,25,30},{189,8,22,30},{218,8,25,31} };
    for (int i = 1; i <= coordinat.Y(); i++)
    {
        Copy(Agent, coordinat.I(1,i), coordinat.I(2, i),
coordinat.I(3, i), coordinat.I(4, i), AgentRun[i-1]);
        PasteNon0(AgentRun[i-1], 33*i, 58, Corridor);
    }
    ICBYTES TV;
    MagnifyX3(Corridor, TV);
    DisplayImage(F1, TV);
}
```

At first glance, we see the AgentRun[7] array, which is a variable in ICBYTES defined as a global array. This directory is used to keep each frame image of the running agent in order. Inside the function, we see a 4x7 integer matrix called "coordinate". What are these coordinates? If we open the sprite sheet above

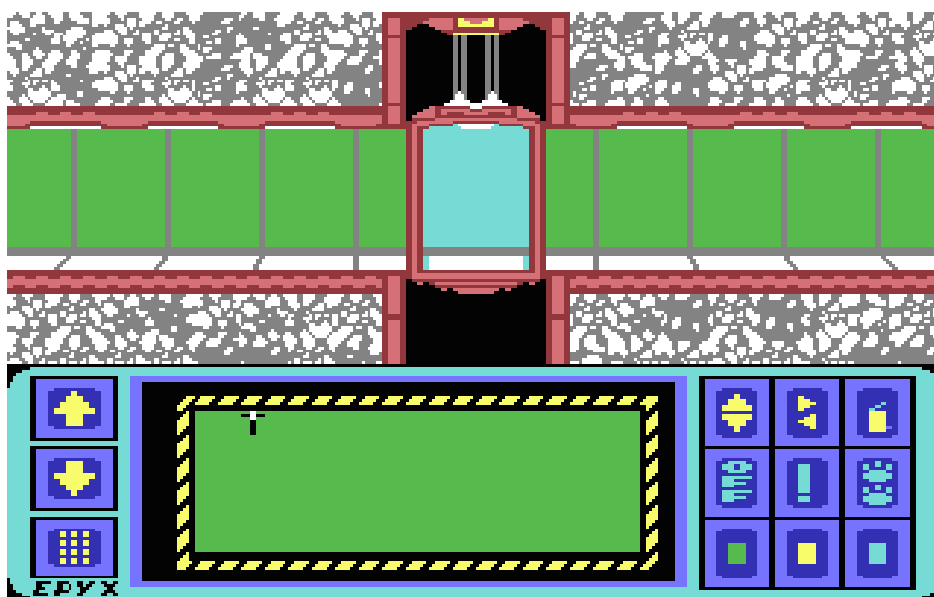


with the program called Paint in Windows and move the mouse cursor to the upper left corner of the first frame of the running agent as shown on the side, we will see the x and y coordinates of that point on the image in the lower left corner of the Paint program. However, in the Paint program, the upper left corner coordinate of the image starts from (0,0). On the other hand, in matrices, the upper left corner coordinate starts from (1,1). Therefore, in the code above, instead of (44,8) the first two coordinates are:

```
ICBYTES coordinat{ {45, 9, 13, 30},
```

13 and 30 are the width and height (in pixels) of the first frame of the running agent. From here, it is understood that the remaining elements of the matrix named "coordinat" are the x and y coordinates and widths and height of the other frames of the running agent.

In the remaining lines of the function named LoadAgent(), 7 film frames are cut from the sprite sheet and pasted into the image named "Corridor" in a for loop. It is not shown here, but the ICBYTES fluid named "Corridor" actually carries the following image:



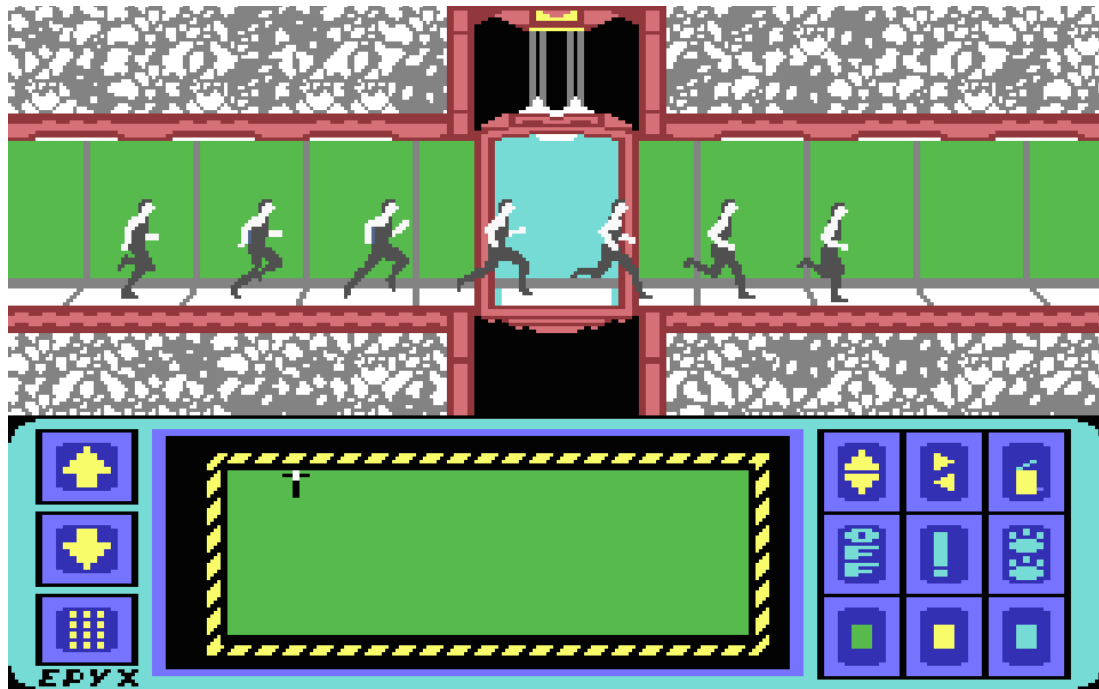
```
PasteNon0(AgentRun[i-1], 33*i, 58, Corridor);
```

With this command, while the agent is running, the movie frames are pasted sequentially to the picture above, 33 pixels apart. However, the background image in the "sprite sheet" image containing the agent's images is black. In the image PasteNone0() function pastes, pixels whose color is completely black, that is, all red, green and blue (KYM) channels are zero, are not pasted. Later,

```
ICBYTES TV;  
MagnifyX3(Corridor, TV);  
DisplayImage(F1, TV);
```

With these commands, the entire image is enlarged 3 times and printed on the screen. Below is the image printed on the screen.

If we want to animate the agent while running, before pasting the agent movie frame, we need to temporarily copy the area where it will be pasted to a matrix with the `Copy()` command and then paste the movie frame there. Thus, while the agent is being shifted sideways, we can paste this temporary matrix back to that point with the `Paste()` command and delete the agent without disturbing the background image.



7.8. Continuous Operation Button

As we mentioned before, the Windows operating system works with messages, and your program must process these messages and constantly react to them. If your program is delayed in processing these messages, your program will start to freeze. If it stops processing messages, it becomes completely unresponsive. Widgets such as buttons, menus and tap bars all become unresponsive. You will not be able to minimize the main window of your program, change its location and size, or even close it. For this reason, in the section called "Continuing 7.3 Pacman", we had to create a parallel running thread to make the animation continuous.

However, dealing with threads requires special expertise, and even most computer engineers have not received training on this subject. For this reason, making the user deal with strings can be frightening for them and dangerous in terms of consequences. In order to prevent problems that may arise in this regard, a different button definition has been made in the I-See-Bytes library.

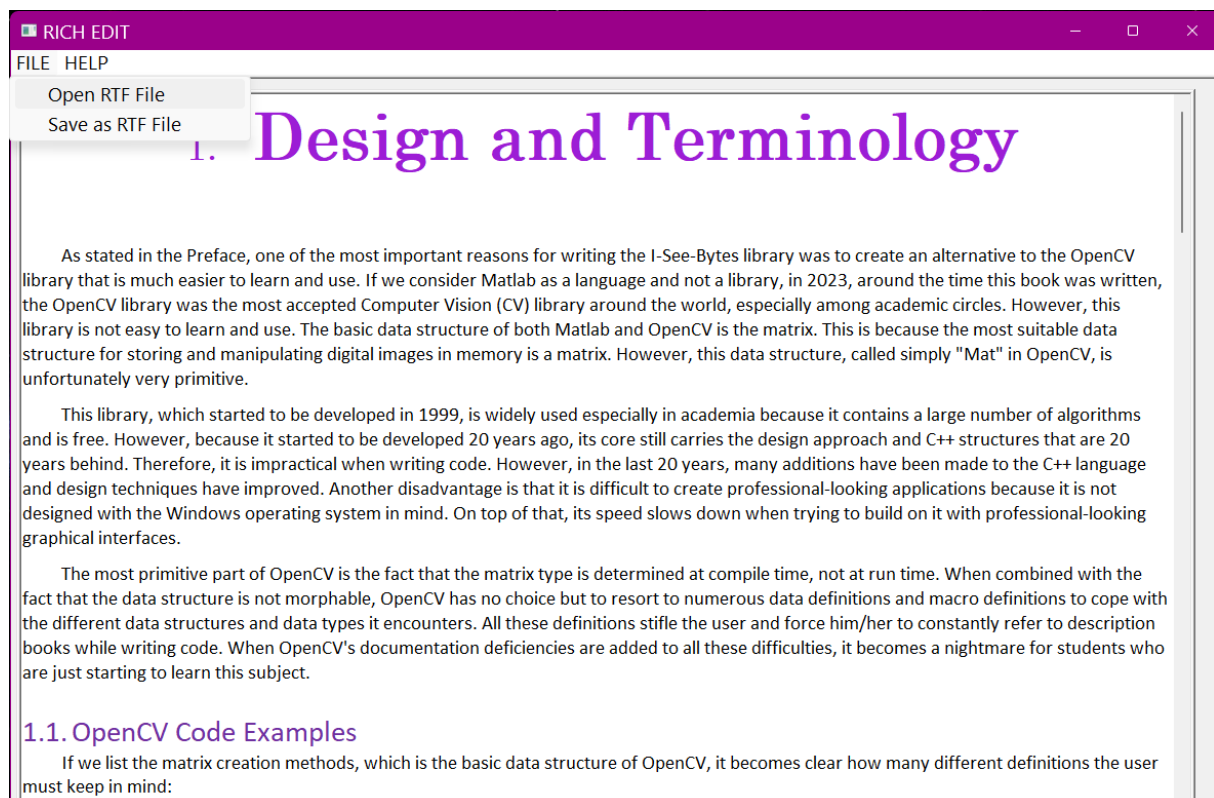
The function called by this button is actually a thread that runs parallel to the main program. However, the user does not need to know this. This button, which we call the On/Off button, is created just like other buttons:

```
ICG_TButton(5, 5, 120, 25, "BUTTON", ButonFunction, void* arguman);
```

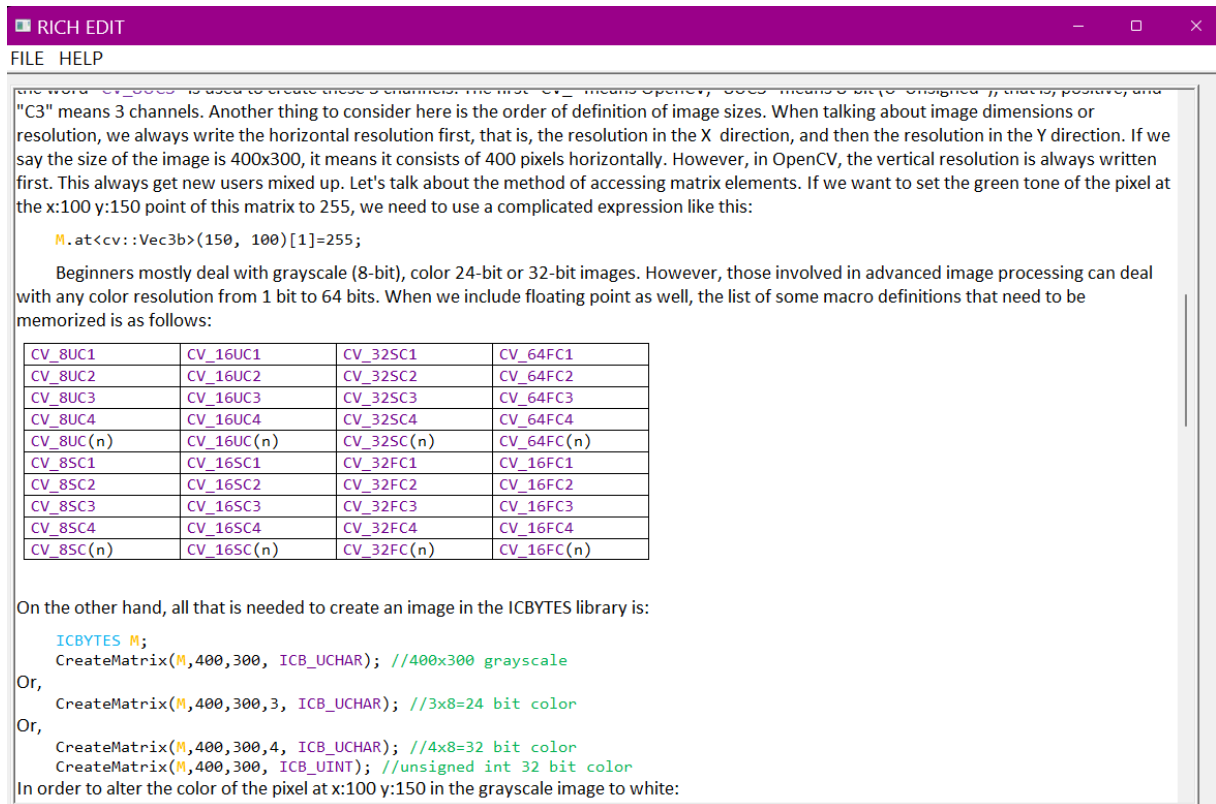
By using this button, you can create the continuous animations you need in games or multimedia applications without dealing with threads.

7.9. Rich Edit Widget

The edit boxes we have used so far only allow us to use black and a single font that is selected from the beginning. However, in programs such as Microsoft Word or Wordpad, we can change the size, color and type of the font. In this section, we will explain how the “Rich Edit” application that comes with the I-See-Bytes library distribution works. Among these project files is a file called “Document.rtf”. After compiling and running the project, when you click on the FILE→“Open RTF File” option from the menu, the file selection menu appears. When you select the “Document.rtf” file from here, the following screen shot will appear. This document is actually the .rtf file saved as the first chapter of this book.



When we scroll down this document, we see texts and even tables that use different colors, sizes and fonts.



Here are the different functions we can use to create rich edit widgets:

```
int ICG_RichEdit(...);
int ICG_RichEditSunken(...);
int ICG_RichEditPanel(...);
int ICG_RichEditThick(...);
int ICG_RichEditResizable(...);
```

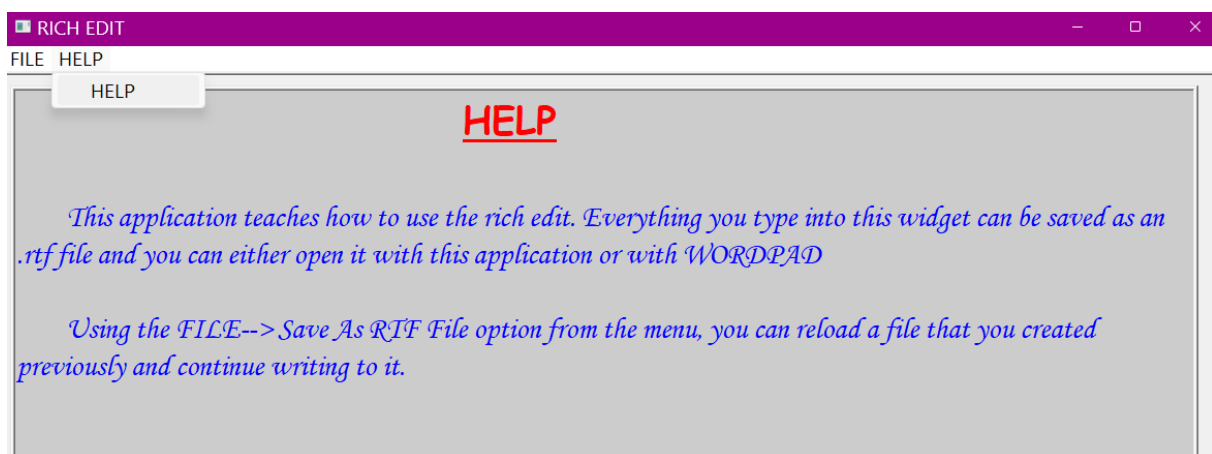
Since the arguments of these functions are the same as the arguments of the other edit boxes, there is no need to explain them again. When we examine the function that opens the file (OpenFileFunc()), we see two new commands:

```
void OpenFileFunc()
{
    ICBYTES filepath;
    ICG_SetRichColor(RICH, 0xffffffff);
    LoadFileToRichEdit(RICH, OpenFileMenu(filepath, "RTF\0*.RTF\0"));
}
```

The first of these is the ICG_SetRichColor() function. It determines the background color of the rich text box. Since the RGB (RGB) values are set to the

The source code of this project also shows how to save the content of your rich text box as an RTF file using the `SaveRichEditToFile()` function. Finally, if you select `HELP→HELP` from the menu, you will see an example of how to print a rich text box using different fonts on a gray background. The `HelpFunction()` shown above uses the `ICG_SetprintFont()` command to change the font name and size. `ICG_setprintfStyle()` allows you to specify level of boldness-lightness and whether it will be written in italics and underlines.

```
void HelpFunction()
{
    ICG_ClearText(RICH);
    ICG_SetRichColor(RICH, 0xcccccc);
    ICG_SetprintfFont("Comic Sans MS", 28);
    ICG_printf(RICH, "\t\t\t\t\t\t\t\t\t\t");
    ICG_SetprintfColor(0xff);
    ICG_SetprintfStyle(9, false, true);
    ICG_printf(RICH, "HELP\n\n");
```



WARNING: Be careful when using fonts that do not support Turkish (or your native alphabet) characters.

APPENDIX-1 ICGUI Functions

Basic Functions:

Core functions are functions that affect the entire application.

void ICGUI_Create()

Function that creates the outermost main window of the application. If you are using ICGUI as a graphical interface, it is one of the 2 functions you must call. General adding colors, fonts, etc. styles; The location and dimensions of the main window and the name of the application written in the title bar at the top of the window can be determined here.

void ICGUI_main()

The part where the application actually works. The boxes that will exist on the main window are created within this function. It works like main in C/C++.

void ICG_MWSize(int width, int height)

Sets the width and height of the main window.

void ICG_MWPosition(int X, int Y)

Determines the upper left corner coordinates of the main window.

void ICG_MWTitle(const TCHAR* title)

Determines the name written in the title bar at the top of the main window. If this function is not used, the window name is default ICB_GUI.

void ICG_MWColor(unsigned R, unsigned G, unsigned B)

Sets the background color of the main window and some tiles. Any color can be achieved by mixing Red (R), Green (G), and Blue (B) tones with brightness levels between 0-255.

void ICG_MW_RemoveTitleBar()

Eliminates the title bar at the top of the main window. This function should be used very carefully because the buttons that minimize, maximize and terminate the application also disappear. Other mechanisms must be in place to terminate the program, otherwise you will not be able to close the application.

HWND ICG_GetHWND(int handle)

In the Windows operating system, a handle is created for each window and tile. This handle is in **HWND**. In the ICGUI library, the handles of all boxes are integers (**int**). This function allows accessing the Windows handle of the tiles created by ICGUI. If you do not know the WIN32 API, it is recommended not to use it.

bool ICG_SetFont(int H, int W, const char* fontname)

Selects a font other than Windows' default font. This function affects the created tiles from the moment it is called. Its first two inputs are the height (H) and width (W) of the font. However, if the width is given as 0, the same height is used. Fontnames are names such as "Times New Roman", "Arial", or the font we currently use, "Calibri", which are listed in the MS Word program. If the requested font is not installed on the system, it returns **false**.

void ICG_SetSystemFont()

It changes the font of the tiles created as soon as it is called back to the Windows default font.

void ICG_DestroyWidget(int handle)

Destroys any type of widget. The handle of the widget is the input of the function.

bool ICG_SetWindowText(int handle, const char* string)

Changes the text of any widget that has text.

void ICG_SetOnMouseMove(void (*MouseFunc)(int, int))

Selects the function to run each time the mouse moves.

int ICG_GetMouseX()

It tells you the current horizontal position (X) of the mouse relative to the upper left corner of the main window, in pixels.

int ICG_GetMouseY()

It tells you the current vertical position (Y) of the mouse relative to the upper left corner of the main window, in pixels.

void ICG_SetOnMouseLClick(void (*MouseFunc)())

Selects the function to be called when the left mouse button is pressed.

void CG_SetOnKeyPressed(void(*OnKeyPressedFunc)(int))

Selects the function to be run when any key of the keyboard is pressed.

void SetText(int handle, ICBYTES& m)

Displays the character string in the ICBYTES fluid (m), inside the widget selected by the handle.

void GetText(int handle, ICBYTES& m)

It copies the text of the edit window of the handle into the ICBYTES fluid (m).

Edit Window Functions:

The functions described below are used to create or print various types of edit boxes.

int ICG_SLEdit()

Creates a frameless single-line text box.

int ICG_SLEditBorder()

Creates a single-line text box with a thin black line border.

int ICG_SLEditThick()

Creates a convex single-line text box with a thick frame.

int ICG_SLEditSunken()

Creates a single-line text box whose frame looks collapsed.

int ICG_SLPASSWORDB()

Creates a single-line password entry box with a thin black line border.

int ICG_SLPASSWORDSunken()

It creates a single-line password entry box whose frame looks like it's caved in.

int ICG_IPADDRESSSunken()

It creates a single-line internet address entry (IP version 4) box whose frame looks like it's collapsed inwards.

int ICG_IPAddressThick()

It creates a convex single-line internet address entry (IP v4) box with a thick frame.

int Print(int handle, ICBYTES& i)

If there is a character string in the fluid "i", it prints it in the edit box connected to the handle. If the matrix i has a single row, that is, if it is a vector, it writes a single row. If i contains a matrix, it writes each row of the matrix one under the other.

int ICG_printf(int handle, const char* format, ...)

It prints formatted string into the edit box given by the handle, imitating the behavior of the printf() function in C/C++. It exhibits the same format and variable type rules.

void ICG_SetPrintWindow(int handle)

If you do not want to constantly select the text box handle as above, that is, if you are going to constantly print in the same text box, you must specify this from the beginning and then call this function to call the ICG_printf() function without using the handle.

void ICG_SetPrintWindow(HWND c)

It does the same function as the previous one, but uses the Windows operating system handle instead of the ICGUI handle.

int ICG_printf(const char* format, ...)

If the edit window to be written is determined from the beginning, it behaves exactly the same as the printf() function in C/C++.

APPENDIX-2

ICBYTES Matrix Functions

Matrix Creation Methods:

In this section, matrix creation methods within the ICBYTES fluid will be demonstrated.

ICBYTES **A** = {{1,2,3},{4,5,6},{7,8,9}};

Creates an integer 3×3 matrix.

ICBYTES **A** = {{1.0,2.0,3.0},{4.0,5.0,6.0},{7.0,8.0,9.0}};

Creates an double 3×3 matrix.

CreateMatrix(A, X,Y,Z,W, ICB_TYPE)

Creates a matrix of type ICB_TYPE with dimensions X×Y×Z×W.

CreateImage(A, X,Y,Z,W, ICB_TYPE)

Creates a matrix of type ICB_TYPE with dimensions X×Y×Z×W. It is faster to display on the screen. It is slower to create and destroy than the plain matrix. It consumes more RAM.

CreateSound(i, x, y, ICB_TYPE, Hz)

Creates a matrix containing X channels and Y length digital audio samples to be played at Hz sampling rate. Suitable for rapid playback on audio input/output devices.

ICBYTES A;

A=5.3; or A=-2;

It creates a 1x1 matrix inside fluid A. The matrix type is either double (5.3) or int (-2).

A = "ABCDEF";

A 6×1 char matrix is created inside the fluid A and (ABCDEF) is copied into it.

Matrix Query:

This section lists the functions that you can query the matrix to obtain information about itself.

bool IsFloating(ICBYTES& i)

Returns **true** if the matrix type is float or double.

bool IsMatrix(ICBYTES& i)

Returns **true** if the fluid sent as input carries a matrix.

bool AreDimsEqual(ICBYTES& i, ICBYTES& j)

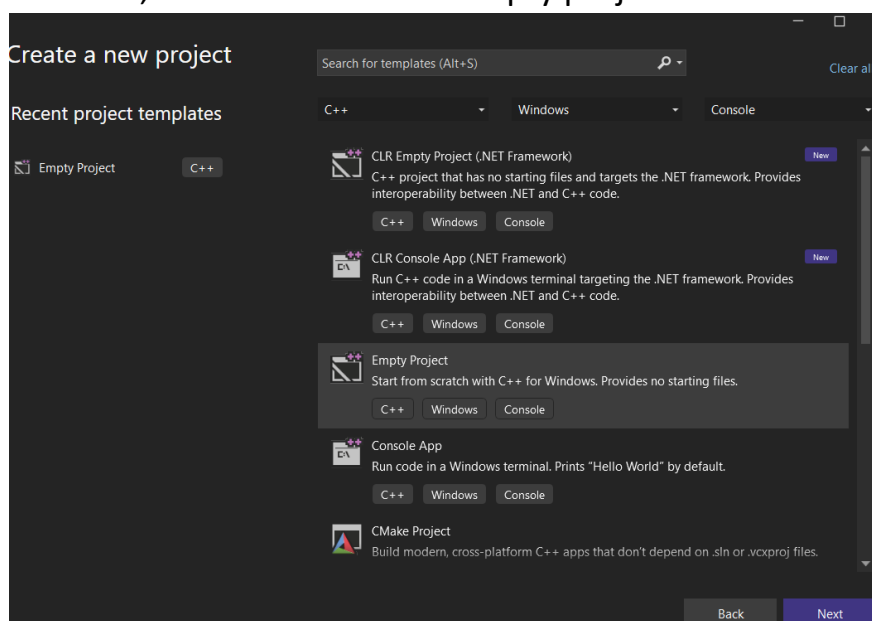
Returns **true** if the sizes of the matrices sent as input are equal.

bool AreEqualImage(ICBYTES& i, ICBYTES& j)

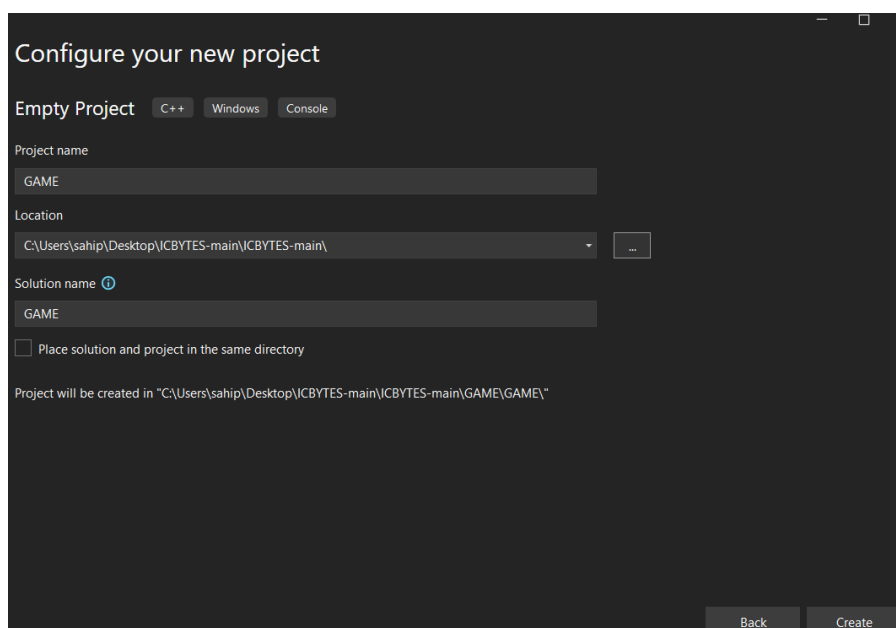
Returns **true** if the sizes and color depths of the matrices sent as input are equal.

APPENDIX-3 Creating an ICBYTES Project

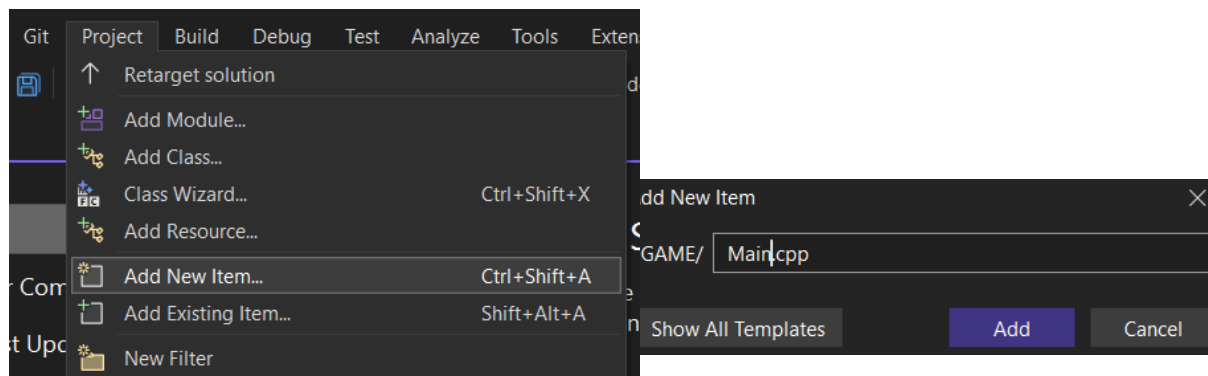
This section explains how to create a new project using the I-See-Bytes library. First, download the ICBYTES library, extract (unzip) the zipped files and copy them to your hard disk. After opening Visual Studio, choose to create a new project from the Menu, and then select the empty project as shown below:



Then select the project folder in the ICBYTES-main folder and give the project a name. For example, let's say "GAME":



Now it's time to add C/C++ files to the project. Since it is mandatory to add at least one C/C++ file in which our ICGUI_Main() function will be located, or our main() function if we are not going to use GUI, or our WinMain() function if we are going to use the WIN32 API. If we are going to name this file Main.cpp, first, to add the C++ file, go to Project->Add New Item from the menu. We must choose. Then, we can write "Main.cpp" (for example) in the small window that appears in front of us:



Then let's write something in Main.cpp:

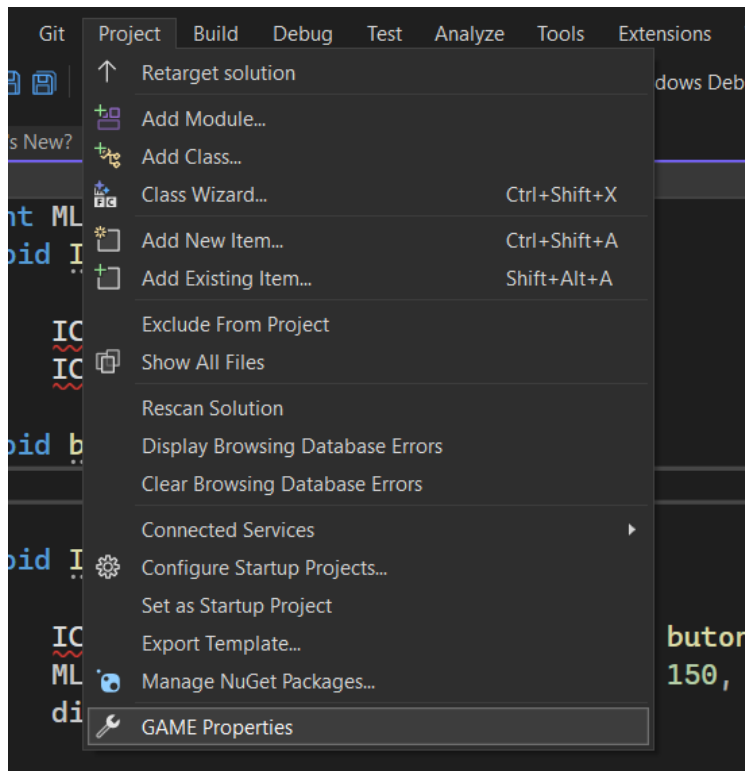
```

1  #include"icb_gui.h"
2  extern ICBDIAG diag;
3
4  int MLE1;
5  void ICGUI_Create()
6  {
7      ICG_MWTitle("GAME");
8      ICG_MWSize(430, 300);
9  }
10 void butonfunc()
11 {
12 }
13 void ICGUI_main()
14 {
15     ICG_Button(5, 5, 120, 25, "Button1", butonfunc);
16     MLE1 = ICG_MLEditSunken(5, 50, 250, 150, "", SCROLLBAR_V);
17     diag.SetOutput(ICG_GetHWND(MLE1));
18 }

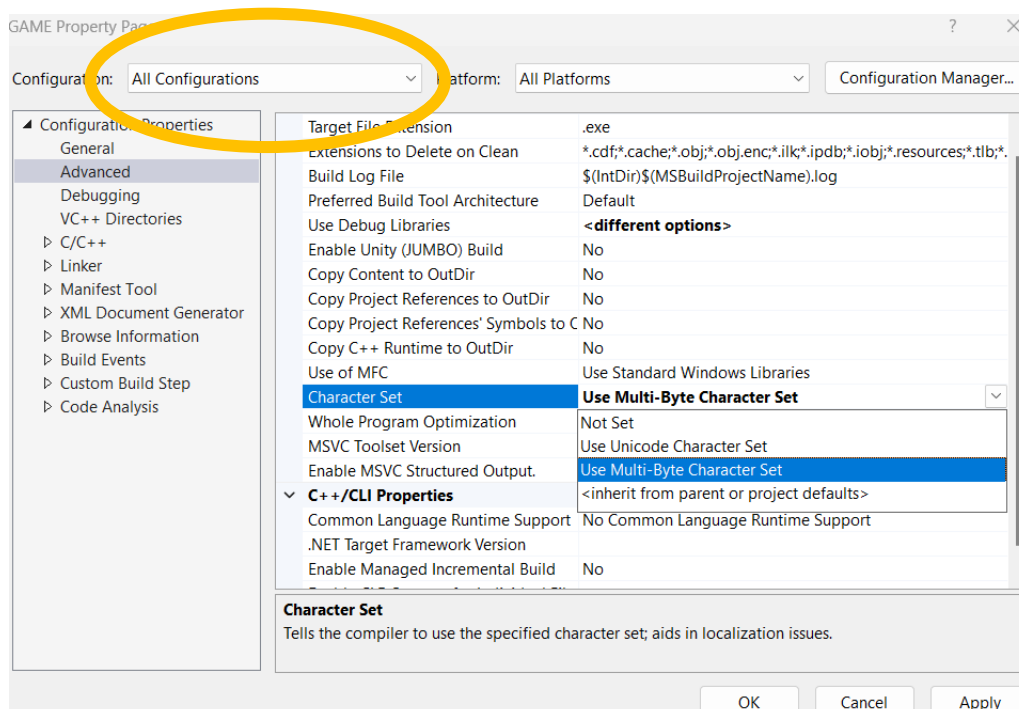
```

Our project now has the files it needs to work. Now it is time to include the I-See-Bytes library in the project and adjust the project settings.

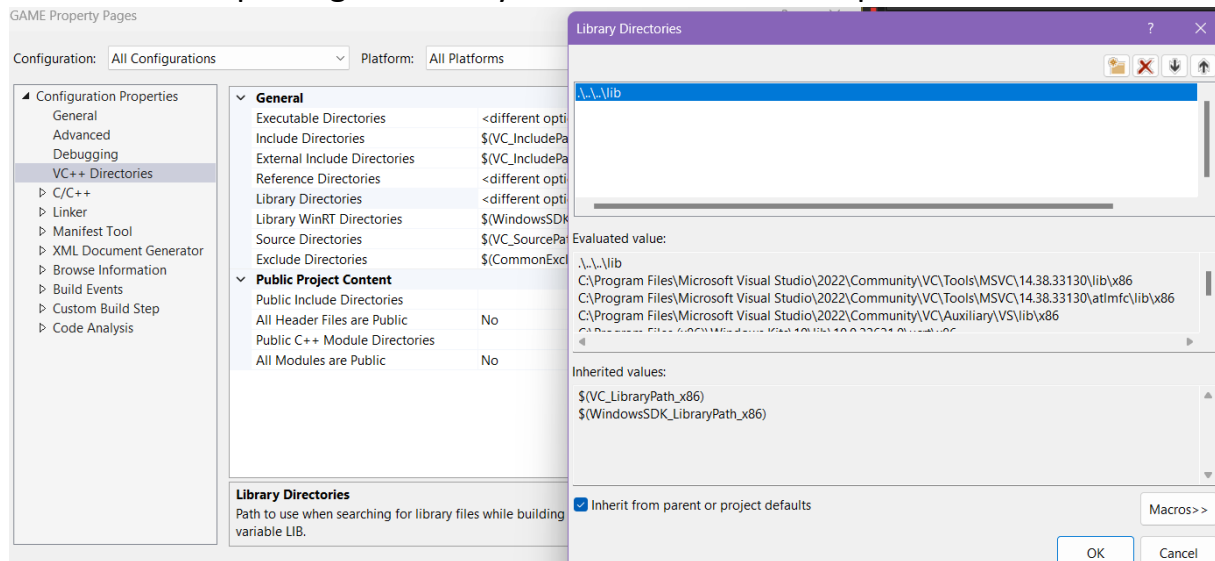
To enter the project properties, let's select Project->Game Properties from the Visual Studio menu:



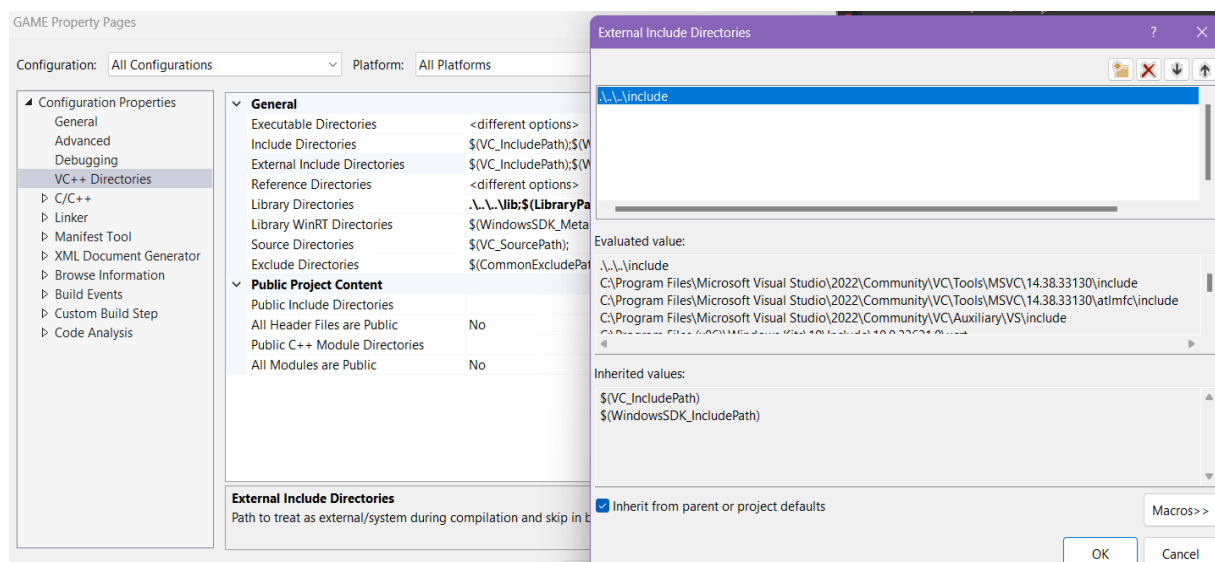
In the resulting window, select all configurations at the top and then select "Use Multibyte Character Sets".



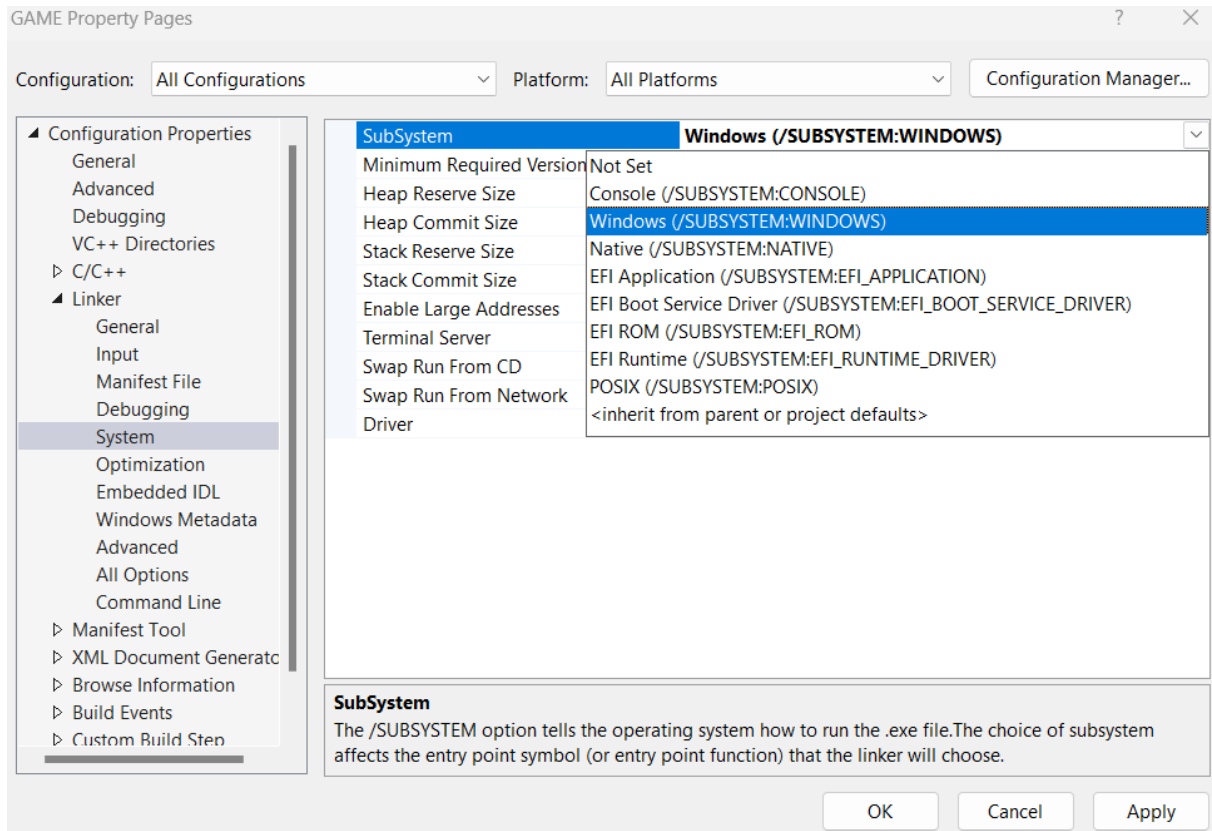
Next is reporting the library directories to the compiler:



When we open the ICBYTES-main.zip file that we downloaded from the internet, you will notice that there are two folders called "lib" and "include". These folders contain the compiled ".lib" and ".h" files of the ICBYTES library. We tell the compiler the location of these folders relative to our project folder. In this way, if the project we have written needs to be compiled on another computer, for example if we need to send it as an assignment to our colleagues or the course assistant, we can ensure that it is compiled on their computer without any problems. Now we tell the location of the "include" files:



And finally, we say that we want to create a windows application, not a console application. Of course, if we want to make a console application, we do not need to change this.



AT THIS POINT PRESS OK AND CLOSE THE WINDOW.

But it's not over yet. Continued on next page.

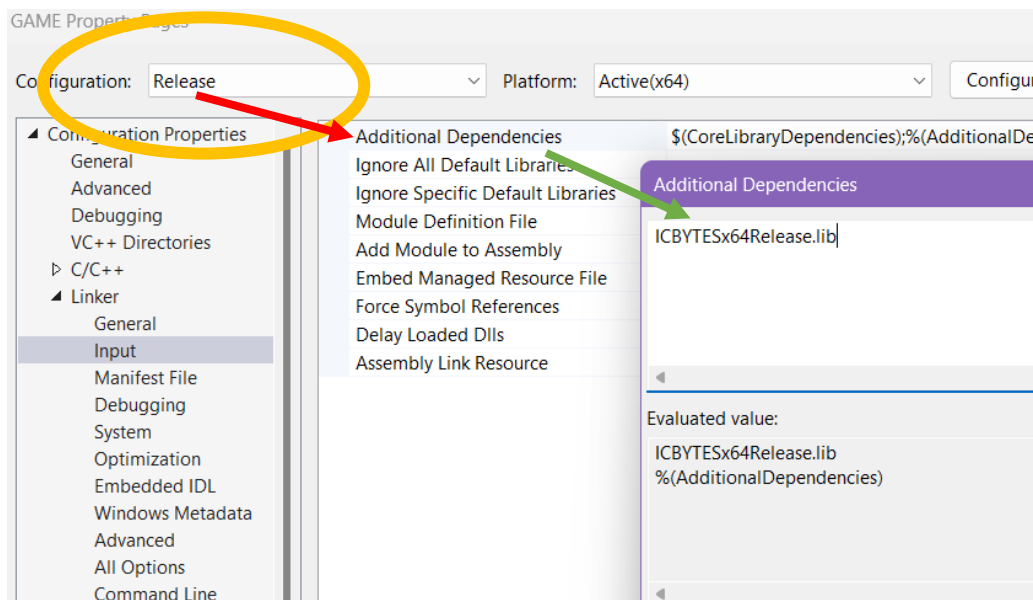
Now it's time to add the ".lib" files. However, as we see in the "lib" folder, there are two different lib (library) files. These:

ICBYTESx64Debug.lib

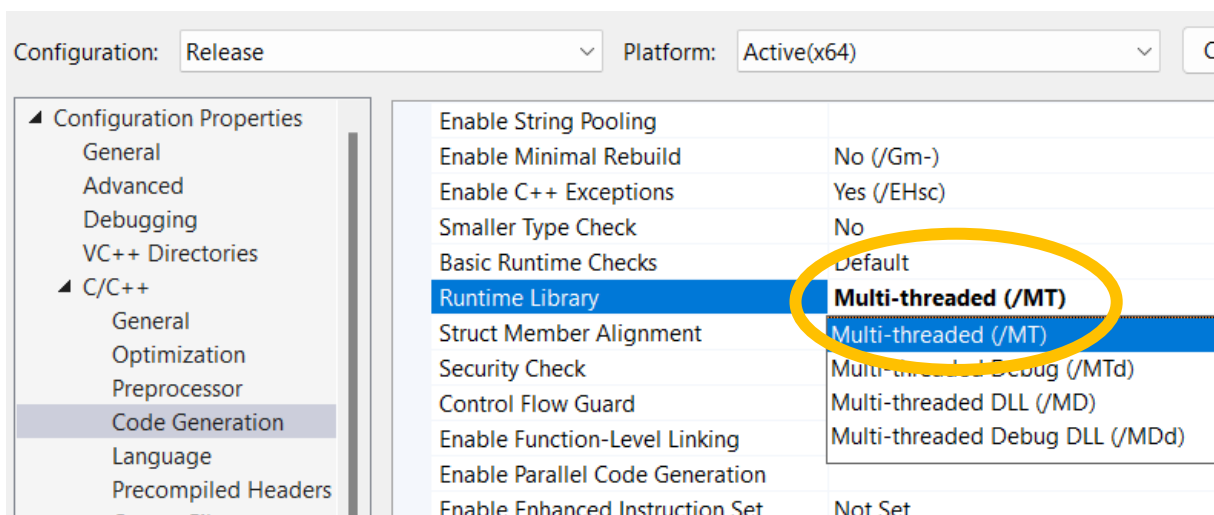
And,

ICBYTESx64Release.lib

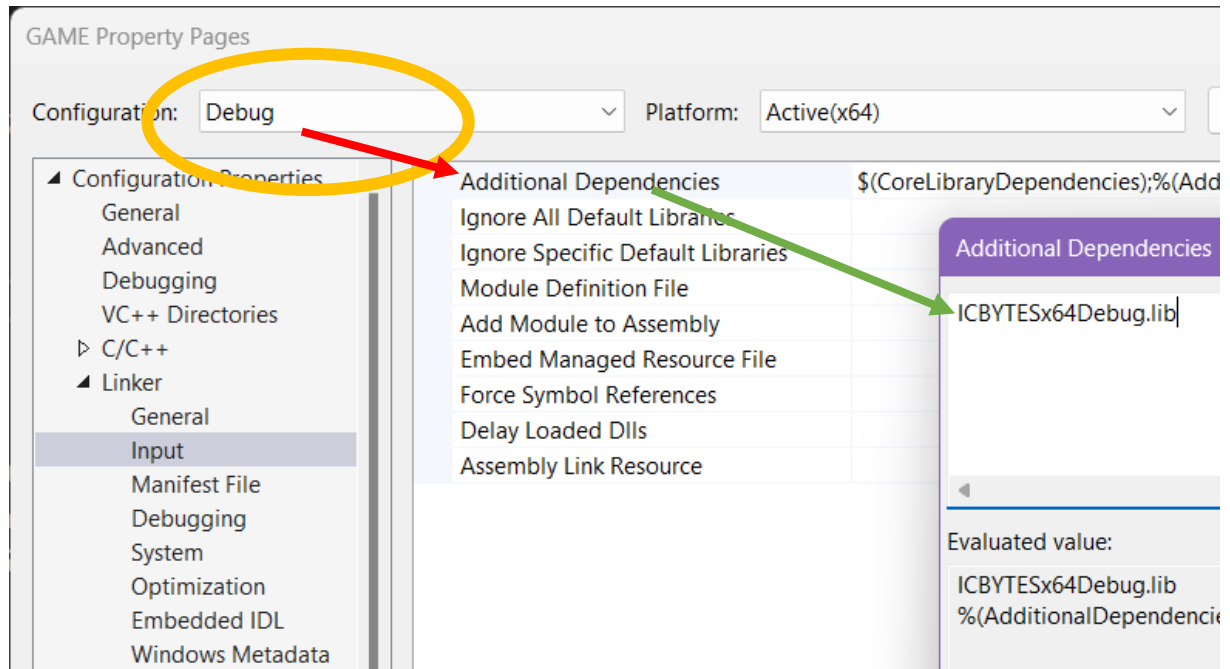
files. When compiling our program, we can compile it in two different modes. These are Release and Debug modes. In these modes, different libraries and compilation options must be selected. To enter the project properties, let's select Project->Game Properties from the Visual Studio menu again. However, this time we select RELEASE mode from the upper left corner, select Linker→Input→Additional Dependencies to use the release library and write the name of the library file:



Then we enter C/C++ → code generation and change the “Runtime Library” option to “Multi-threaded (/MT)” and press “OK”.



To adjust the settings in Debug mode, we need to go to Project->Game Properties again. After selecting "Debug" from the "Configuration" options in the upper left corner, we select Connector→Input→Additional Dependencies and this time write the name of the debug library file:



As in Release mode, we enter C/C++ → code generation in Debug mode, change the “Runtime Library” option to “Multi-threaded (/MTd)” and press “OK”. (It was /MT in Release.) Our project is now ready to be compiled.

