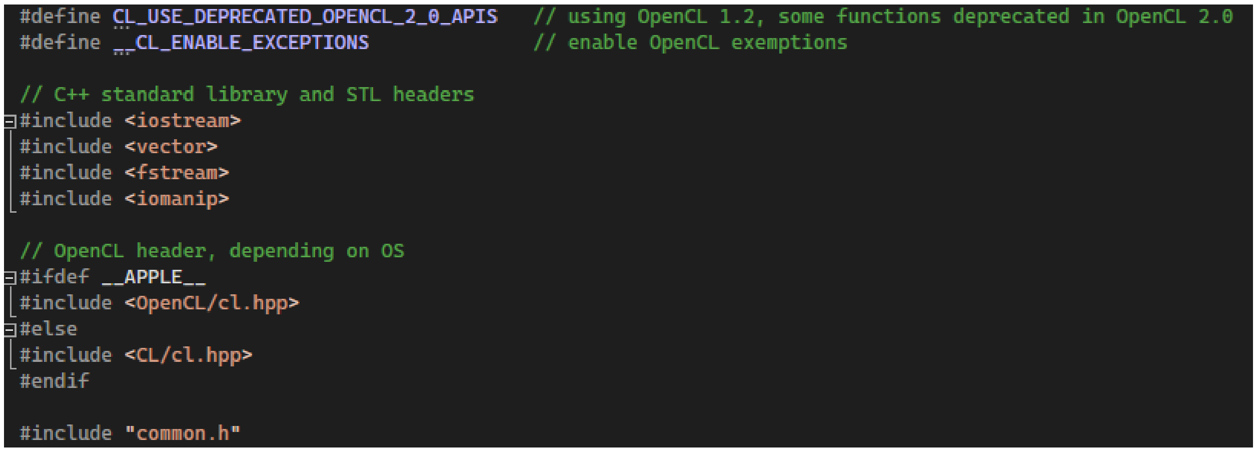
A2 Report

Standard Headers and Libraries



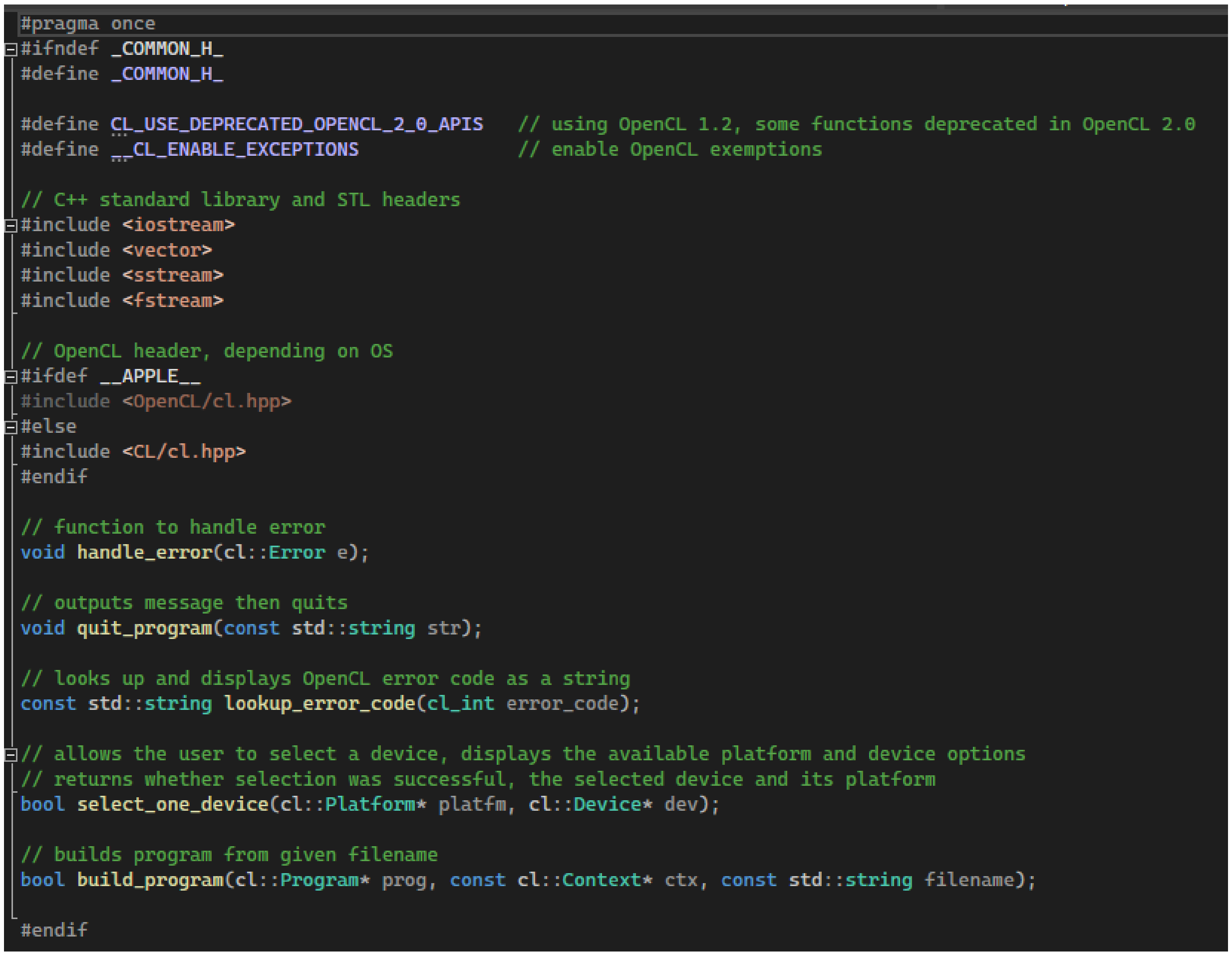
```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS   // using OpenCL 1.2, some functions deprecated in OpenCL 2.0
#define __CL_ENABLE_EXCEPTIONS              // enable OpenCL exemptions

// C++ standard library and STL headers
#include <iostream>
#include <vector>
#include <fstream>
#include <iomanip>

// OpenCL header, depending on OS
#ifdef __APPLE__
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif

#include "common.h"
```

The #define statements enable the use of deprecated OpenCL 2.0 APIs and the use of OpenCL exceptions in the code.

The #include statements include standard C++ library headers such as iostream, vector, sstream, and fstream, as well as the OpenCL header file cl.hpp, which is specific to the OS being used.

Finally, the #include statement includes a custom header file common.h, which likely contains function declarations or macros used throughout the OpenCL program.

Common.h



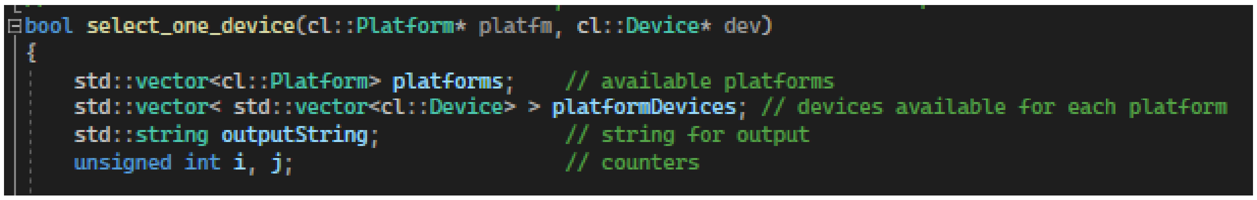
```
#pragma once
#ifndef _COMMON_H_
#define _COMMON_H_

#define CL_USE_DEPRECATED_OPENCL_2_0_APIS   // using OpenCL 1.2, some functions deprecated in OpenCL 2.0
#define __CL_ENABLE_EXCEPTIONS              // enable OpenCL exemptions

// C++ standard library and STL headers
#include <iostream>
#include <vector>
#include <sstream>
#include <fstream>

// OpenCL header, depending on OS
#ifdef __APPLE__
#include <OpenCL/cl.hpp>
#else
#include <CL/cl.hpp>
#endif

// function to handle error
void handle_error(cl::Error e);

// outputs message then quits
void quit_program(const std::string str);

// looks up and displays OpenCL error code as a string
const std::string lookup_error_code(cl_int error_code);

// allows the user to select a device, displays the available platform and device options
// returns whether selection was successful, the selected device and its platform
bool select_one_device(cl::Platform* platfm, cl::Device* dev);

// builds program from given filename
bool build_program(cl::Program* prog, const cl::Context* ctx, const std::string filename);

#endif
```

The header file includes the same preprocessor directives and header files as in the previous code snippet. Additionally, it declares several functions used throughout the OpenCL program: handle_error(), quit_program(), lookup_error_code(), select_one_device(), and build_program().

The handle_error() function handles OpenCL errors by outputting the error message and code to the console. The quit_program() function outputs a message and exits the program. The lookup_error_code() function converts OpenCL error codes into human-readable strings. The select_one_device() function allows the user to select an OpenCL device and returns whether the selection was successful along with the selected device and its platform. The build_program() function builds an OpenCL program from a given filename.

Common.cpp

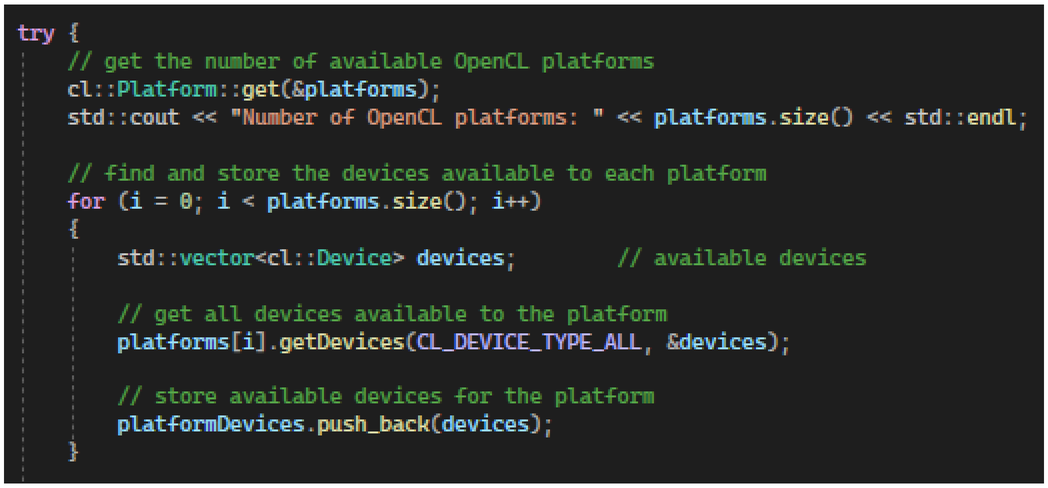
Select one device function



```
bool select_one_device(cl::Platform* platfm, cl::Device* dev)
{
    std::vector<cl::Platform> platforms;    // available platforms
    std::vector< std::vector<cl::Device> > platformDevices; // devices available for each platform
    std::string outputString;               // string for output
    unsigned int i, j;                      // counters
```

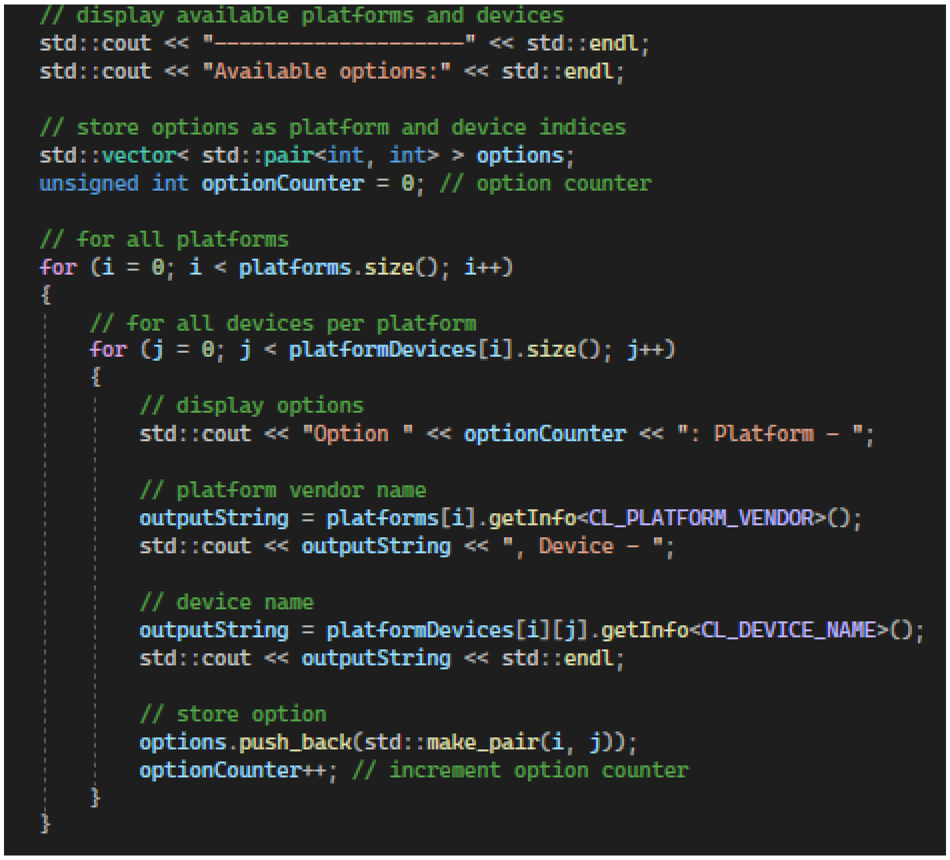
The function is defined with two pointer parameters, platfm and dev, which will be used to return the selected OpenCL platform and device, respectively.

The function declares several variables to be used later. platforms is a vector that will store all available OpenCL platforms, and platformDevices is a vector of vectors that will store all available devices for each platform. outputString is a string that will be used to store various output messages, and i and j are counters that will be used in the for-loops later.



```
try {
    // get the number of available OpenCL platforms
    cl::Platform::get(&platforms);
    std::cout << "Number of OpenCL platforms: " << platforms.size() << std::endl;

    // find and store the devices available to each platform
    for (i = 0; i < platforms.size(); i++)
    {
        std::vector<cl::Device> devices;        // available devices

        // get all devices available to the platform
        platforms[i].getDevices(CL_DEVICE_TYPE_ALL, &devices);

        // store available devices for the platform
        platformDevices.push_back(devices);
    }
```

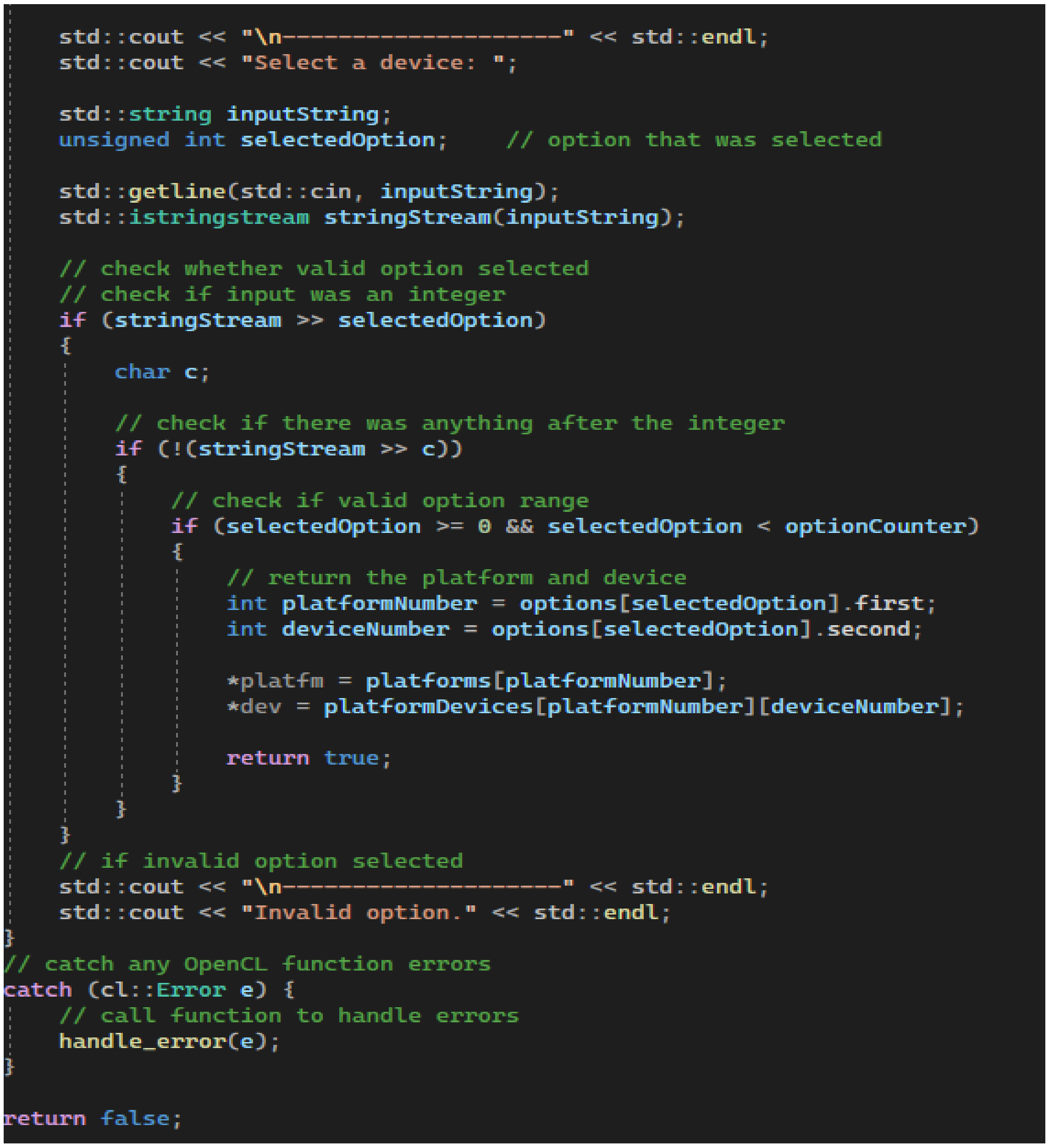
The function starts by using the cl::Platform::get() method to retrieve the number of available OpenCL platforms and stores them in the platforms vector. Then, for each platform, the function retrieves all available devices using the getDevices() method and stores them in platformDevices.



```
// display available platforms and devices
std::cout << "--------------------" << std::endl;
std::cout << "Available options:" << std::endl;

// store options as platform and device indices
std::vector< std::pair<int, int> > options;
unsigned int optionCounter = 0; // option counter

// for all platforms
for (i = 0; i < platforms.size(); i++)
{
    // for all devices per platform
    for (j = 0; j < platformDevices[i].size(); j++)
    {
        // display options
        std::cout << "Option " << optionCounter << ": Platform - ";

        // platform vendor name
        outputString = platforms[i].getInfo<CL_PLATFORM_VENDOR>();
        std::cout << outputString << ", Device - ";

        // device name
        outputString = platformDevices[i][j].getInfo<CL_DEVICE_NAME>();
        std::cout << outputString << std::endl;

        // store option
        options.push_back(std::make_pair(i, j));
        optionCounter++; // increment option counter
    }
}
```

The function then displays all available options for the user to select from. It first declares an options vector that will store all the available platform and device indices for each option. It also declares an optionCounter variable that will be used to keep track of the number of available options.

For each platform and device combination, the function displays an option number, the platform vendor name, and the device name. It then stores the platform and device indices in the options vector and increments the optionCounter.



```
    std::cout << "\n--------------------" << std::endl;
    std::cout << "Select a device: ";

    std::string inputString;
    unsigned int selectedOption;    // option that was selected

    std::getline(std::cin, inputString);
    std::istringstream stringStream(inputString);

    // check whether valid option selected
    // check if input was an integer
    if (stringStream >> selectedOption)
    {
        char c;

        // check if there was anything after the integer
        if (!(stringStream >> c))
        {
            // check if valid option range
            if (selectedOption >= 0 && selectedOption < optionCounter)
            {
                // return the platform and device
                int platformNumber = options[selectedOption].first;
                int deviceNumber = options[selectedOption].second;

                *platfm = platforms[platformNumber];
                *dev = platformDevices[platformNumber][deviceNumber];

                return true;
            }
        }
    }
    // if invalid option selected
    std::cout << "\n--------------------" << std::endl;
    std::cout << "Invalid option." << std::endl;
}
// catch any OpenCL function errors
catch (cl::Error e) {
    // call function to handle errors
    handle_error(e);
}

return false;
```

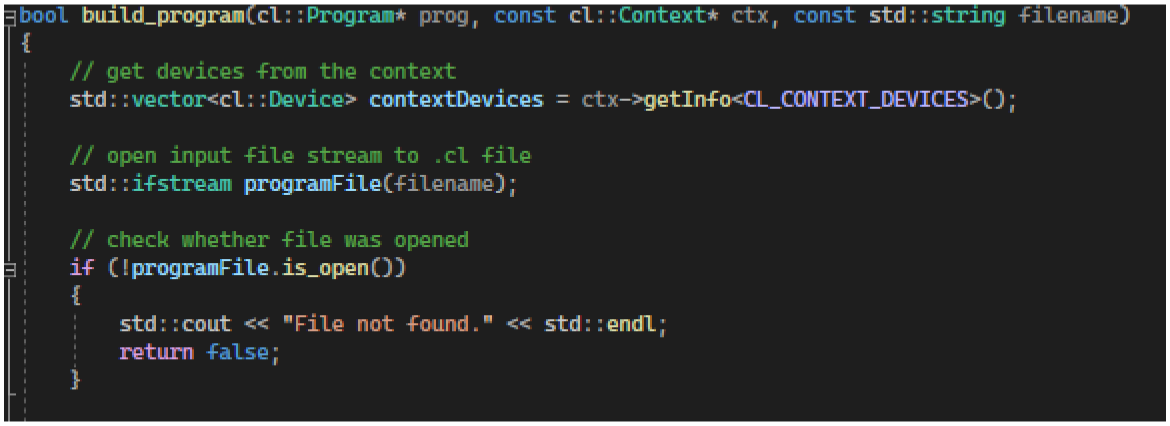
Finally, the function prompts the user to select an option by entering its corresponding number. The user's input is stored in a string called inputString, which is then converted to an integer using istringstream.

The function then checks whether the selected option is a valid integer, whether there are no extra characters after the integer, and whether the selected option is within the range of available options. If all checks pass, the function returns the selected platform and device by setting the values of platfm and dev pointers and returns true to indicate success.

If any of the checks fail, the function displays an error message and returns false to indicate failure.

If there are any OpenCL function errors, they are caught by a try-catch block and handled by calling the handle_error() function.

Build program function

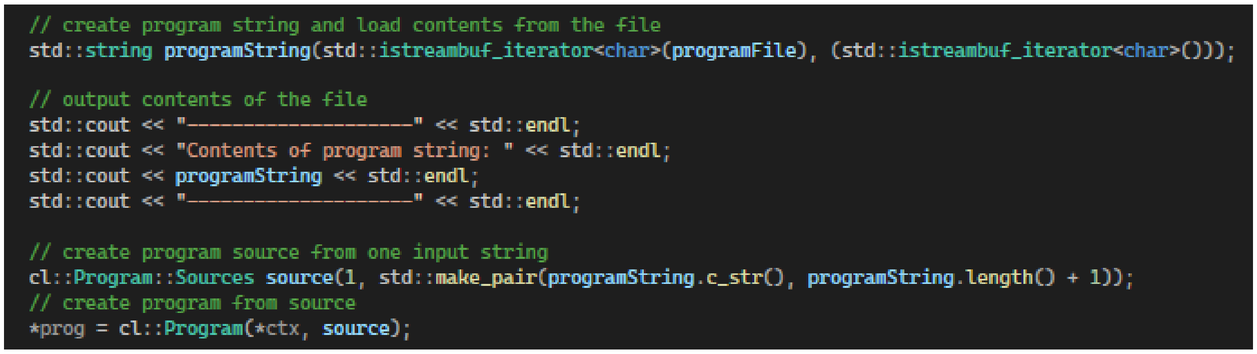


```
bool build_program(cl::Program* prog, const cl::Context* ctx, const std::string filename)
{
    // get devices from the context
    std::vector<cl::Device> contextDevices = ctx->getInfo<CL_CONTEXT_DEVICES>();

    // open input file stream to .cl file
    std::ifstream programFile(filename);

    // check whether file was opened
    if (!programFile.is_open())
    {
        std::cout << "File not found." << std::endl;
        return false;
    }
```

This function builds an OpenCL program from a source file. It takes three arguments, a pointer to a cl::Program object, a pointer to a cl::Context object, and a string filename of the source file. The function returns a boolean value to indicate whether the program build was successful.

The function first retrieves the devices associated with the given OpenCL context and stores them in a vector called contextDevices.

The function then opens the source file with the given filename using an input file stream called programFile. If the filename is invalid, the function outputs an error message and returns false.

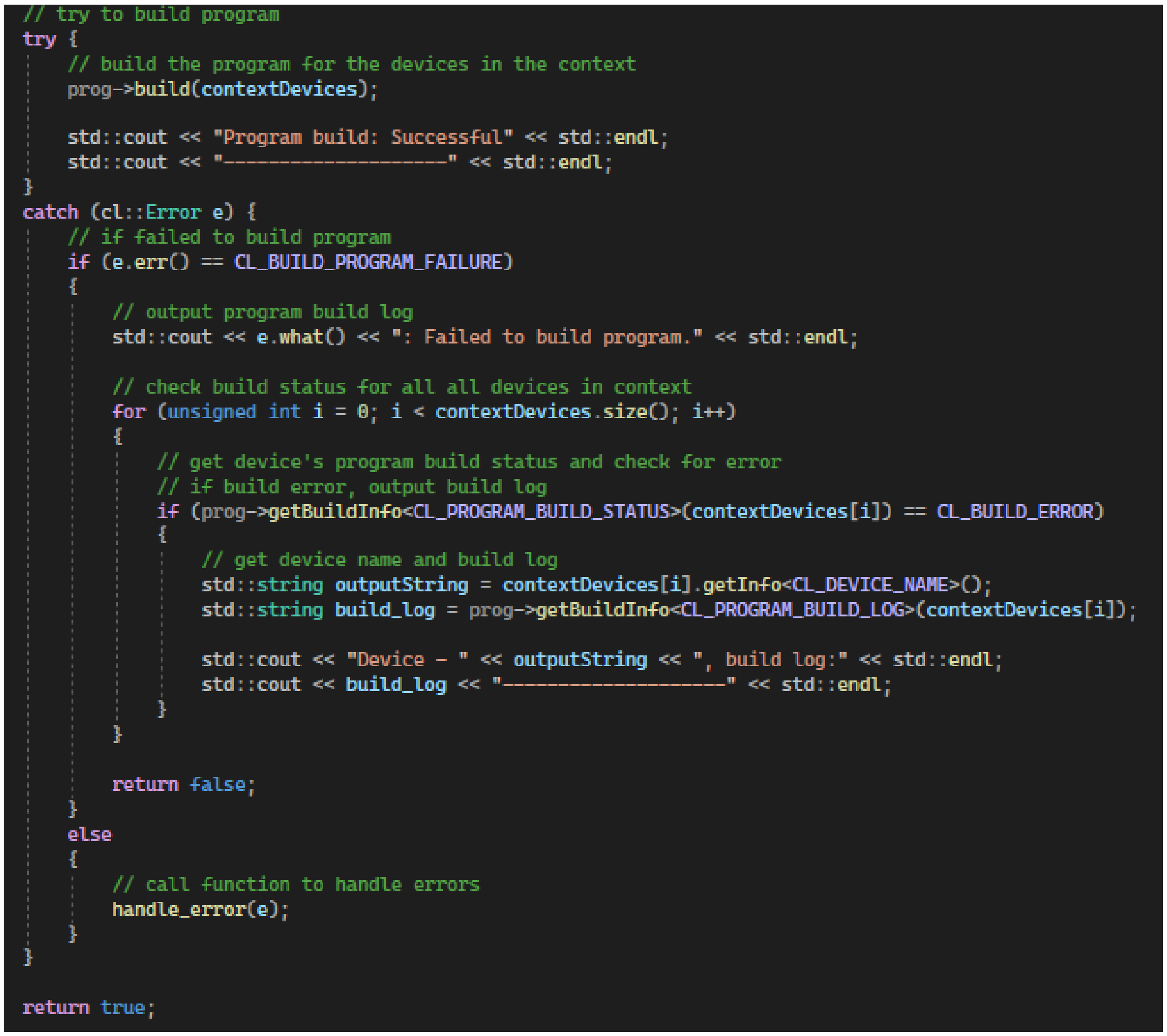


```
// create program string and load contents from the file
std::string programString(std::istreambuf_iterator<char>(programFile), (std::istreambuf_iterator<char>()));

// output contents of the file
std::cout << "--------------------" << std::endl;
std::cout << "Contents of program string: " << std::endl;
std::cout << programString << std::endl;
std::cout << "--------------------" << std::endl;

// create program source from one input string
cl::Program::Sources source(1, std::make_pair(programString.c_str(), programString.length() + 1));
// create program from source
*prog = cl::Program(*ctx, source);
```

The function then creates a string called programString and loads its contents from the file using the istreambuf_iterator<char> method. This method reads the file contents and stores them in programString.

The function then outputs the contents of programString to the console to verify that the correct file was loaded.

The function creates a cl::Program::Sources object called source that contains the program source code in programString. It then creates a cl::Program object by calling the cl::Program constructor with the context and source as arguments. The resulting program is stored in the prog pointer variable.

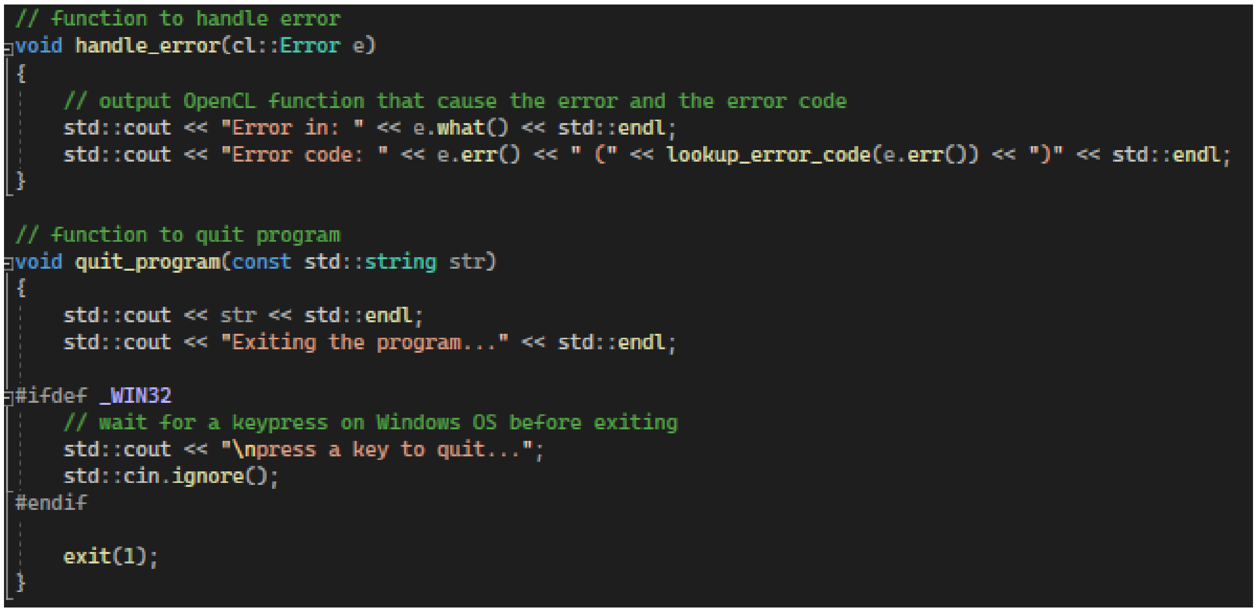


```cpp
// try to build program
try {
    // build the program for the devices in the context
    prog->build(contextDevices);

    std::cout << "Program build: Successful" << std::endl;
    std::cout << "--------------------" << std::endl;
}
catch (cl::Error e) {
    // if failed to build program
    if (e.err() == CL_BUILD_PROGRAM_FAILURE)
    {
        // output program build log
        std::cout << e.what() << ": Failed to build program." << std::endl;

        // check build status for all all devices in context
        for (unsigned int i = 0; i < contextDevices.size(); i++)
        {
            // get device's program build status and check for error
            // if build error, output build log
            if (prog->getBuildInfo<CL_PROGRAM_BUILD_STATUS>(contextDevices[i]) == CL_BUILD_ERROR)
            {
                // get device name and build log
                std::string outputString = contextDevices[i].getInfo<CL_DEVICE_NAME>();
                std::string build_log = prog->getBuildInfo<CL_PROGRAM_BUILD_LOG>(contextDevices[i]);

                std::cout << "Device - " << outputString << ", build log:" << std::endl;
                std::cout << build_log << "--------------------" << std::endl;
            }
        }

        return false;
    }
    else
    {
        // call function to handle errors
        handle_error(e);
    }
}

return true;
```

The function then tries to build the program using the devices in the contextDevices vector. If the build is successful, the function outputs a message indicating success.

If the build fails, the function catches a cl::Error exception and checks whether the error is due to a build failure. If so, the function outputs an error message and the build log for each device in the context that experienced a build error. The function then returns false to indicate failure.

Handle error & quit program function

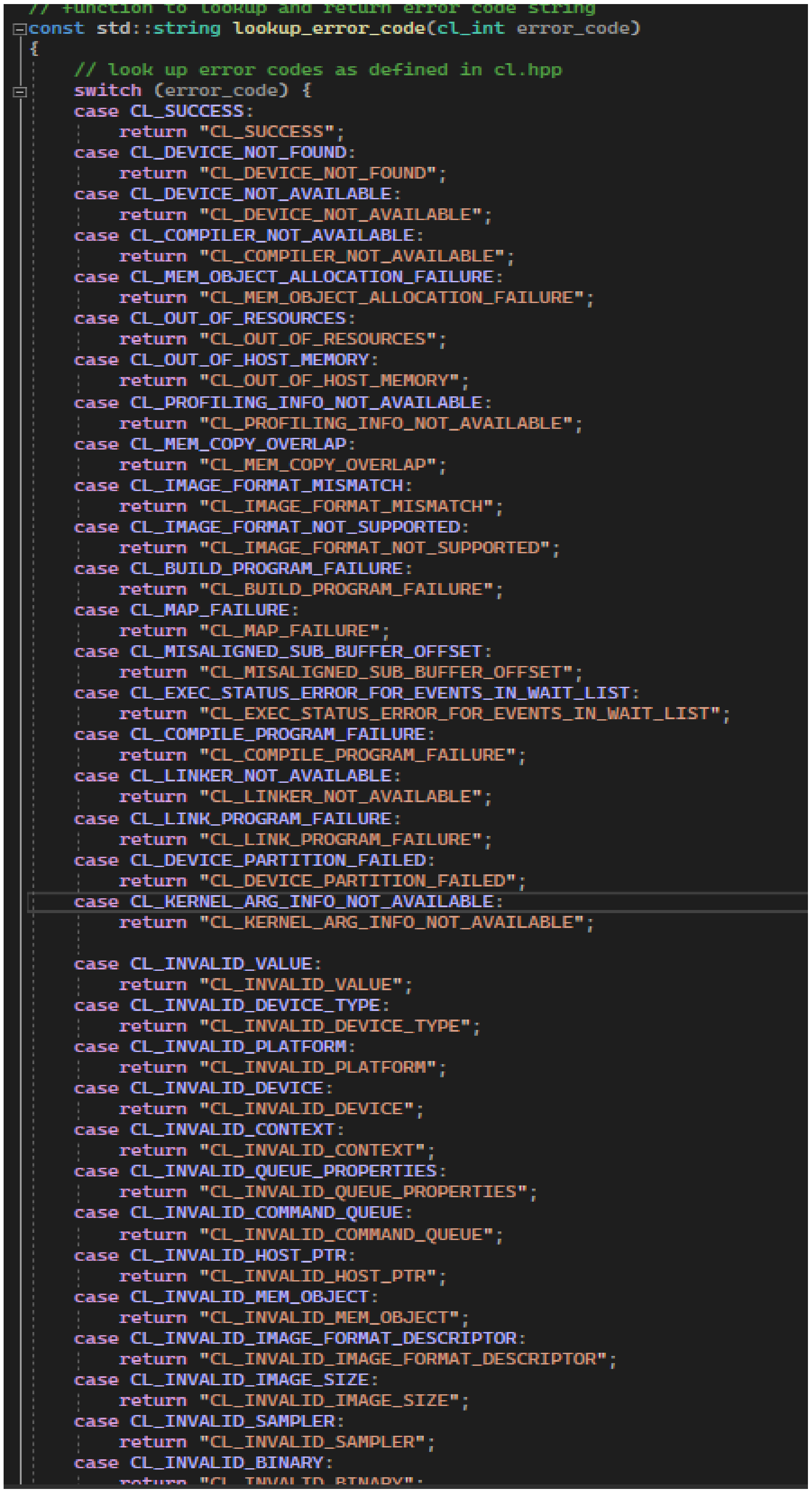


```cpp
// function to handle error
void handle_error(cl::Error e)
{
    // output OpenCL function that cause the error and the error code
    std::cout << "Error in: " << e.what() << std::endl;
    std::cout << "Error code: " << e.err() << " (" << lookup_error_code(e.err()) << ")" << std::endl;
}

// function to quit program
void quit_program(const std::string str)
{
    std::cout << str << std::endl;
    std::cout << "Exiting the program..." << std::endl;

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    exit(1);
}
```

The handle_error() function takes a cl::Error object as an argument and outputs the OpenCL function that caused the error and the error code. It does not return a value.

The quit_program() function takes a string argument that is printed to the console along with a message indicating that the program is exiting

Error Codes



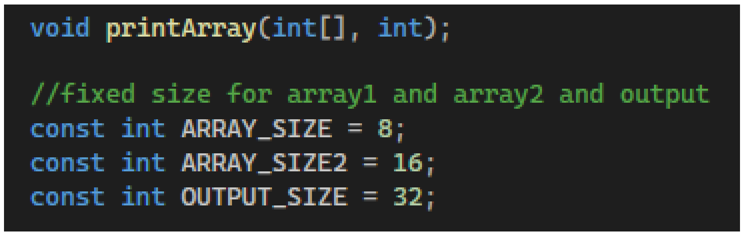
```
// function to lookup and return error code string
const std::string lookup_error_code(cl_int error_code)
{
    // look up error codes as defined in cl.hpp
    switch (error_code) {
    case CL_SUCCESS:
        return "CL_SUCCESS";
    case CL_DEVICE_NOT_FOUND:
        return "CL_DEVICE_NOT_FOUND";
    case CL_DEVICE_NOT_AVAILABLE:
        return "CL_DEVICE_NOT_AVAILABLE";
    case CL_COMPILER_NOT_AVAILABLE:
        return "CL_COMPILER_NOT_AVAILABLE";
    case CL_MEM_OBJECT_ALLOCATION_FAILURE:
        return "CL_MEM_OBJECT_ALLOCATION_FAILURE";
    case CL_OUT_OF_RESOURCES:
        return "CL_OUT_OF_RESOURCES";
    case CL_OUT_OF_HOST_MEMORY:
        return "CL_OUT_OF_HOST_MEMORY";
    case CL_PROFILING_INFO_NOT_AVAILABLE:
        return "CL_PROFILING_INFO_NOT_AVAILABLE";
    case CL_MEM_COPY_OVERLAP:
        return "CL_MEM_COPY_OVERLAP";
    case CL_IMAGE_FORMAT_MISMATCH:
        return "CL_IMAGE_FORMAT_MISMATCH";
    case CL_IMAGE_FORMAT_NOT_SUPPORTED:
        return "CL_IMAGE_FORMAT_NOT_SUPPORTED";
    case CL_BUILD_PROGRAM_FAILURE:
        return "CL_BUILD_PROGRAM_FAILURE";
    case CL_MAP_FAILURE:
        return "CL_MAP_FAILURE";
    case CL_MISALIGNED_SUB_BUFFER_OFFSET:
        return "CL_MISALIGNED_SUB_BUFFER_OFFSET";
    case CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST:
        return "CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST";
    case CL_COMPILE_PROGRAM_FAILURE:
        return "CL_COMPILE_PROGRAM_FAILURE";
    case CL_LINKER_NOT_AVAILABLE:
        return "CL_LINKER_NOT_AVAILABLE";
    case CL_LINK_PROGRAM_FAILURE:
        return "CL_LINK_PROGRAM_FAILURE";
    case CL_DEVICE_PARTITION_FAILED:
        return "CL_DEVICE_PARTITION_FAILED";
    case CL_KERNEL_ARG_INFO_NOT_AVAILABLE:
        return "CL_KERNEL_ARG_INFO_NOT_AVAILABLE";

    case CL_INVALID_VALUE:
        return "CL_INVALID_VALUE";
    case CL_INVALID_DEVICE_TYPE:
        return "CL_INVALID_DEVICE_TYPE";
    case CL_INVALID_PLATFORM:
        return "CL_INVALID_PLATFORM";
    case CL_INVALID_DEVICE:
        return "CL_INVALID_DEVICE";
    case CL_INVALID_CONTEXT:
        return "CL_INVALID_CONTEXT";
    case CL_INVALID_QUEUE_PROPERTIES:
        return "CL_INVALID_QUEUE_PROPERTIES";
    case CL_INVALID_COMMAND_QUEUE:
        return "CL_INVALID_COMMAND_QUEUE";
    case CL_INVALID_HOST_PTR:
        return "CL_INVALID_HOST_PTR";
    case CL_INVALID_MEM_OBJECT:
        return "CL_INVALID_MEM_OBJECT";
    case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:
        return "CL_INVALID_IMAGE_FORMAT_DESCRIPTOR";
    case CL_INVALID_IMAGE_SIZE:
        return "CL_INVALID_IMAGE_SIZE";
    case CL_INVALID_SAMPLER:
        return "CL_INVALID_SAMPLER";
    case CL_INVALID_BINARY:
        return "CL_INVALID_BINARY";
```

The lookup_error_code() function takes an error code of type cl_int as an argument and returns a string representing the error code. The function contains a switch statement that maps each error code to its corresponding string representation as defined in the OpenCL header file cl.hpp.

If the error code is not defined in the switch statement, the function returns a string "Unknown error code". This function is used to provide a more user-friendly output when an OpenCL error occurs by converting the error code into a human-readable format.

Task1.cpp

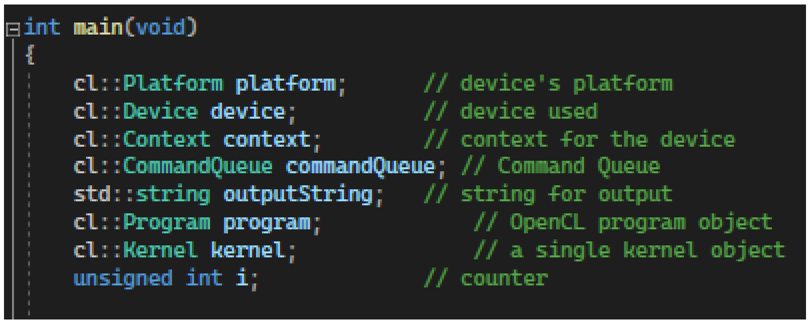
Declaring function and constants



```
void printArray(int[], int);

//fixed size for array1 and array2 and output
const int ARRAY_SIZE = 8;
const int ARRAY_SIZE2 = 16;
const int OUTPUT_SIZE = 32;
```

The function printArray is declared with a return type of void and takes two parameters: an array of integers int[] and an integer int.
The three constant variables ARRAY_SIZE, ARRAY_SIZE2, and OUTPUT_SIZE are defined and assigned specific values.

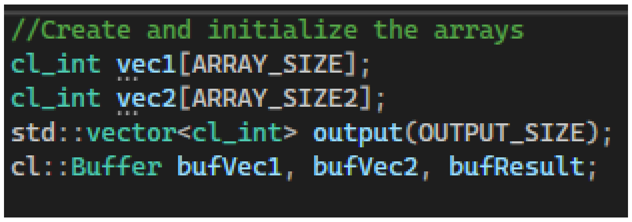
Declaring a number of data structures and variables



```
int main(void)
{
    cl::Platform platform;      // device's platform
    cl::Device device;          // device used
    cl::Context context;        // context for the device
    cl::CommandQueue commandQueue; // Command Queue
    std::string outputString;   // string for output
    cl::Program program;            // OpenCL program object
    cl::Kernel kernel;              // a single kernel object
    unsigned int i;             // counter
```

These lines declare variables that will be used throughout the program, such as platform, device, context, command queue, and program.

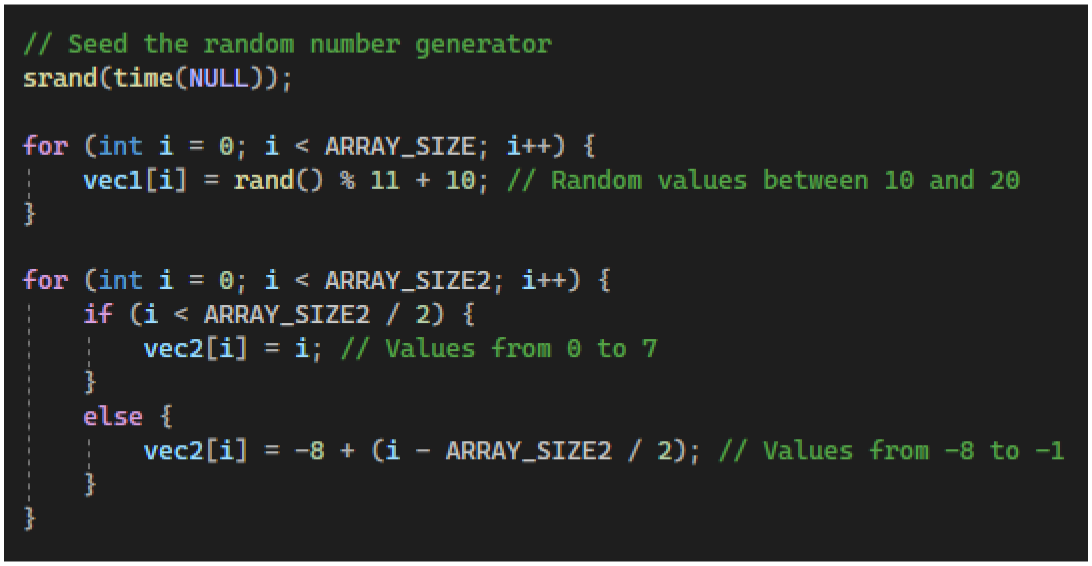
Declaring arrays and buffers



```
//Create and initialize the arrays
cl_int vec1[ARRAY_SIZE];
cl_int vec2[ARRAY_SIZE2];
std::vector<cl_int> output(OUTPUT_SIZE);
cl::Buffer bufVec1, bufVec2, bufResult;
```

Array for vec1 of size 8, array for vec2 of size 12 and the vector of output of size 32
Declare three OpenCL buffer objects: bufVec1, bufVec2, and bufResult.

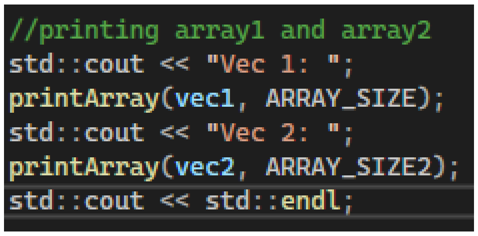
Initializing arrays



```
// Seed the random number generator
srand(time(NULL));

for (int i = 0; i < ARRAY_SIZE; i++) {
    vec1[i] = rand() % 11 + 10; // Random values between 10 and 20
}

for (int i = 0; i < ARRAY_SIZE2; i++) {
    if (i < ARRAY_SIZE2 / 2) {
        vec2[i] = i; // Values from 0 to 7
    }
    else {
        vec2[i] = -8 + (i - ARRAY_SIZE2 / 2); // Values from -8 to -1
    }
}
```

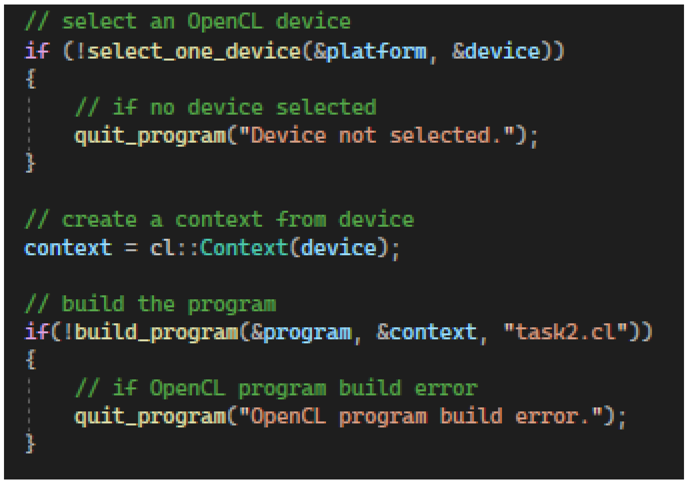
By using the current time as the seed, you ensure that each time you run the program, the random numbers generated will be different.
vec1: An 8-element array of ints that contains random values between 10 and 20.
vec2: A 16-element array of ints. Initialise the first half of the array with values from 0 to 7 (i.e. 0, 1, 2,…, 7) and the second half with values from -8 to -1 (i.e. -8, -7, -6…, -1).

Print array values



```
//printing array1 and array2
std::cout << "Vec 1: ";
printArray(vec1, ARRAY_SIZE);
std::cout << "Vec 2: ";
printArray(vec2, ARRAY_SIZE2);
std::cout << std::endl;
```

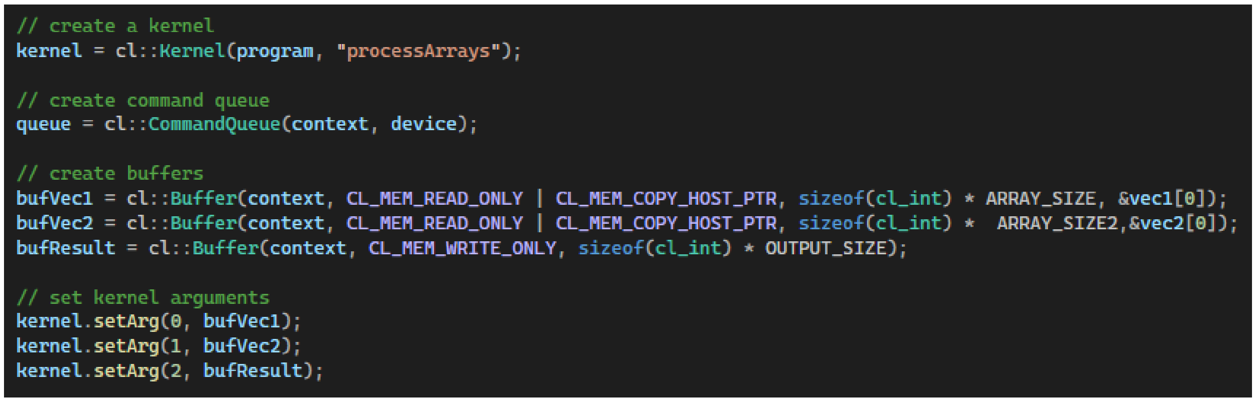
Select Device , Creating context and build program



```
// select an OpenCL device
if (!select_one_device(&platform, &device))
{
    // if no device selected
    quit_program("Device not selected.");
}

// create a context from device
context = cl::Context(device);

// build the program
if(!build_program(&program, &context, "task2.cl"))
{
    // if OpenCL program build error
    quit_program("OpenCL program build error.");
}
```

This section of code attempts to select an OpenCL device. If no device is selected, the program will terminate with an error message. The context is constructed then the program is built using the build_program function

Kernel creation & buffer creation & set kernel arguments



```
// create a kernel
kernel = cl::Kernel(program, "processArrays");

// create command queue
queue = cl::CommandQueue(context, device);

// create buffers
bufVec1 = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_int) * ARRAY_SIZE, &vec1[0]);
bufVec2 = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_int) *  ARRAY_SIZE2,&vec2[0]);
bufResult = cl::Buffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_int) * OUTPUT_SIZE);

// set kernel arguments
kernel.setArg(0, bufVec1);
kernel.setArg(1, bufVec2);
kernel.setArg(2, bufResult);
```

kernel object is created using the OpenCL program object program and the kernel name "task2".

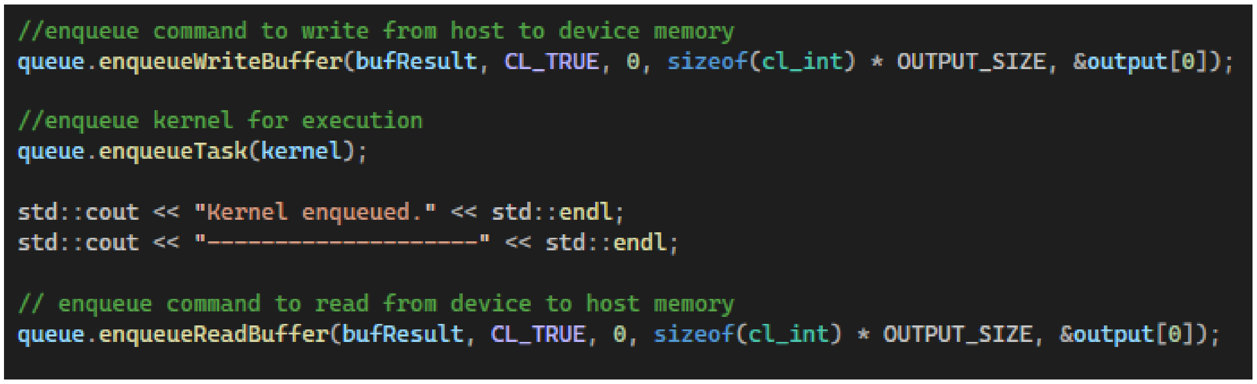
The command queue is created using the cl::CommandQueue constructor with the context and device as arguments.Creates an OpenCL buffer named bufVec1 for read-only memory, and the data is copied from the host (vec1) to the device.
The buffer size is determined by multiplying the size of cl_int by ARRAY_SIZE.

Creates an OpenCL buffer named bufVec2 for read-only memory, and the data is copied from the host (vec2) to the device.
The buffer size is determined by multiplying the size of cl_int by ARRAY_SIZE2.

Creates an OpenCL buffer named bufResult for write-only memory, used to store the computation result.
The buffer size is determined by multiplying the size of cl_int by OUTPUT_SIZE.

Sets the first argument of an OpenCL kernel named kernel to bufVec1.
Sets the second argument of the OpenCL kernel to bufVec2.
Sets the third argument of the OpenCL kernel to bufResult.

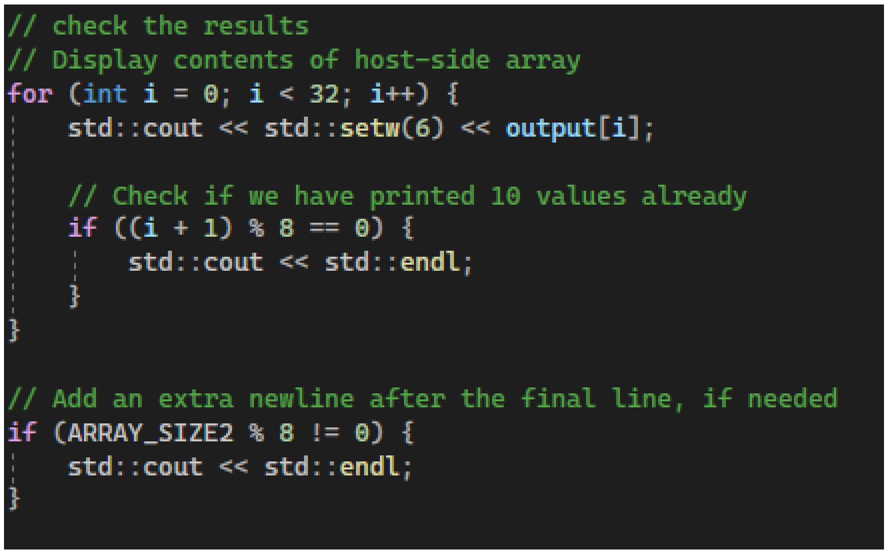
Enqueue



```
//enqueue command to write from host to device memory
queue.enqueueWriteBuffer(bufResult, CL_TRUE, 0, sizeof(cl_int) * OUTPUT_SIZE, &output[0]);

//enqueue kernel for execution
queue.enqueueTask(kernel);

std::cout << "Kernel enqueued." << std::endl;
std::cout << "--------------------" << std::endl;

// enqueue command to read from device to host memory
queue.enqueueReadBuffer(bufResult, CL_TRUE, 0, sizeof(cl_int) * OUTPUT_SIZE, &output[0]);
```

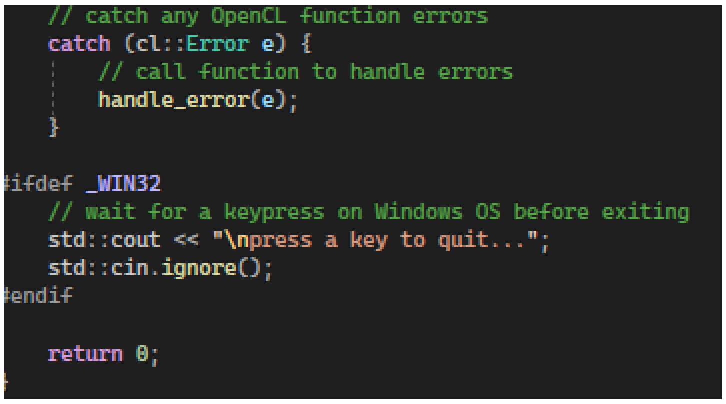
Enqueues the kernel for execution on the device.
This command starts the execution of the OpenCL kernel kernel.
Prints the message "Kernel enqueued." to the console.
Enqueues a command to read data from the device memory (bufResult) to the host memory (output).

Print Results



```
// check the results
// Display contents of host-side array
for (int i = 0; i < 32; i++) {
    std::cout << std::setw(6) << output[i];

    // Check if we have printed 10 values already
    if ((i + 1) % 8 == 0) {
        std::cout << std::endl;
    }
}

// Add an extra newline after the final line, if needed
if (ARRAY_SIZE2 % 8 != 0) {
    std::cout << std::endl;
}
```

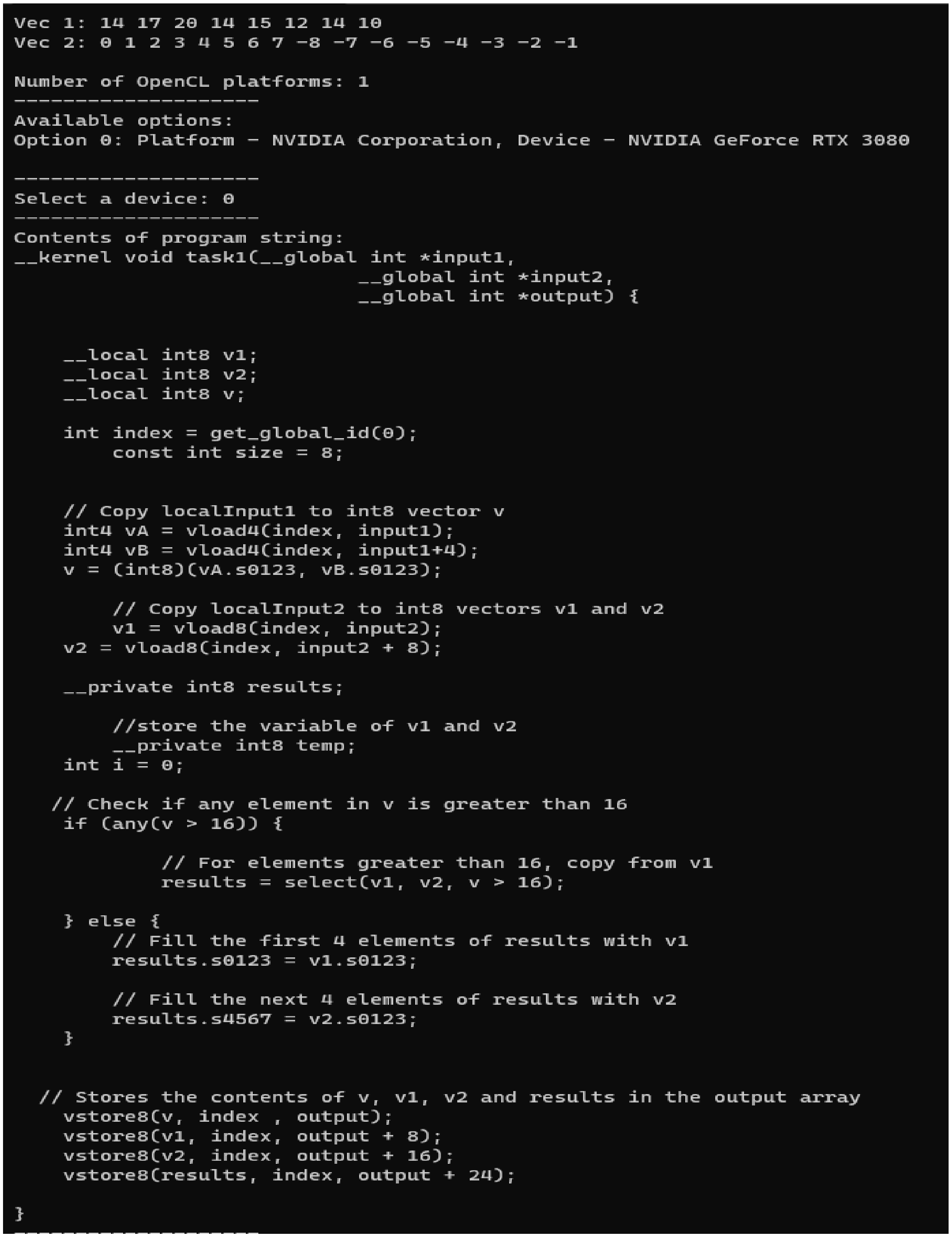
Error handling and quitting program



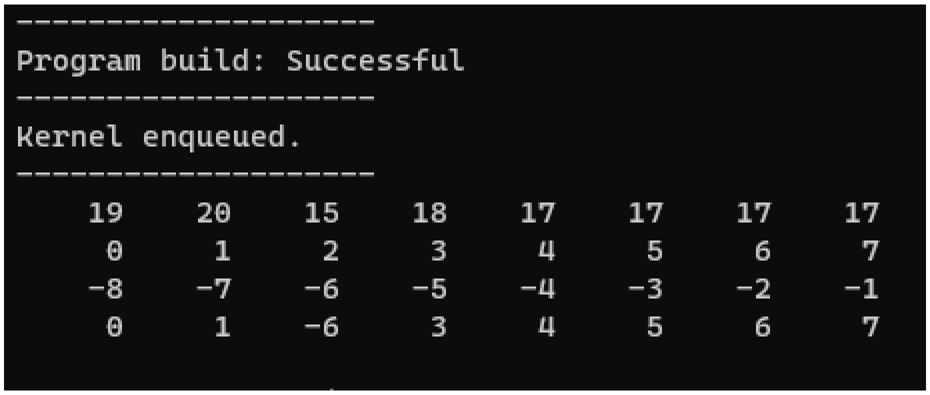
```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

Task 1 Program screenshots

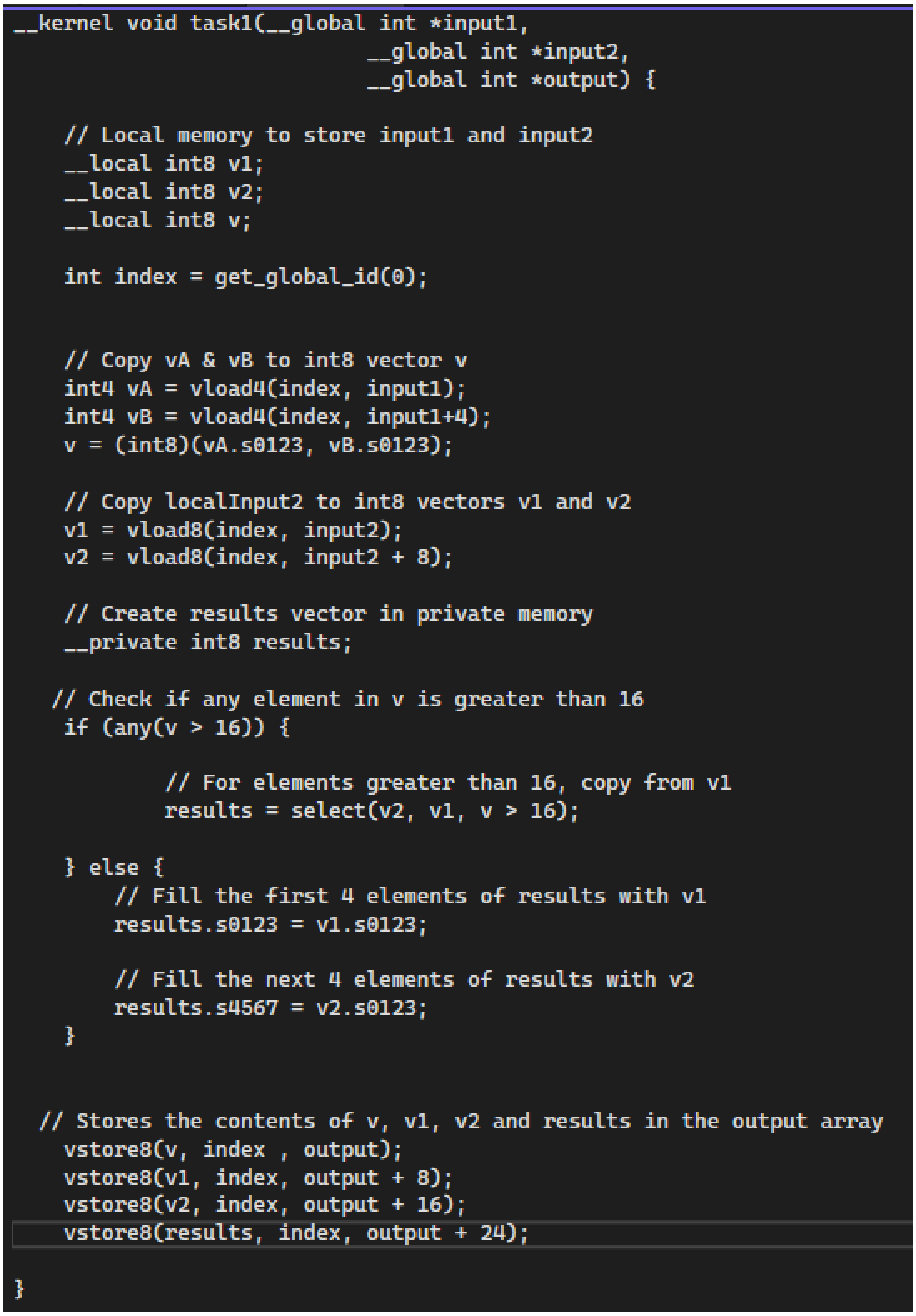


```
Vec 1: 14 17 20 14 15 12 14 10
Vec 2: 0 1 2 3 4 5 6 7 -8 -7 -6 -5 -4 -3 -2 -1

Number of OpenCL platforms: 1
--------------------
Available options:
Option 0: Platform - NVIDIA Corporation, Device - NVIDIA GeForce RTX 3080

--------------------
Select a device: 0
--------------------
Contents of program string:
__kernel void task1(__global int *input1,
                            __global int *input2,
                            __global int *output) {


    __local int8 v1;
    __local int8 v2;
    __local int8 v;

    int index = get_global_id(0);
        const int size = 8;


    // Copy localInput1 to int8 vector v
    int4 vA = vload4(index, input1);
    int4 vB = vload4(index, input1+4);
    v = (int8)(vA.s0123, vB.s0123);

        // Copy localInput2 to int8 vectors v1 and v2
        v1 = vload8(index, input2);
    v2 = vload8(index, input2 + 8);

    __private int8 results;

        //store the variable of v1 and v2
        __private int8 temp;
    int i = 0;

   // Check if any element in v is greater than 16
    if (any(v > 16)) {

            // For elements greater than 16, copy from v1
            results = select(v1, v2, v > 16);

    } else {
        // Fill the first 4 elements of results with v1
        results.s0123 = v1.s0123;

        // Fill the next 4 elements of results with v2
        results.s4567 = v2.s0123;
    }


  // Stores the contents of v, v1, v2 and results in the output array
    vstore8(v, index , output);
    vstore8(v1, index, output + 8);
    vstore8(v2, index, output + 16);
    vstore8(results, index, output + 24);

}
--------------------
```



```
--------------------
Program build: Successful
--------------------
Kernel enqueued.
--------------------
    19    20    15    18    17    17    17    17
     0     1     2     3     4     5     6     7
    -8    -7    -6    -5    -4    -3    -2    -1
     0     1    -6     3     4     5     6     7
```

Task1.cl



```
__kernel void task1(__global int *input1,
                            __global int *input2,
                            __global int *output) {

    // Local memory to store input1 and input2
    __local int8 v1;
    __local int8 v2;
    __local int8 v;

    int index = get_global_id(0);


    // Copy vA & vB to int8 vector v
    int4 vA = vload4(index, input1);
    int4 vB = vload4(index, input1+4);
    v = (int8)(vA.s0123, vB.s0123);

    // Copy localInput2 to int8 vectors v1 and v2
    v1 = vload8(index, input2);
    v2 = vload8(index, input2 + 8);

    // Create results vector in private memory
    __private int8 results;

   // Check if any element in v is greater than 16
    if (any(v > 16)) {

            // For elements greater than 16, copy from v1
            results = select(v2, v1, v > 16);

    } else {
        // Fill the first 4 elements of results with v1
        results.s0123 = v1.s0123;

        // Fill the next 4 elements of results with v2
        results.s4567 = v2.s0123;
    }


  // Stores the contents of v, v1, v2 and results in the output array
    vstore8(v, index , output);
    vstore8(v1, index, output + 8);
    vstore8(v2, index, output + 16);
    vstore8(results, index, output + 24);

}
```

The kernel function is defined with the name "task1" and takes three input arguments: input1, input2, and output. These arguments are pointers to global memory, denoted by the __global qualifier.

Local variables v1, v2, and v are declared using the __local qualifier. Local memory is shared among work-items within a work-group and can be accessed faster than global memory.

The index variable is assigned the global ID of the work-item executing the kernel. It is used to determine the element of the input arrays that each work-item will process.

The code then proceeds to copy the data from input1 to the int8 vector v. It uses two int4 vectors, vA and vB, to load 4 elements each from input1 and then combines them into an int8 vector v.

Next, the code copies the data from input2 to the int8 vectors v1 and v2 using the vload8 function. It loads 8 elements from input2 starting from the current index.

A private variable results of type int8 is declared. It will store the results of the computation.

Another private int8 variable temp is declared, but it is not used in the code and seems to be unnecessary.

The code checks if any element in v is greater than 16 using the any function. If the condition is true for any element, it proceeds to the if-block.

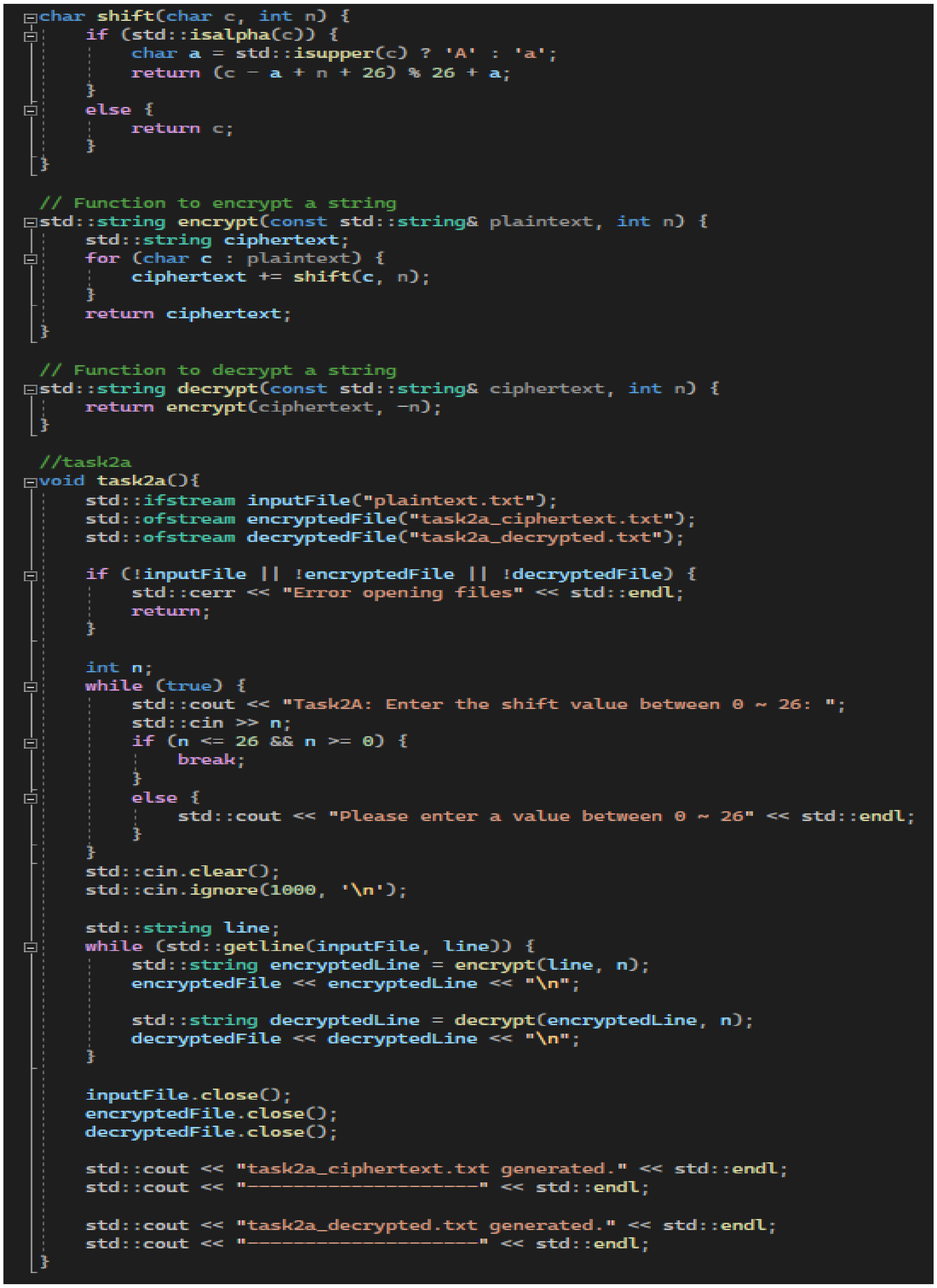
Inside the if-block, the select function is used to copy elements from v1 or v2 to results based on the condition v > 16. If an element in v is greater than 16, the corresponding element from v1 is copied to results; otherwise, the element from v2 is copied.

If the condition in step 9 is false (i.e., no element in v is greater than 16), the else-block is executed. It fills the first 4 elements of results with the elements from v1 and the next 4 elements with the elements from v2.

Finally, the contents of v, v1, v2, and results are stored back to the output array using the vstore8 function. The index is used to determine the position in the output array where each vector will be stored.

Task2.cpp

Task2a



```cpp
char shift(char c, int n) {
    if (std::isalpha(c)) {
        char a = std::isupper(c) ? 'A' : 'a';
        return (c - a + n + 26) % 26 + a;
    }
    else {
        return c;
    }
}

// Function to encrypt a string
std::string encrypt(const std::string& plaintext, int n) {
    std::string ciphertext;
    for (char c : plaintext) {
        ciphertext += shift(c, n);
    }
    return ciphertext;
}

// Function to decrypt a string
std::string decrypt(const std::string& ciphertext, int n) {
    return encrypt(ciphertext, -n);
}

//task2a
void task2a(){
    std::ifstream inputFile("plaintext.txt");
    std::ofstream encryptedFile("task2a_ciphertext.txt");
    std::ofstream decryptedFile("task2a_decrypted.txt");

    if (!inputFile || !encryptedFile || !decryptedFile) {
        std::cerr << "Error opening files" << std::endl;
        return;
    }

    int n;
    while (true) {
        std::cout << "Task2A: Enter the shift value between 0 ~ 26: ";
        std::cin >> n;
        if (n <= 26 && n >= 0) {
            break;
        }
        else {
            std::cout << "Please enter a value between 0 ~ 26" << std::endl;
        }
    }
    std::cin.clear();
    std::cin.ignore(1000, '\n');

    std::string line;
    while (std::getline(inputFile, line)) {
        std::string encryptedLine = encrypt(line, n);
        encryptedFile << encryptedLine << "\n";

        std::string decryptedLine = decrypt(encryptedLine, n);
        decryptedFile << decryptedLine << "\n";
    }

    inputFile.close();
    encryptedFile.close();
    decryptedFile.close();

    std::cout << "task2a_ciphertext.txt generated." << std::endl;
    std::cout << "---------------------" << std::endl;

    std::cout << "task2a_decrypted.txt generated." << std::endl;
    std::cout << "---------------------" << std::endl;
}
```

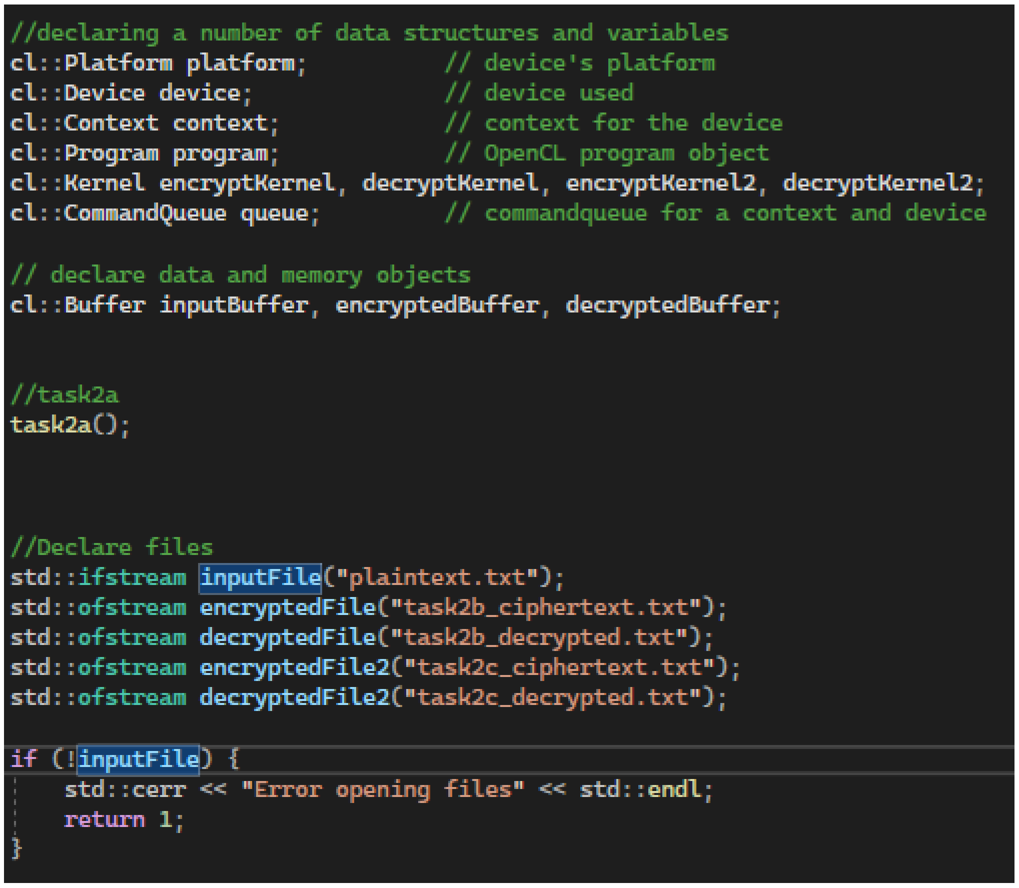
- The shift function takes two parameters: c (the character to be shifted) and n (the shift amount).
  It checks if c is an alphabetic character using the std::isalpha function.
  If c is alphabetic:
  It determines whether c is uppercase or lowercase using the std::isupper function.
  It calculates a shift value a based on the case of c.
  It performs the character shift operation using the formula (c - a + n + 26) % 26 + a, where n is the shift amount. This formula shifts the character by n positions in the alphabet, wrapping around if necessary.
  The shifted character is returned.
  If c is not alphabetic, it is returned as is.

- The encrypt function takes two parameters: plaintext (the string to be encrypted) and n (the shift amount).
  It initializes an empty string, ciphertext, to store the encrypted result.
  It iterates over each character c in the plaintext string.
  For each character, it calls the shift function to perform the character shift operation with c and n.
  The shifted character is appended to the ciphertext string.
  After processing all characters in the plaintext string, the resulting ciphertext string is returned as the encrypted result.

- The decrypt function takes two parameters: ciphertext (the string to be decrypted) and n (the shift amount).
  It calls the encrypt function with the ciphertext string and a negated value of n (-n).
  The encrypt function performs the encryption operation in reverse, effectively decrypting the ciphertext string.
  The resulting decrypted string is returned as the output.

- The task2a function opens the input, encrypted output, and decrypted output files.
  It prompts the user to enter a shift value between 0 and 26.
  It reads each line from the input file.
  For each line, it encrypts the line using the encrypt function with the provided shift value and writes the encrypted line to the "task2a_ciphertext.txt" file.
  It also decrypts the encrypted line using the decrypt function with the same shift value and writes the decrypted line to the "task2a_decrypted.txt" file.
  After processing all lines, it closes all file streams.
  Finally, it displays messages indicating the successful generation of the ciphertext and decrypted files.

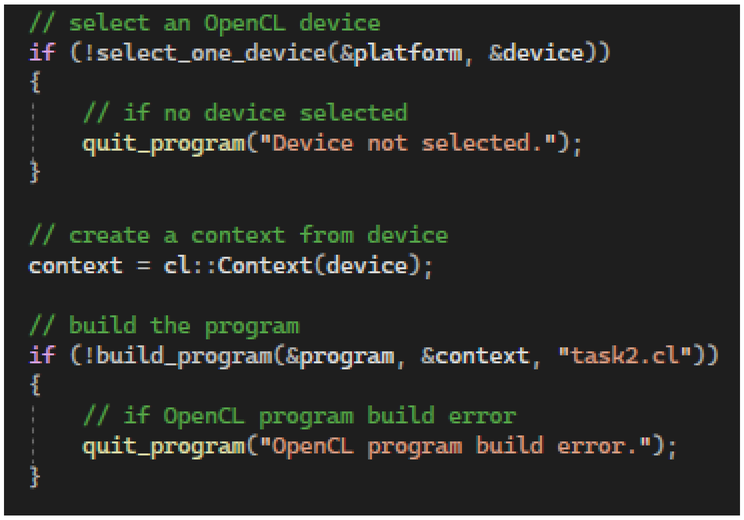
declaring a number of data structures and variables



```
//declaring a number of data structures and variables
cl::Platform platform;          // device's platform
cl::Device device;              // device used
cl::Context context;            // context for the device
cl::Program program;            // OpenCL program object
cl::Kernel encryptKernel, decryptKernel, encryptKernel2, decryptKernel2;
cl::CommandQueue queue;         // commandqueue for a context and device

// declare data and memory objects
cl::Buffer inputBuffer, encryptedBuffer, decryptedBuffer;


//task2a
task2a();



//Declare files
std::ifstream inputFile("plaintext.txt");
std::ofstream encryptedFile("task2b_ciphertext.txt");
std::ofstream decryptedFile("task2b_decrypted.txt");
std::ofstream encryptedFile2("task2c_ciphertext.txt");
std::ofstream decryptedFile2("task2c_decrypted.txt");

if (!inputFile) {
    std::cerr << "Error opening files" << std::endl;
    return 1;
}
```

Declares several data structures and variables of OpenCL classes, such as Platform, Device, Context, Program, Kernel, and CommandQueue
Declare files required for input and output
Check for file opening error

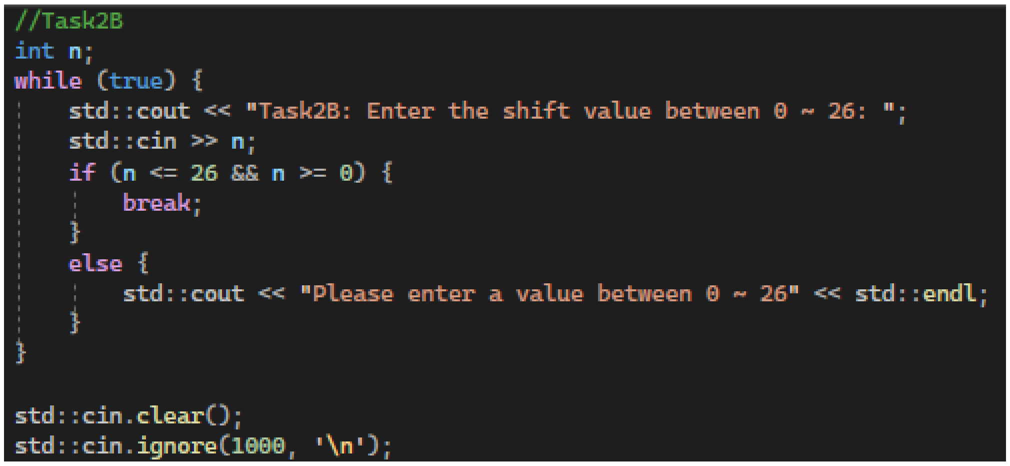
Select Device , Creating context and build program



```
// select an OpenCL device
if (!select_one_device(&platform, &device))
{
    // if no device selected
    quit_program("Device not selected.");
}

// create a context from device
context = cl::Context(device);

// build the program
if (!build_program(&program, &context, "task2.cl"))
{
    // if OpenCL program build error
    quit_program("OpenCL program build error.");
}
```

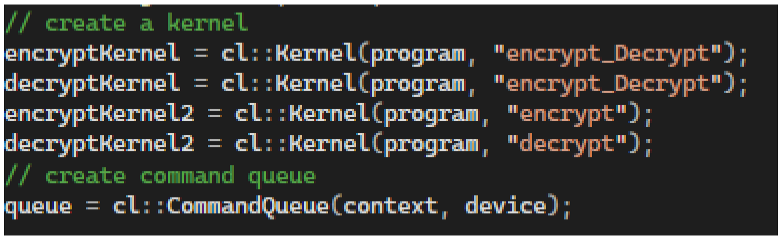
This section of code attempts to select an OpenCL device. If no device is selected, the program will terminate with an error message. The context is constructed then the program is built using the build_program function

Check if N input is between 0 - 26



```
//Task2B
int n;
while (true) {
    std::cout << "Task2B: Enter the shift value between 0 ~ 26: ";
    std::cin >> n;
    if (n <= 26 && n >= 0) {
        break;
    }
    else {
        std::cout << "Please enter a value between 0 ~ 26" << std::endl;
    }
}

std::cin.clear();
std::cin.ignore(1000, '\n');
```

Kernel creation & command queue

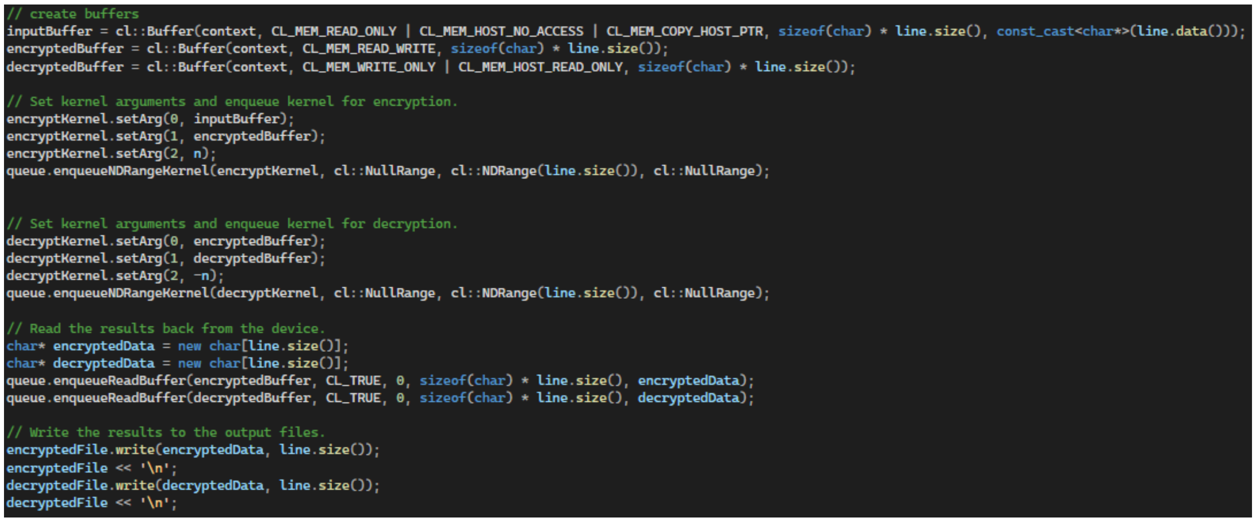


```
// create a kernel
encryptKernel = cl::Kernel(program, "encrypt_Decrypt");
decryptKernel = cl::Kernel(program, "encrypt_Decrypt");
encryptKernel2 = cl::Kernel(program, "encrypt");
decryptKernel2 = cl::Kernel(program, "decrypt");
// create command queue
queue = cl::CommandQueue(context, device);
```

kernel object is created using the OpenCL program object program and the kernel name "encrypt_Decrypt, encrypt, decrypt".

The command queue is created using the cl::CommandQueue constructor with the context and device as arguments.

Task2B



```
// create buffers
inputBuffer = cl::Buffer(context, CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS | CL_MEM_COPY_HOST_PTR, sizeof(char) * line.size(), const_cast<char*>(line.data()));
encryptedBuffer = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(char) * line.size());
decryptedBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY, sizeof(char) * line.size());

// Set kernel arguments and enqueue kernel for encryption.
encryptKernel.setArg(0, inputBuffer);
encryptKernel.setArg(1, encryptedBuffer);
encryptKernel.setArg(2, n);
queue.enqueueNDRangeKernel(encryptKernel, cl::NullRange, cl::NDRange(line.size()), cl::NullRange);


// Set kernel arguments and enqueue kernel for decryption.
decryptKernel.setArg(0, encryptedBuffer);
decryptKernel.setArg(1, decryptedBuffer);
decryptKernel.setArg(2, -n);
queue.enqueueNDRangeKernel(decryptKernel, cl::NullRange, cl::NDRange(line.size()), cl::NullRange);

// Read the results back from the device.
char* encryptedData = new char[line.size()];
char* decryptedData = new char[line.size()];
queue.enqueueReadBuffer(encryptedBuffer, CL_TRUE, 0, sizeof(char) * line.size(), encryptedData);
queue.enqueueReadBuffer(decryptedBuffer, CL_TRUE, 0, sizeof(char) * line.size(), decryptedData);

// Write the results to the output files.
encryptedFile.write(encryptedData, line.size());
encryptedFile << '\n';
decryptedFile.write(decryptedData, line.size());
decryptedFile << '\n';
```

Buffer Creation:

The input data is stored in an OpenCL buffer (inputBuffer) that is read-only and inaccessible to the host.

Two additional buffers are created: encryptedBuffer for storing the encrypted data, and decryptedBuffer for storing the decrypted data.

Encryption:

The encryption kernel (encryptKernel) is executed with the input buffer, output buffer, and encryption key (n) set as arguments.

The encryptKernel is enqueued to execute on the device, processing the data in parallel.

Decryption:

The decryption kernel (decryptKernel) is executed with the encrypted buffer, decrypted buffer, and decryption key (-n) set as arguments.

The decryptKernel is enqueued to execute on the device, processing the data in parallel.

Data Retrieval:

Temporary host buffers (encryptedData and decryptedData) are created to store the encrypted and decrypted data retrieved from the device's buffers.

The encrypted and decrypted data are read from the respective device buffers into the host buffers.

Output Writing:

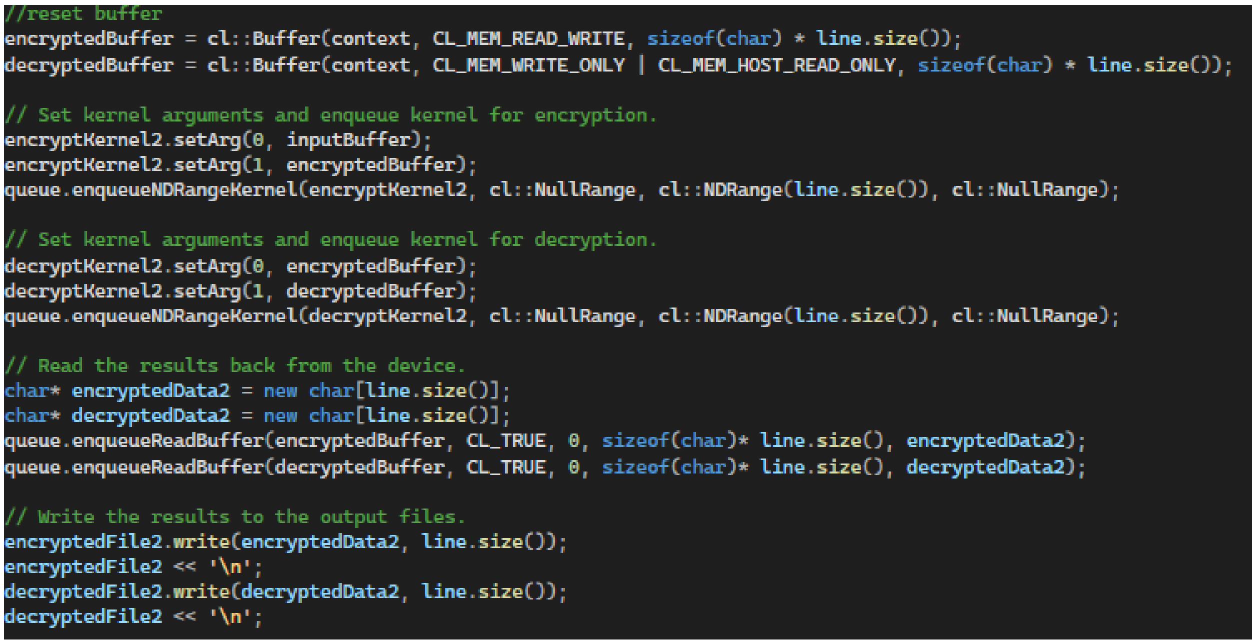
The encrypted data is written to an output file (encryptedFile) using the encryptedData buffer.

A newline character is added after writing each line of encrypted data.

Similarly, the decrypted data is written to an output file (decryptedFile) using the decryptedData buffer.

A newline character is added after writing each line of decrypted data.

Task2c



```
//reset buffer
encryptedBuffer = cl::Buffer(context, CL_MEM_READ_WRITE, sizeof(char) * line.size());
decryptedBuffer = cl::Buffer(context, CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY, sizeof(char) * line.size());

// Set kernel arguments and enqueue kernel for encryption.
encryptKernel2.setArg(0, inputBuffer);
encryptKernel2.setArg(1, encryptedBuffer);
queue.enqueueNDRangeKernel(encryptKernel2, cl::NullRange, cl::NDRange(line.size()), cl::NullRange);

// Set kernel arguments and enqueue kernel for decryption.
decryptKernel2.setArg(0, encryptedBuffer);
decryptKernel2.setArg(1, decryptedBuffer);
queue.enqueueNDRangeKernel(decryptKernel2, cl::NullRange, cl::NDRange(line.size()), cl::NullRange);

// Read the results back from the device.
char* encryptedData2 = new char[line.size()];
char* decryptedData2 = new char[line.size()];
queue.enqueueReadBuffer(encryptedBuffer, CL_TRUE, 0, sizeof(char)* line.size(), encryptedData2);
queue.enqueueReadBuffer(decryptedBuffer, CL_TRUE, 0, sizeof(char)* line.size(), decryptedData2);

// Write the results to the output files.
encryptedFile2.write(encryptedData2, line.size());
encryptedFile2 << '\n';
decryptedFile2.write(decryptedData2, line.size());
decryptedFile2 << '\n';
```

Buffer Reset:

The encryptedBuffer is recreated as a read-write buffer.

The decryptedBuffer is recreated as a write-only buffer that is read-only by the host.

Encryption:

The encryptKernel2 is executed with the input buffer and the reset encryptedBuffer.

The encryptKernel2 is enqueued to execute on the device.

Decryption:

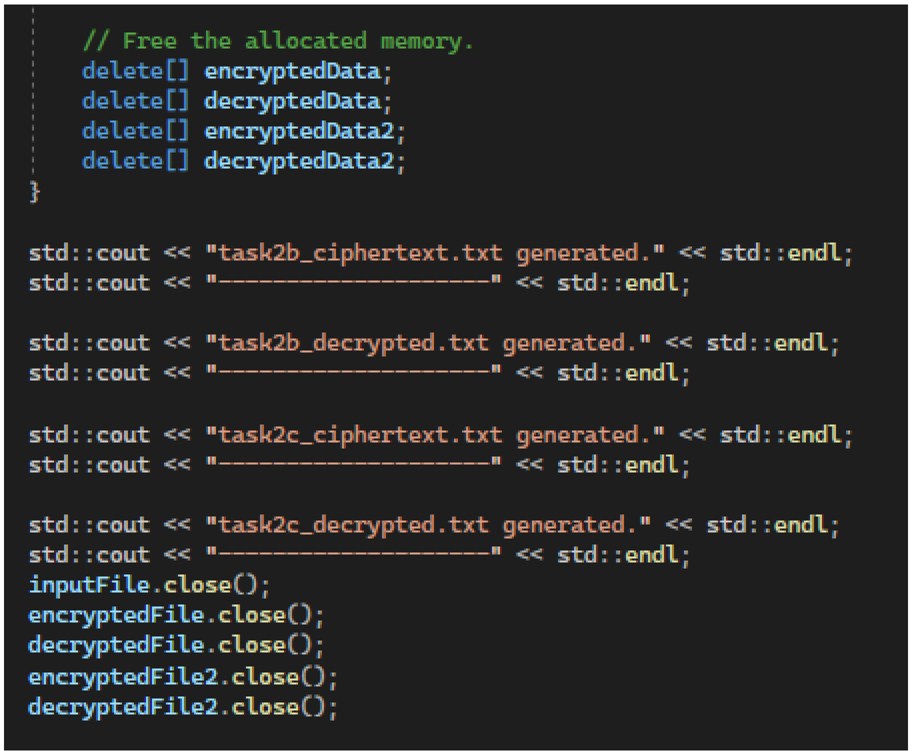
The decryptKernel2 is executed with the reset encryptedBuffer as the input buffer and the reset decryptedBuffer as the output buffer.

The decryptKernel2 is enqueued to execute on the device.

Data Retrieval:
Temporary host buffers are created to store the encrypted and decrypted data retrieved from the device's buffers.
The encrypted and decrypted data are read from the reset buffers into the host buffers.

Output Writing:
The encrypted data is written to an output file using the host buffer.
A newline character is added after writing each line of encrypted data.
Similarly, the decrypted data is written to an output file.
A newline character is added after writing each line of decrypted data.

Memory Deallocation & File Closure



```
    // Free the allocated memory.
    delete[] encryptedData;
    delete[] decryptedData;
    delete[] encryptedData2;
    delete[] decryptedData2;
}

std::cout << "task2b_ciphertext.txt generated." << std::endl;
std::cout << "--------------------" << std::endl;

std::cout << "task2b_decrypted.txt generated." << std::endl;
std::cout << "--------------------" << std::endl;

std::cout << "task2c_ciphertext.txt generated." << std::endl;
std::cout << "--------------------" << std::endl;

std::cout << "task2c_decrypted.txt generated." << std::endl;
std::cout << "--------------------" << std::endl;
inputFile.close();
encryptedFile.close();
decryptedFile.close();
encryptedFile2.close();
decryptedFile2.close();
```

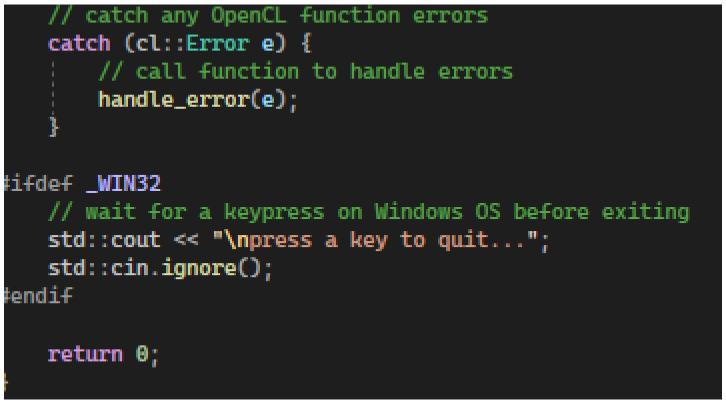
Memory Deallocation:
The memory allocated for encryptedData and decryptedData arrays is freed.
Similarly, the memory allocated for encryptedData2 and decryptedData2 arrays is also freed.

Output Messages:
Messages are displayed to indicate the successful generation of "task2b_ciphertext.txt", "task2b_decrypted.txt", "task2c_ciphertext.txt", and "task2c_decrypted.txt" files.

File Closure:

The file streams for the input file, encrypted output file, decrypted output file, additional encrypted output file, and additional decrypted output file are closed.

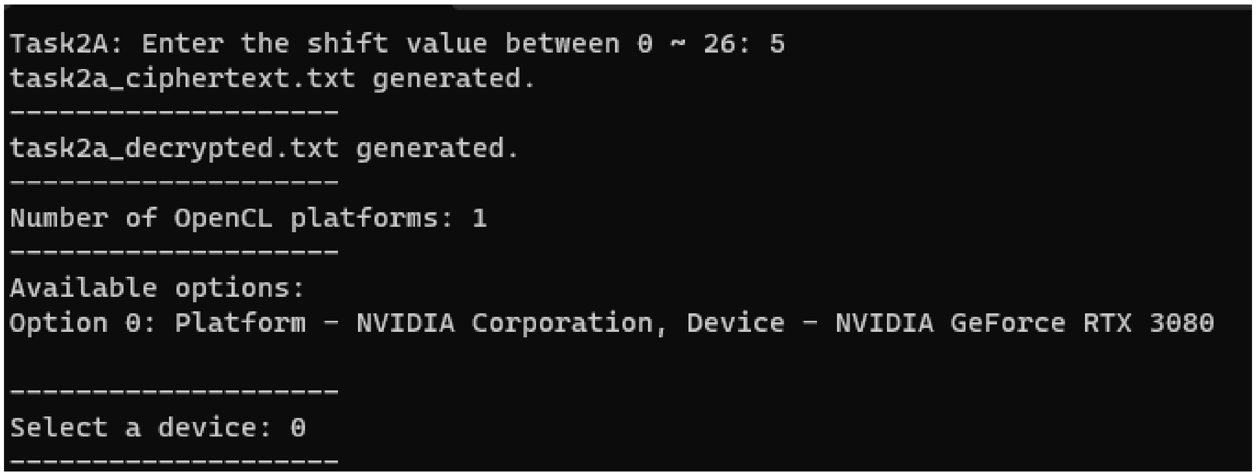
Error handling and quitting program



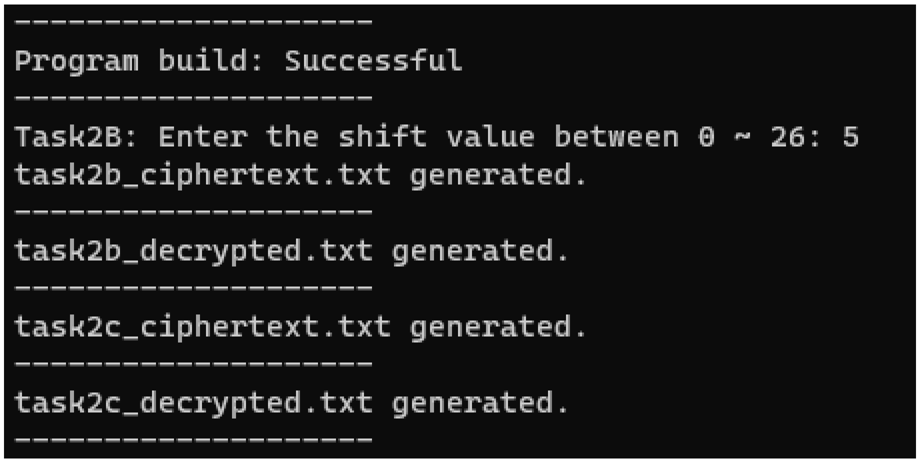
```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

Task 2 Program screenshots

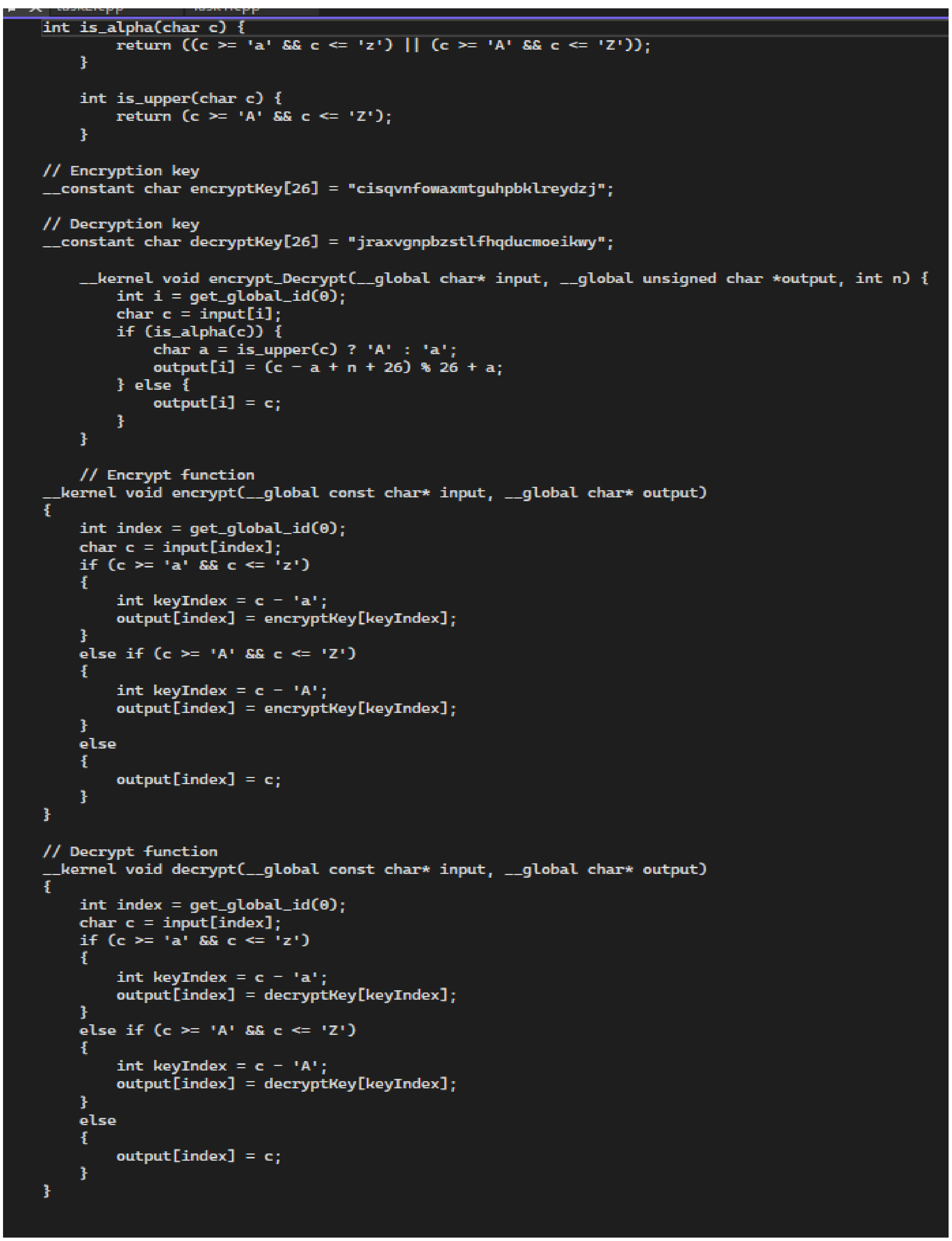


```
Task2A: Enter the shift value between 0 ~ 26: 5
task2a_ciphertext.txt generated.
--------------------
task2a_decrypted.txt generated.
--------------------
Number of OpenCL platforms: 1
--------------------
Available options:
Option 0: Platform - NVIDIA Corporation, Device - NVIDIA GeForce RTX 3080

--------------------
Select a device: 0
--------------------
```



```
--------------------
Program build: Successful
--------------------
Task2B: Enter the shift value between 0 ~ 26: 5
task2b_ciphertext.txt generated.
--------------------
task2b_decrypted.txt generated.
--------------------
task2c_ciphertext.txt generated.
--------------------
task2c_decrypted.txt generated.
--------------------
```

Task2.cl



```
int is_alpha(char c) {
        return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
    }

    int is_upper(char c) {
        return (c >= 'A' && c <= 'Z');
    }

// Encryption key
__constant char encryptKey[26] = "cisqvnfowaxmtguhpbklreydzj";

// Decryption key
__constant char decryptKey[26] = "jraxvgnpbzstlfhqducmoeikwy";

    __kernel void encrypt_Decrypt(__global char* input, __global unsigned char *output, int n) {
        int i = get_global_id(0);
        char c = input[i];
        if (is_alpha(c)) {
            char a = is_upper(c) ? 'A' : 'a';
            output[i] = (c - a + n + 26) % 26 + a;
        } else {
            output[i] = c;
        }
    }

    // Encrypt function
__kernel void encrypt(__global const char* input, __global char* output)
{
    int index = get_global_id(0);
    char c = input[index];
    if (c >= 'a' && c <= 'z')
    {
        int keyIndex = c - 'a';
        output[index] = encryptKey[keyIndex];
    }
    else if (c >= 'A' && c <= 'Z')
    {
        int keyIndex = c - 'A';
        output[index] = encryptKey[keyIndex];
    }
    else
    {
        output[index] = c;
    }
}

// Decrypt function
__kernel void decrypt(__global const char* input, __global char* output)
{
    int index = get_global_id(0);
    char c = input[index];
    if (c >= 'a' && c <= 'z')
    {
        int keyIndex = c - 'a';
        output[index] = decryptKey[keyIndex];
    }
    else if (c >= 'A' && c <= 'Z')
    {
        int keyIndex = c - 'A';
        output[index] = decryptKey[keyIndex];
    }
    else
    {
        output[index] = c;
    }
}
```

- is_alpha(char c): This function checks whether a given character c is an alphabetic character. It returns 1 if c is a lowercase or uppercase letter (a-z or A-Z) and 0 otherwise.
  The expression ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')) evaluates to true if c is within the range of lowercase letters (a-z) or uppercase letters (A-Z).
  If c is a lowercase letter (a-z), the condition (c >= 'a' && c <= 'z') will evaluate to true.
  If c is an uppercase letter (A-Z), the condition (c >= 'A' && c <= 'Z') will evaluate to true.
  Therefore, the function returns 1 if c is an alphabetic character and 0 otherwise.

- is_upper(char c): This function checks whether a given character c is an uppercase letter. It returns 1 if c is an uppercase letter (A-Z) and 0 otherwise.
  The expression (c >= 'A' && c <= 'Z') evaluates to true if c is within the range of uppercase letters (A-Z).
  Therefore, the function returns 1 if c is an uppercase letter and 0 otherwise.

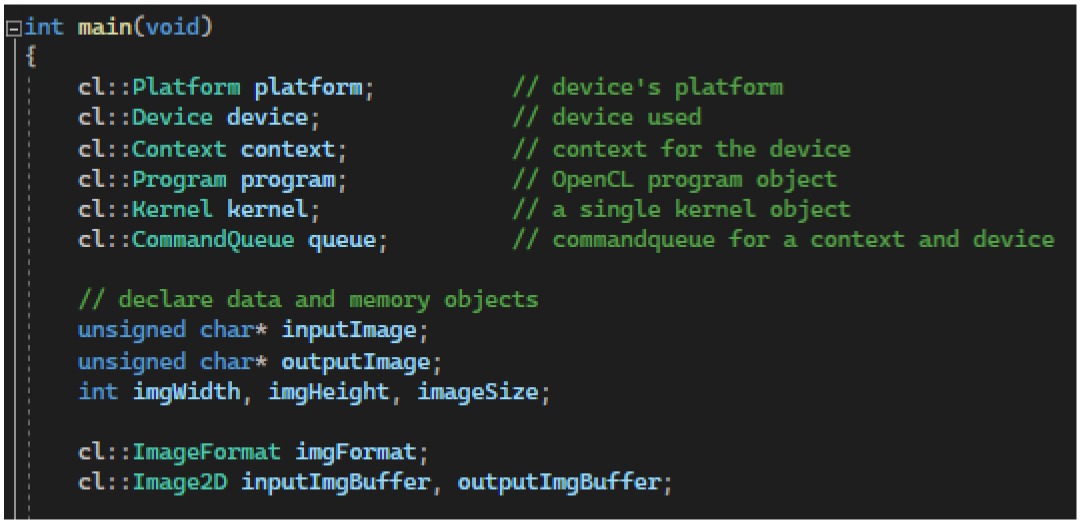
- The encrypt_Decrypt kernel function receives three parameters: input (global memory buffer containing the input string), output (global memory buffer to store the result), and n (an integer representing the encryption/decryption key).
  It operates on individual characters of the input string using the global ID of the work item.
  For each character c in the input:
  It checks if c is alphabetic using the is_alpha function.
  If c is alphabetic:
  It determines if c is uppercase or lowercase using the is_upper function.
  It calculates a shift value a based on the case of c.
  It computes the encrypted/decrypted character using the formula (c - a + n + 26) % 26 + a, where n is the encryption key.
  If c is not alphabetic, it is copied directly to the output buffer.
  The result is stored in the output buffer.

- The encrypt kernel function takes two parameters: input (global memory buffer containing the input string) and output (global memory buffer to store the encrypted string).
  It operates on individual characters of the input string using the global ID of the work item.
  For each character c in the input:
  If c is a lowercase letter (c >= 'a' && c <= 'z'), it calculates the index keyIndex by subtracting the character 'a' from c. This index represents the position of c in the alphabet.
  If c is an uppercase letter (c >= 'A' && c <= 'Z'), it calculates the index keyIndex by subtracting the character 'A' from c.

The character at index keyIndex in the encryptKey array is assigned to the corresponding position in the output buffer.
If c is not a letter, it is copied directly to the output buffer.
The resulting encrypted string is stored in the output buffer.

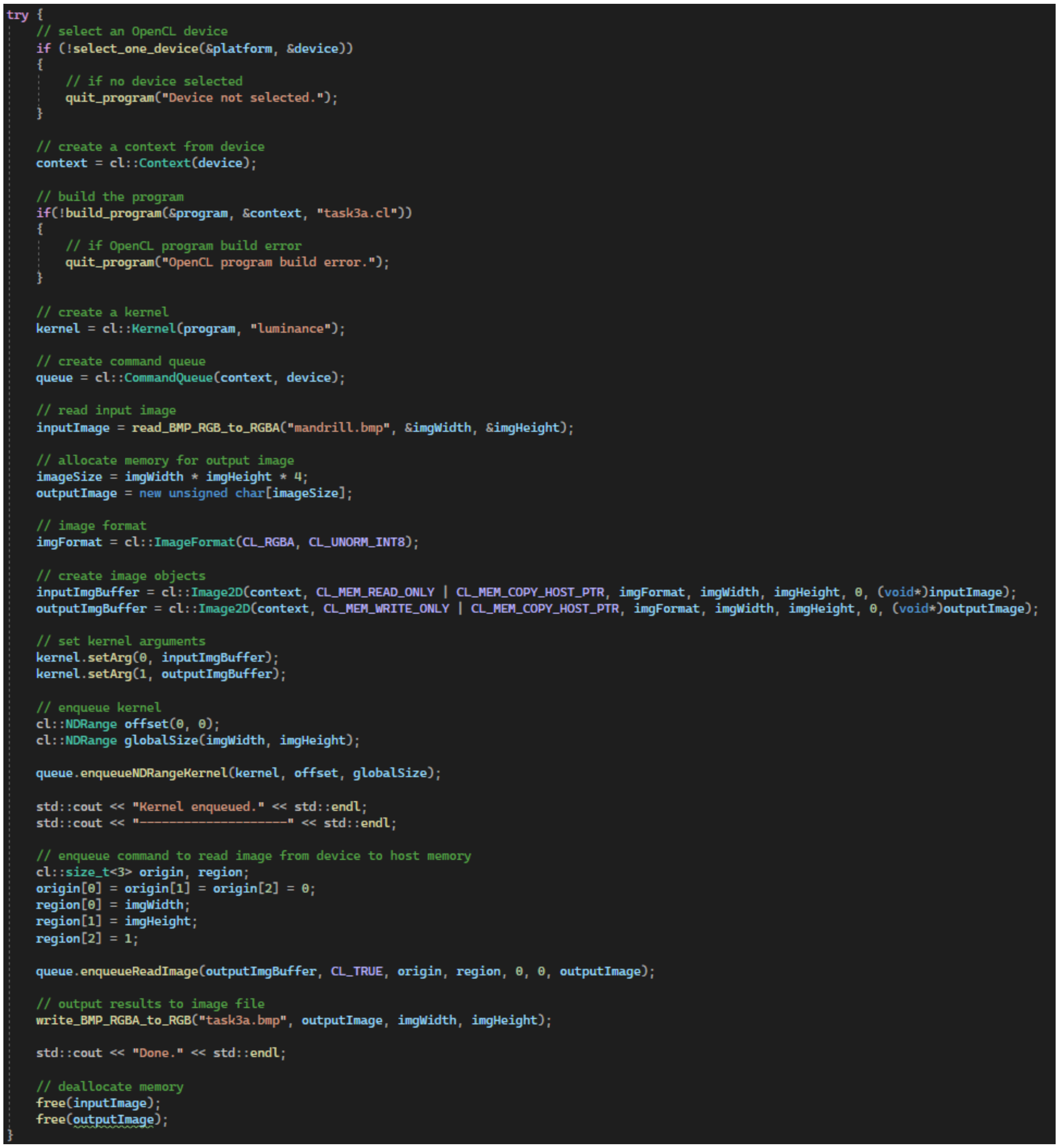
- The decrpyt kernel function acts the same as the encrypt kernel function but the decrypt key is used

Task3a.cpp



```
int main(void)
{
    cl::Platform platform;          // device's platform
    cl::Device device;              // device used
    cl::Context context;            // context for the device
    cl::Program program;            // OpenCL program object
    cl::Kernel kernel;              // a single kernel object
    cl::CommandQueue queue;         // commandqueue for a context and device

    // declare data and memory objects
    unsigned char* inputImage;
    unsigned char* outputImage;
    int imgWidth, imgHeight, imageSize;

    cl::ImageFormat imgFormat;
    cl::Image2D inputImgBuffer, outputImgBuffer;
```

Declares several data structures and variables of OpenCL classes, such as Platform, Device, Context, Program, Kernel, and CommandQueue
Declares memory objects
Declares Image format , declares Image2D buffers

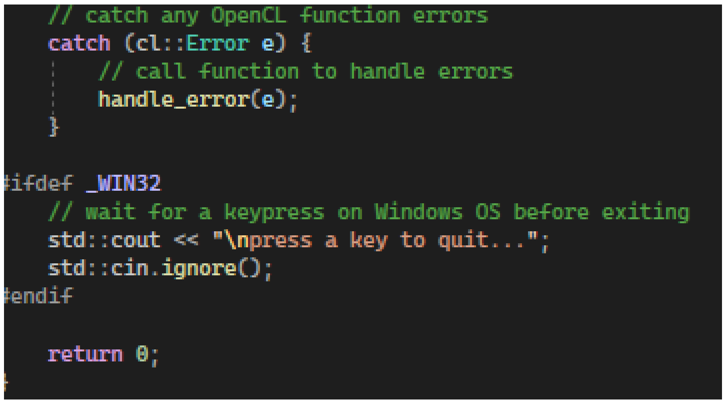


```
try {
    // select an OpenCL device
    if (!select_one_device(&platform, &device))
    {
        // if no device selected
        quit_program("Device not selected.");
    }

    // create a context from device
    context = cl::Context(device);

    // build the program
    if(!build_program(&program, &context, "task3a.cl"))
    {
        // if OpenCL program build error
        quit_program("OpenCL program build error.");
    }

    // create a kernel
    kernel = cl::Kernel(program, "luminance");

    // create command queue
    queue = cl::CommandQueue(context, device);

    // read input image
    inputImage = read_BMP_RGB_to_RGBA("mandrill.bmp", &imgWidth, &imgHeight);

    // allocate memory for output image
    imageSize = imgWidth * imgHeight * 4;
    outputImage = new unsigned char[imageSize];

    // image format
    imgFormat = cl::ImageFormat(CL_RGBA, CL_UNORM_INT8);

    // create image objects
    inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)inputImage);
    outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputImage);

    // set kernel arguments
    kernel.setArg(0, inputImgBuffer);
    kernel.setArg(1, outputImgBuffer);

    // enqueue kernel
    cl::NDRange offset(0, 0);
    cl::NDRange globalSize(imgWidth, imgHeight);

    queue.enqueueNDRangeKernel(kernel, offset, globalSize);

    std::cout << "Kernel enqueued." << std::endl;
    std::cout << "--------------------" << std::endl;

    // enqueue command to read image from device to host memory
    cl::size_t<3> origin, region;
    origin[0] = origin[1] = origin[2] = 0;
    region[0] = imgWidth;
    region[1] = imgHeight;
    region[2] = 1;

    queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputImage);

    // output results to image file
    write_BMP_RGBA_to_RGB("task3a.bmp", outputImage, imgWidth, imgHeight);

    std::cout << "Done." << std::endl;

    // deallocate memory
    free(inputImage);
    free(outputImage);
}
```

- Select an OpenCL device by calling the select_one_device function. If no device is selected, the program quits with an error message.
- Create a context from the selected device.
- Build an OpenCL program by calling the build_program function, passing the context and the filename of the OpenCL program. If there is an error in building the program, the program quits with an error message.
- Create a kernel object named "luminance" from the built program.
- Create a command queue associated with the context and device.

- Read an input image from the file "mandrill.bmp" using the read_BMP_RGB_to_RGBA function, which returns the input image data along with its width and height.
- Allocate memory for the output image based on the image dimensions.
- Define the image format using the CL_RGBA and CL_UNORM_INT8 constants.
- Create image objects inputImgBuffer and outputImgBuffer from the input image data, using the appropriate flags and image format.
- Set the kernel arguments, specifying the input and output image buffers.
- Enqueue the kernel for execution using enqueueNDRangeKernel, passing the offset and global size.
- Print a message indicating that the kernel has been enqueued.
- Enqueue a command to read the output image from the device to the host memory using enqueueReadImage.
- Write the output image data to the file "task3a.bmp" using the write_BMP_RGBA_to_RGB function.
- Print a "Done" message indicating successful completion.
- Deallocate the memory for the input and output image data.

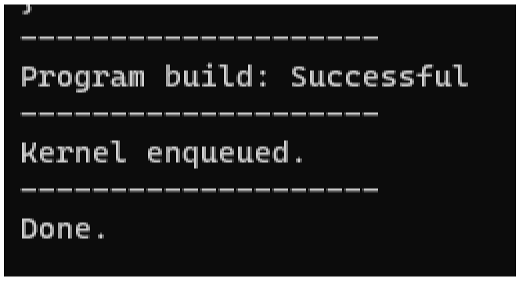
Error handling and quitting program



```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

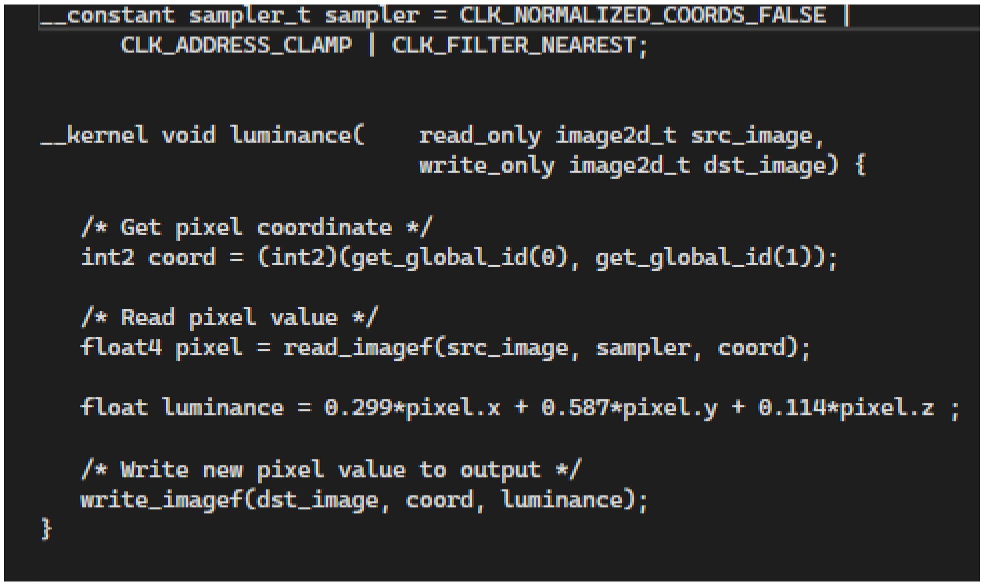
If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

Task 3a Program screenshots



```
}
--------------------
Program build: Successful
--------------------
Kernel enqueued.
--------------------
Done.
```

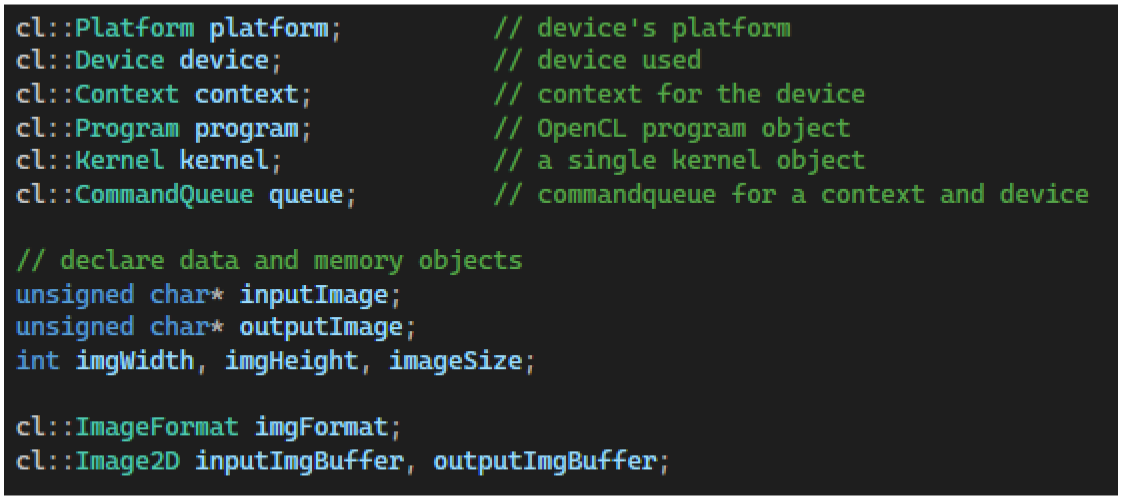
Task3a.cl



```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;


__kernel void luminance(    read_only image2d_t src_image,
                            write_only image2d_t dst_image) {

   /* Get pixel coordinate */
   int2 coord = (int2)(get_global_id(0), get_global_id(1));

   /* Read pixel value */
   float4 pixel = read_imagef(src_image, sampler, coord);

   float luminance = 0.299*pixel.x + 0.587*pixel.y + 0.114*pixel.z ;

   /* Write new pixel value to output */
   write_imagef(dst_image, coord, luminance);
}
```

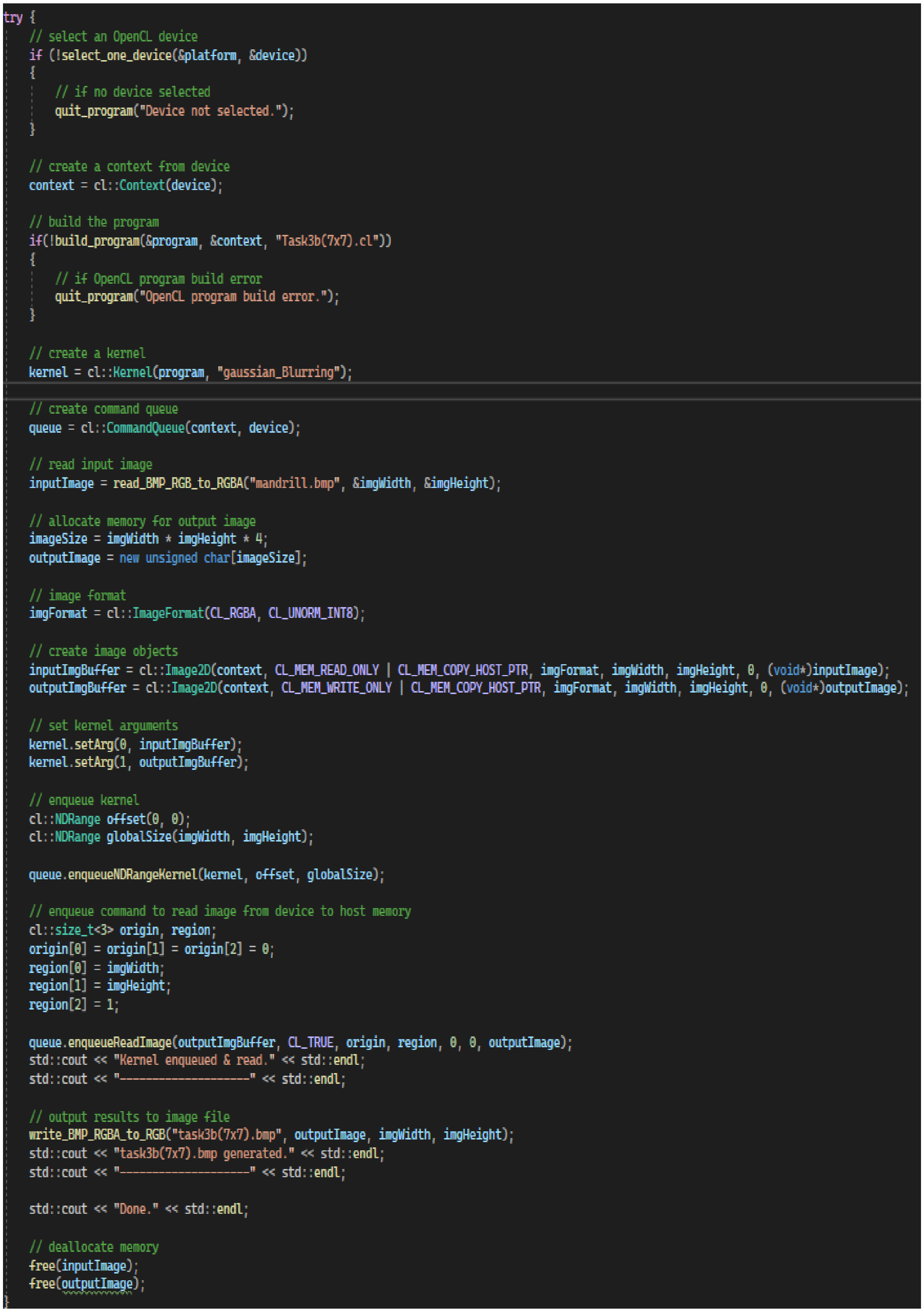
- Defines a constant sampler object named "sampler" with the following properties:
  CLK_NORMALIZED_COORDS_FALSE: Specifies that the coordinates are not normalized.
  CLK_ADDRESS_CLAMP: Specifies that coordinates outside the image boundaries will be clamped to the nearest valid address.
  CLK_FILTER_NEAREST: Specifies the use of nearest-neighbor filtering when reading image pixels.
- Defines a kernel named "luminance" that takes a read-only input image and a write-only output image as arguments.
- Gets the pixel coordinate of the current work-item using get_global_id function and assigns it to the coord variable.
- Reads the pixel value from the input image at the given coordinate using read_imagef function, passing the input image, sampler, and coordinate.
- Calculates the luminance value using the formula: luminance = 0.299*pixel.x + 0.587*pixel.y + 0.114*pixel.z.
- Writes the calculated luminance value to the output image at the same coordinate using write_imagef function, passing the output image, coordinate, and luminance value.

Task3b(7X7).cpp



```
cl::Platform platform;          // device's platform
cl::Device device;              // device used
cl::Context context;            // context for the device
cl::Program program;            // OpenCL program object
cl::Kernel kernel;              // a single kernel object
cl::CommandQueue queue;         // commandqueue for a context and device

// declare data and memory objects
unsigned char* inputImage;
unsigned char* outputImage;
int imgWidth, imgHeight, imageSize;

cl::ImageFormat imgFormat;
cl::Image2D inputImgBuffer, outputImgBuffer;
```

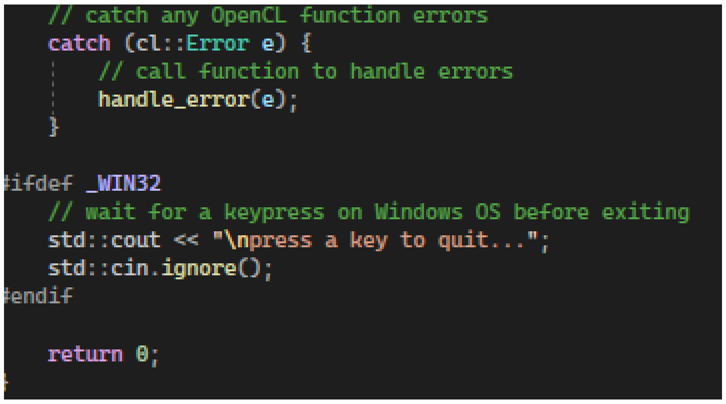
Declares several data structures and variables of OpenCL classes, such as Platform, Device, Context, Program, Kernel, and CommandQueue
Declares memory objects
Declares Image format , declares Image2D buffers



```cpp
try {
    // select an OpenCL device
    if (!select_one_device(&platform, &device))
    {
        // if no device selected
        quit_program("Device not selected.");
    }

    // create a context from device
    context = cl::Context(device);

    // build the program
    if(!build_program(&program, &context, "Task3b(7x7).cl"))
    {
        // if OpenCL program build error
        quit_program("OpenCL program build error.");
    }

    // create a kernel
    kernel = cl::Kernel(program, "gaussian_Blurring");

    // create command queue
    queue = cl::CommandQueue(context, device);

    // read input image
    inputImage = read_BMP_RGB_to_RGBA("mandrill.bmp", &imgWidth, &imgHeight);

    // allocate memory for output image
    imageSize = imgWidth * imgHeight * 4;
    outputImage = new unsigned char[imageSize];

    // image format
    imgFormat = cl::ImageFormat(CL_RGBA, CL_UNORM_INT8);

    // create image objects
    inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)inputImage);
    outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputImage);

    // set kernel arguments
    kernel.setArg(0, inputImgBuffer);
    kernel.setArg(1, outputImgBuffer);

    // enqueue kernel
    cl::NDRange offset(0, 0);
    cl::NDRange globalSize(imgWidth, imgHeight);

    queue.enqueueNDRangeKernel(kernel, offset, globalSize);

    // enqueue command to read image from device to host memory
    cl::size_t<3> origin, region;
    origin[0] = origin[1] = origin[2] = 0;
    region[0] = imgWidth;
    region[1] = imgHeight;
    region[2] = 1;

    queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputImage);
    std::cout << "Kernel enqueued & read." << std::endl;
    std::cout << "--------------------" << std::endl;

    // output results to image file
    write_BMP_RGBA_to_RGB("task3b(7x7).bmp", outputImage, imgWidth, imgHeight);
    std::cout << "task3b(7x7).bmp generated." << std::endl;
    std::cout << "--------------------" << std::endl;

    std::cout << "Done." << std::endl;

    // deallocate memory
    free(inputImage);
    free(outputImage);
}
```

- The code attempts to select an OpenCL device by calling the select_one_device function. If no device is selected, the program quits.
- A context is created using the selected device.
- The OpenCL program is built using the build_program function, which takes care of compiling the program code. If there is an error during program build, the program quits.
- A kernel named "gaussian_Blurring" is created from the program.
- A command queue is created using the context and device.
- The input image is read from a BMP file using the read_BMP_RGB_to_RGBA function. The image width, height, and size are obtained.
- Memory is allocated for the output image.
- An image format object is created for RGBA color format.
- Image buffers (inputImgBuffer and outputImgBuffer) are created using the image format and dimensions. The input image data is copied to inputImgBuffer.
- The kernel arguments are set, with inputImgBuffer assigned to argument 0 and outputImgBuffer assigned to argument 1.
- The kernel is enqueued for execution using enqueueNDRangeKernel with the specified offset and global size.
- The output data is read from outputImgBuffer to outputImage using enqueueReadImage.
- The resulting outputImage data is written to a BMP image file named "task3b(7x7).bmp" using the write_BMP_RGBA_to_RGB function.
- Status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.
- A final message is printed to the console indicating the completion of the entire image processing operation.
- Memory for the input and output images is deallocated

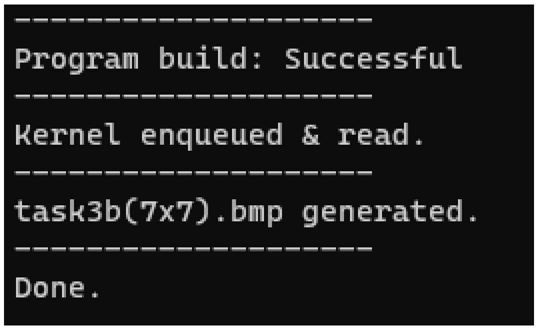
Error handling and quitting program



```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

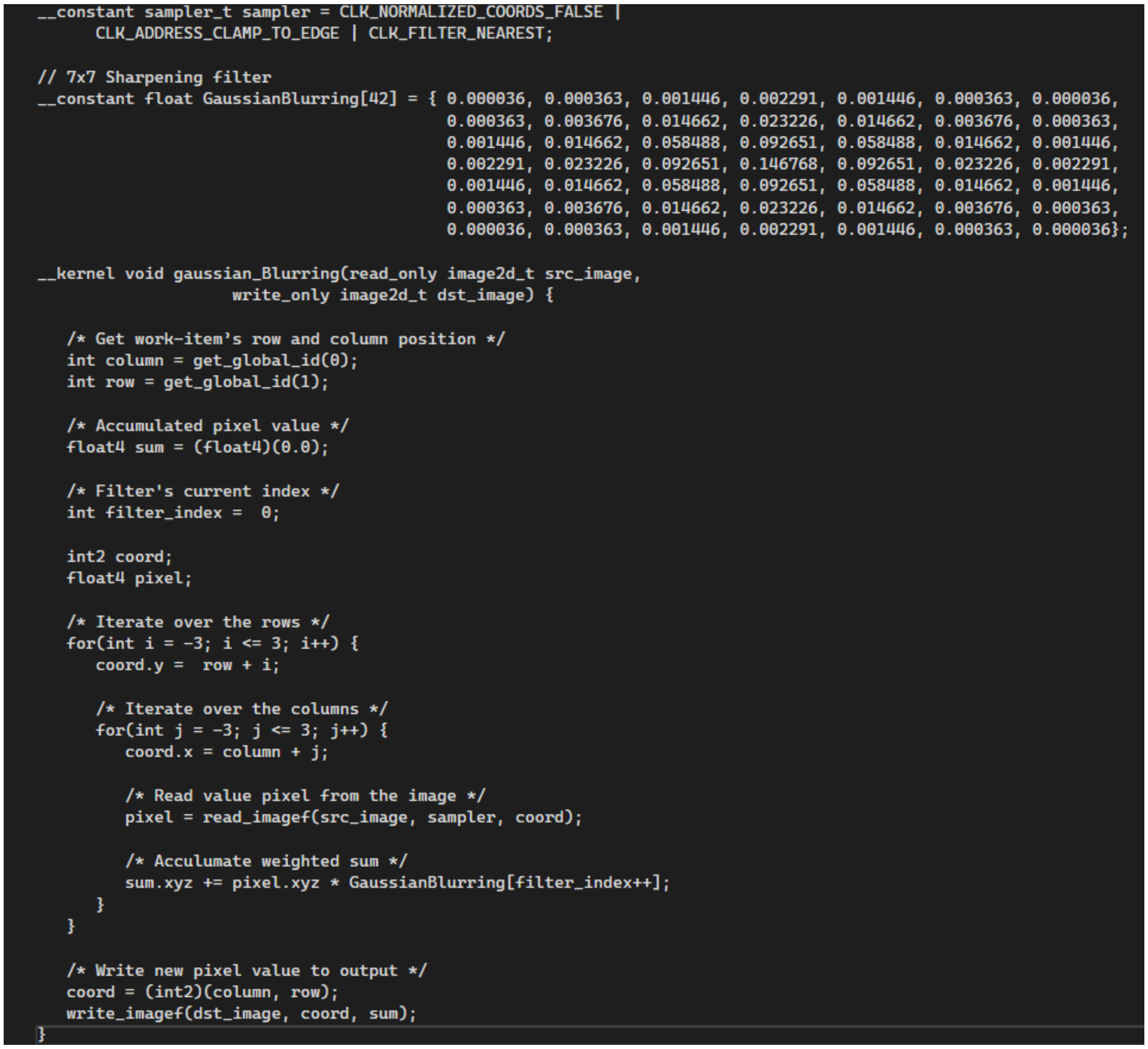
If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

Task 3c Program screenshots



```
---------------------
Program build: Successful
---------------------
Kernel enqueued & read.
---------------------
task3b(7x7).bmp generated.
---------------------
Done.
```

Task3b(7X7).cl

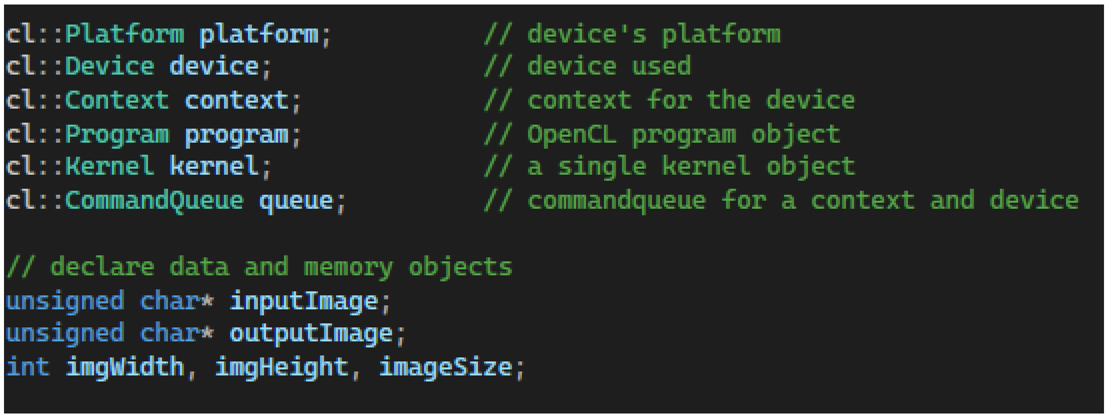


```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

// 7x7 Sharpening filter
__constant float GaussianBlurring[42] = { 0.000036, 0.000363, 0.001446, 0.002291, 0.001446, 0.000363, 0.000036,
                                          0.000363, 0.003676, 0.014662, 0.023226, 0.014662, 0.003676, 0.000363,
                                          0.001446, 0.014662, 0.058488, 0.092651, 0.058488, 0.014662, 0.001446,
                                          0.002291, 0.023226, 0.092651, 0.146768, 0.092651, 0.023226, 0.002291,
                                          0.001446, 0.014662, 0.058488, 0.092651, 0.058488, 0.014662, 0.001446,
                                          0.000363, 0.003676, 0.014662, 0.023226, 0.014662, 0.003676, 0.000363,
                                          0.000036, 0.000363, 0.001446, 0.002291, 0.001446, 0.000363, 0.000036};

__kernel void gaussian_Blurring(read_only image2d_t src_image,
                    write_only image2d_t dst_image) {

   /* Get work-item’s row and column position */
   int column = get_global_id(0);
   int row = get_global_id(1);

   /* Accumulated pixel value */
   float4 sum = (float4)(0.0);

   /* Filter's current index */
   int filter_index =  0;

   int2 coord;
   float4 pixel;

   /* Iterate over the rows */
   for(int i = -3; i <= 3; i++) {
      coord.y =  row + i;

      /* Iterate over the columns */
      for(int j = -3; j <= 3; j++) {
         coord.x = column + j;

         /* Read value pixel from the image */
         pixel = read_imagef(src_image, sampler, coord);

         /* Acculumate weighted sum */
         sum.xyz += pixel.xyz * GaussianBlurring[filter_index++];
      }
   }

   /* Write new pixel value to output */
   coord = (int2)(column, row);
   write_imagef(dst_image, coord, sum);
}
```

- The code defines a constant sampler object named "sampler" with specific properties using the CLK_NORMALIZED_COORDS_FALSE, CLK_ADDRESS_CLAMP_TO_EDGE, and CLK_FILTER_NEAREST flags.
- An array named "GaussianBlurring" is defined as a constant with 42 floating-point values representing a 7x7 sharpening filter.
- The kernel function "gaussian_Blurring" takes two image2d_t arguments: "src_image" (read-only) and "dst_image" (write-only). These arguments represent the source and destination images, respectively.
- The work-item's row and column positions are obtained using the get_global_id function.
- A float4 variable named "sum" is initialized to (0.0, 0.0, 0.0, 0.0) to accumulate the pixel values.

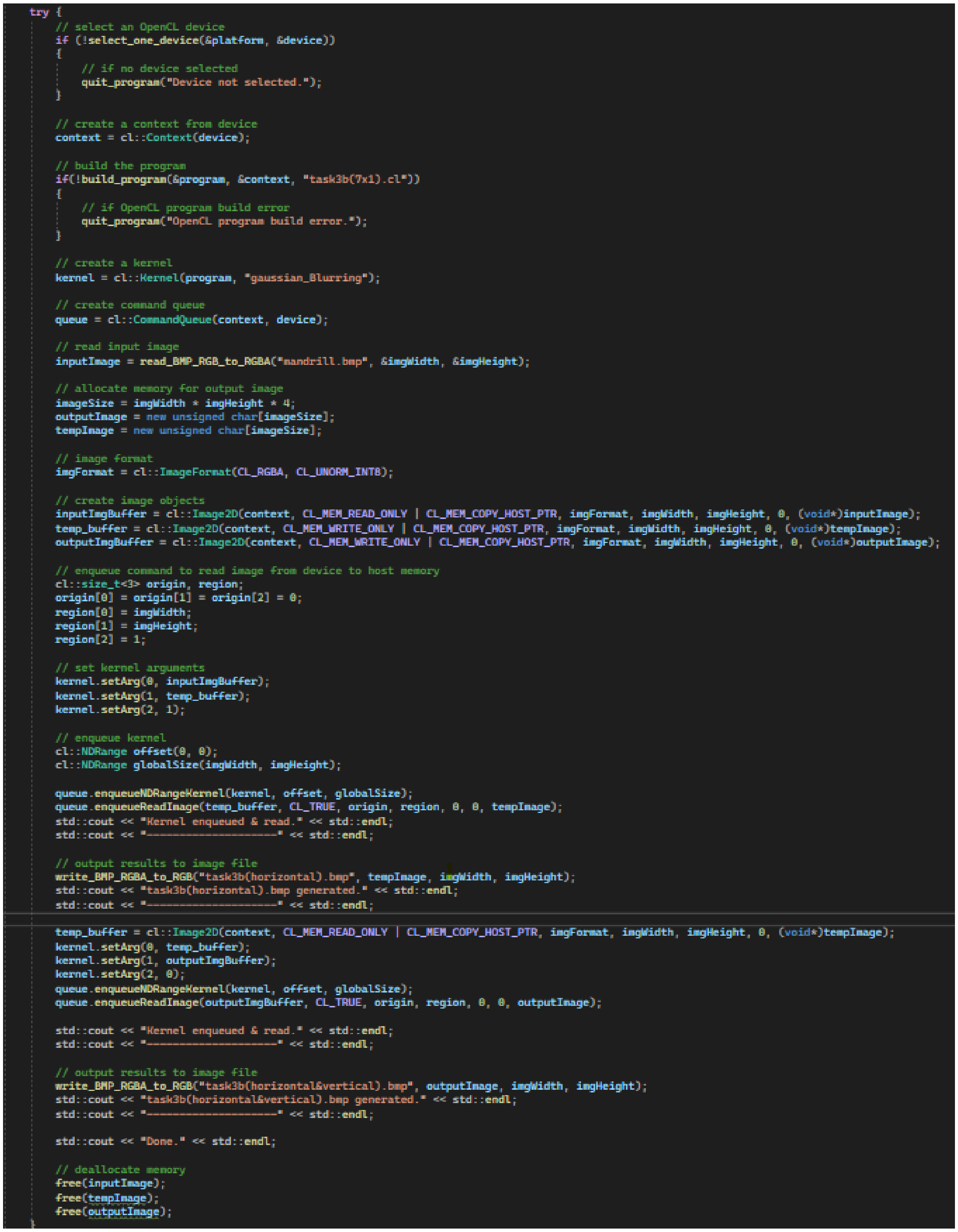
- A filter_index variable is declared and initialized to 0 to keep track of the current index in the GaussianBlurring array.
- Inside nested for loops, the kernel iterates over a 7x7 neighborhood centered at each pixel location.
- For each pixel in the neighborhood, the pixel value is read from the source image using read_imagef function, passing the sampler and coordinate.
- The pixel value is then multiplied by the corresponding weight from the GaussianBlurring array, and the result is accumulated in the "sum" variable.
- After processing all the pixels in the neighborhood, the new pixel value is written to the destination image using write_imagef function.

Task3b(7X1).cpp



```
cl::Platform platform;          // device's platform
cl::Device device;              // device used
cl::Context context;            // context for the device
cl::Program program;            // OpenCL program object
cl::Kernel kernel;              // a single kernel object
cl::CommandQueue queue;         // commandqueue for a context and device

// declare data and memory objects
unsigned char* inputImage;
unsigned char* outputImage;
int imgWidth, imgHeight, imageSize;
```

Declares several data structures and variables of OpenCL classes, such as Platform, Device, Context, Program, Kernel, and CommandQueue
Declares memory objects
Declares Image format , declares Image2D buffers

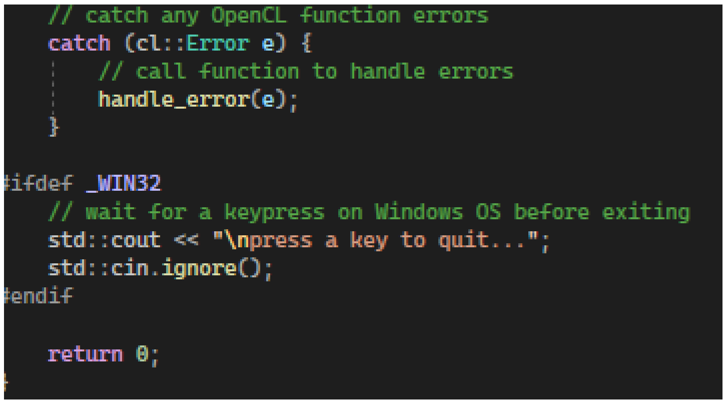


```
try {
    // select an OpenCL device
    if (!select_one_device(&platform, &device))
    {
        // if no device selected
        quit_program("Device not selected.");
    }

    // create a context from device
    context = cl::Context(device);

    // build the program
    if(!build_program(&program, &context, "task3b(7x1).cl"))
    {
        // if OpenCL program build error
        quit_program("OpenCL program build error.");
    }

    // create a kernel
    kernel = cl::Kernel(program, "gaussian_Blurring");

    // create command queue
    queue = cl::CommandQueue(context, device);

    // read input image
    inputImage = read_BMP_RGB_to_RGBA("mandrill.bmp", &imgWidth, &imgHeight);

    // allocate memory for output image
    imageSize = imgWidth * imgHeight * 4;
    outputImage = new unsigned char[imageSize];
    tempImage = new unsigned char[imageSize];

    // image format
    imgFormat = cl::ImageFormat(CL_RGBA, CL_UNORM_INT8);

    // create image objects
    inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)inputImage);
    temp_buffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)tempImage);
    outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputImage);

    // enqueue command to read image from device to host memory
    cl::size_t<3> origin, region;
    origin[0] = origin[1] = origin[2] = 0;
    region[0] = imgWidth;
    region[1] = imgHeight;
    region[2] = 1;

    // set kernel arguments
    kernel.setArg(0, inputImgBuffer);
    kernel.setArg(1, temp_buffer);
    kernel.setArg(2, 1);

    // enqueue kernel
    cl::NDRange offset(0, 0);
    cl::NDRange globalSize(imgWidth, imgHeight);

    queue.enqueueNDRangeKernel(kernel, offset, globalSize);
    queue.enqueueReadImage(temp_buffer, CL_TRUE, origin, region, 0, 0, tempImage);
    std::cout << "Kernel enqueued & read." << std::endl;
    std::cout << "--------------------" << std::endl;

    // output results to image file
    write_BMP_RGBA_to_RGB("task3b(horizontal).bmp", tempImage, imgWidth, imgHeight);
    std::cout << "task3b(horizontal).bmp generated." << std::endl;
    std::cout << "--------------------" << std::endl;

    temp_buffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)tempImage);
    kernel.setArg(0, temp_buffer);
    kernel.setArg(1, outputImgBuffer);
    kernel.setArg(2, 0);
    queue.enqueueNDRangeKernel(kernel, offset, globalSize);
    queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputImage);

    std::cout << "Kernel enqueued & read." << std::endl;
    std::cout << "--------------------" << std::endl;

    // output results to image file
    write_BMP_RGBA_to_RGB("task3b(horizontal&vertical).bmp", outputImage, imgWidth, imgHeight);
    std::cout << "task3b(horizontal&vertical).bmp generated." << std::endl;
    std::cout << "--------------------" << std::endl;

    std::cout << "Done." << std::endl;

    // deallocate memory
    free(inputImage);
    free(tempImage);
    free(outputImage);
}
```

- The code selects an OpenCL device by calling the select_one_device function. If no device is selected, the program quits.

- A context is created using the selected device.
- The OpenCL program is built using the build_program function, which takes care of compiling the program code. If there is an error during program build, the program quits.
- A kernel named "gaussian_Blurring" is created from the program.
- A command queue is created using the context and device.
- The input image is read from a BMP file using the read_BMP_RGB_to_RGBA function. The image width, height, and size are obtained.
- Memory is allocated for output images.
- An image format object is created for RGBA color format.
- Image buffers (inputImgBuffer, temp_buffer, outputImgBuffer) are created using the image format and dimensions. The input image data is copied to inputImgBuffer.
- The kernel arguments are set, with inputImgBuffer assigned to argument 0, temp_buffer assigned to argument 1, and the value 1 assigned to argument 2.
- The kernel is enqueued for execution using enqueueNDRangeKernel with the specified offset and global size.
- The output data is read from temp_buffer to tempImage using enqueueReadImage.
- The resulting tempImage data is written to a BMP image file named "task3b(horizontal).bmp" using the write_BMP_RGBA_to_RGB function.
- The temp_buffer is reassigned as a read-only image buffer using the CL_MEM_READ_ONLY flag and initialized with tempImage data.
- The kernel arguments are updated, with temp_buffer assigned to argument 0, outputImgBuffer assigned to argument 1, and the value 0 assigned to argument 2.
- The kernel is enqueued again for execution, and the output data is read from outputImgBuffer to outputImage.
- The resulting outputImage data is written to a BMP image file named "task3b(horizontal&vertical).bmp".
- A "Done" message is printed to the console.
- Memory for input, temporary, and output images is deallocated.

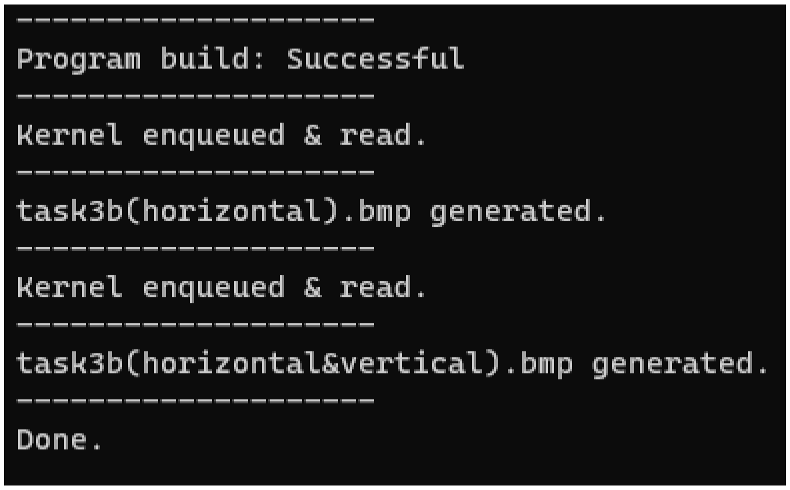
Error handling and quitting program



```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

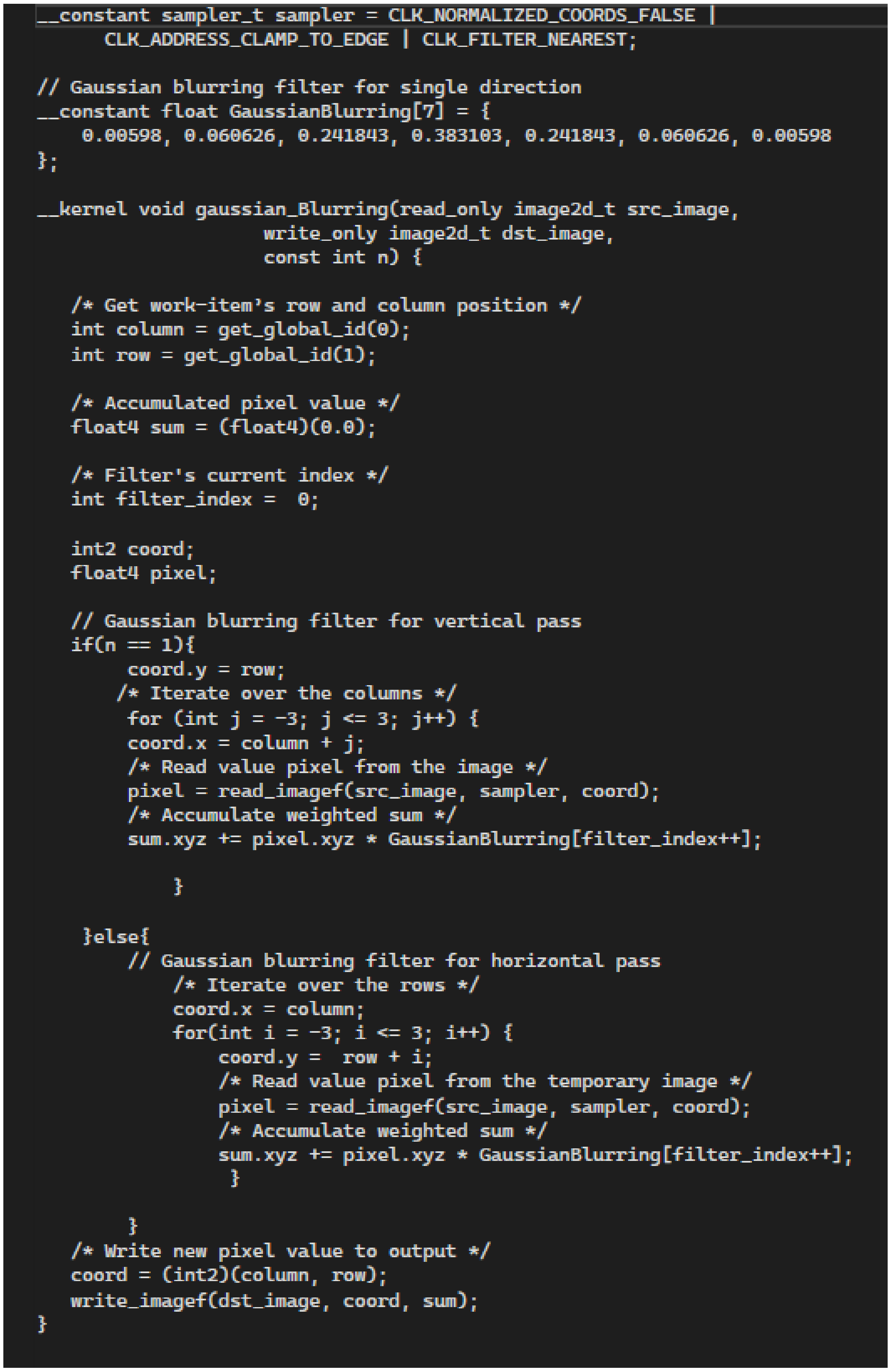
If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

Task 3b(7X1) Program screenshots



```
--------------------
Program build: Successful
--------------------
Kernel enqueued & read.
--------------------
task3b(horizontal).bmp generated.
--------------------
Kernel enqueued & read.
--------------------
task3b(horizontal&vertical).bmp generated.
--------------------
Done.
```

Task3b(7X1).cl

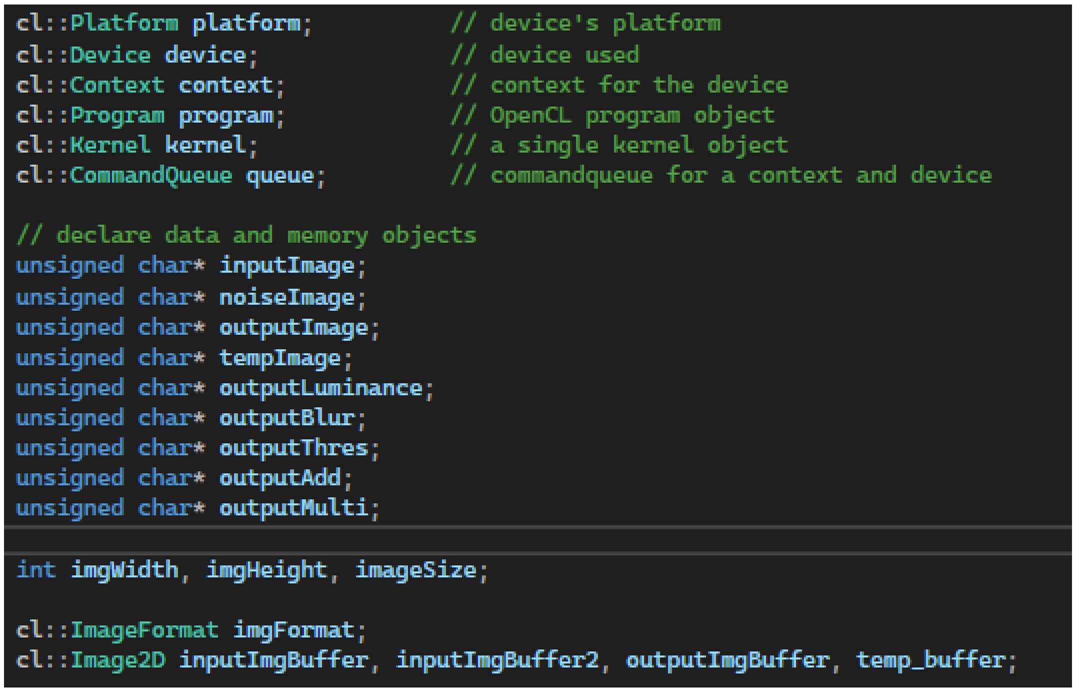


```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP_TO_EDGE | CLK_FILTER_NEAREST;

// Gaussian blurring filter for single direction
__constant float GaussianBlurring[7] = {
    0.00598, 0.060626, 0.241843, 0.383103, 0.241843, 0.060626, 0.00598
};

__kernel void gaussian_Blurring(read_only image2d_t src_image,
                    write_only image2d_t dst_image,
                    const int n) {

   /* Get work-item's row and column position */
   int column = get_global_id(0);
   int row = get_global_id(1);

   /* Accumulated pixel value */
   float4 sum = (float4)(0.0);

   /* Filter's current index */
   int filter_index =  0;

   int2 coord;
   float4 pixel;

   // Gaussian blurring filter for vertical pass
   if(n == 1){
        coord.y = row;
       /* Iterate over the columns */
        for (int j = -3; j <= 3; j++) {
        coord.x = column + j;
        /* Read value pixel from the image */
        pixel = read_imagef(src_image, sampler, coord);
        /* Accumulate weighted sum */
        sum.xyz += pixel.xyz * GaussianBlurring[filter_index++];

            }

    }else{
        // Gaussian blurring filter for horizontal pass
            /* Iterate over the rows */
            coord.x = column;
            for(int i = -3; i <= 3; i++) {
                coord.y =  row + i;
                /* Read value pixel from the temporary image */
                pixel = read_imagef(src_image, sampler, coord);
                /* Accumulate weighted sum */
                sum.xyz += pixel.xyz * GaussianBlurring[filter_index++];
                 }

        }
   /* Write new pixel value to output */
   coord = (int2)(column, row);
   write_imagef(dst_image, coord, sum);
}
```

- The array called GaussianBlurring, which represents the values of a Gaussian blur filter in a single direction
- The kernel function takes three arguments: a read-only input image (src_image), a write-only output image (dst_image), and an integer value (n) that determines the direction of the blur pass.
- The function retrieves the row and column positions of the current work-item using get_global_id(0) and get_global_id(1).
- An initial sum variable is initialized to (0.0) to accumulate the pixel values.
- Depending on the value of n, the function executes either the vertical or horizontal pass of the Gaussian blur.
- If n is equal to 1 (indicating the vertical pass), the function iterates over the columns with an offset of -3 to +3. It reads the pixel values from the input image at the corresponding coordinates, multiplies them with the corresponding Gaussian blur weight from the GaussianBlurring array, and accumulates the weighted sum in the sum variable.
- If n is not equal to 1 (indicating the horizontal pass), the function iterates over the rows with an offset of -3 to +3. It reads the pixel values from the input image, applies the same operations as in the vertical pass, and accumulates the weighted sum in the sum variable.
- Finally, the modified pixel value (the accumulated sum) is written to the output image at the original coordinates using the write_imagef function.

Task3c.cpp

declaring a number of data structures and variables



```
cl::Platform platform;          // device's platform
cl::Device device;              // device used
cl::Context context;            // context for the device
cl::Program program;            // OpenCL program object
cl::Kernel kernel;              // a single kernel object
cl::CommandQueue queue;         // commandqueue for a context and device

// declare data and memory objects
unsigned char* inputImage;
unsigned char* noiseImage;
unsigned char* outputImage;
unsigned char* tempImage;
unsigned char* outputLuminance;
unsigned char* outputBlur;
unsigned char* outputThres;
unsigned char* outputAdd;
unsigned char* outputMulti;

int imgWidth, imgHeight, imageSize;

cl::ImageFormat imgFormat;
cl::Image2D inputImgBuffer, inputImgBuffer2, outputImgBuffer, temp_buffer;
```

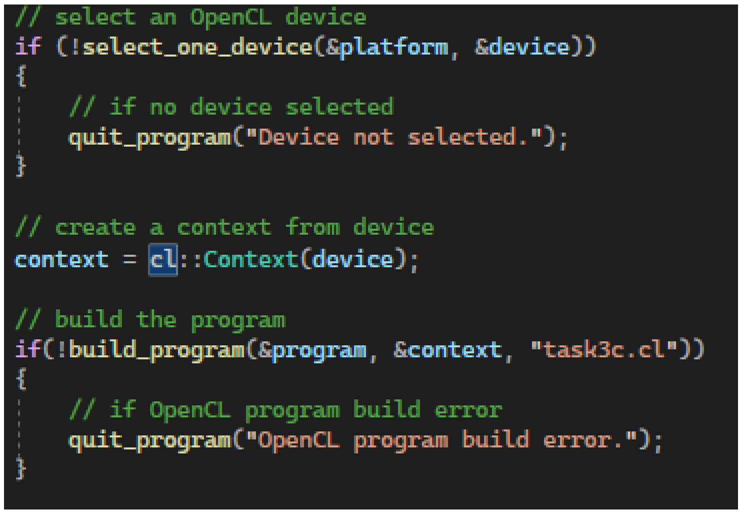
Declares several data structures and variables of OpenCL classes, such as Platform, Device, Context, Program, Kernel, and CommandQueue

Declares memory objects

Declares Image format

Declares Image2D buffers

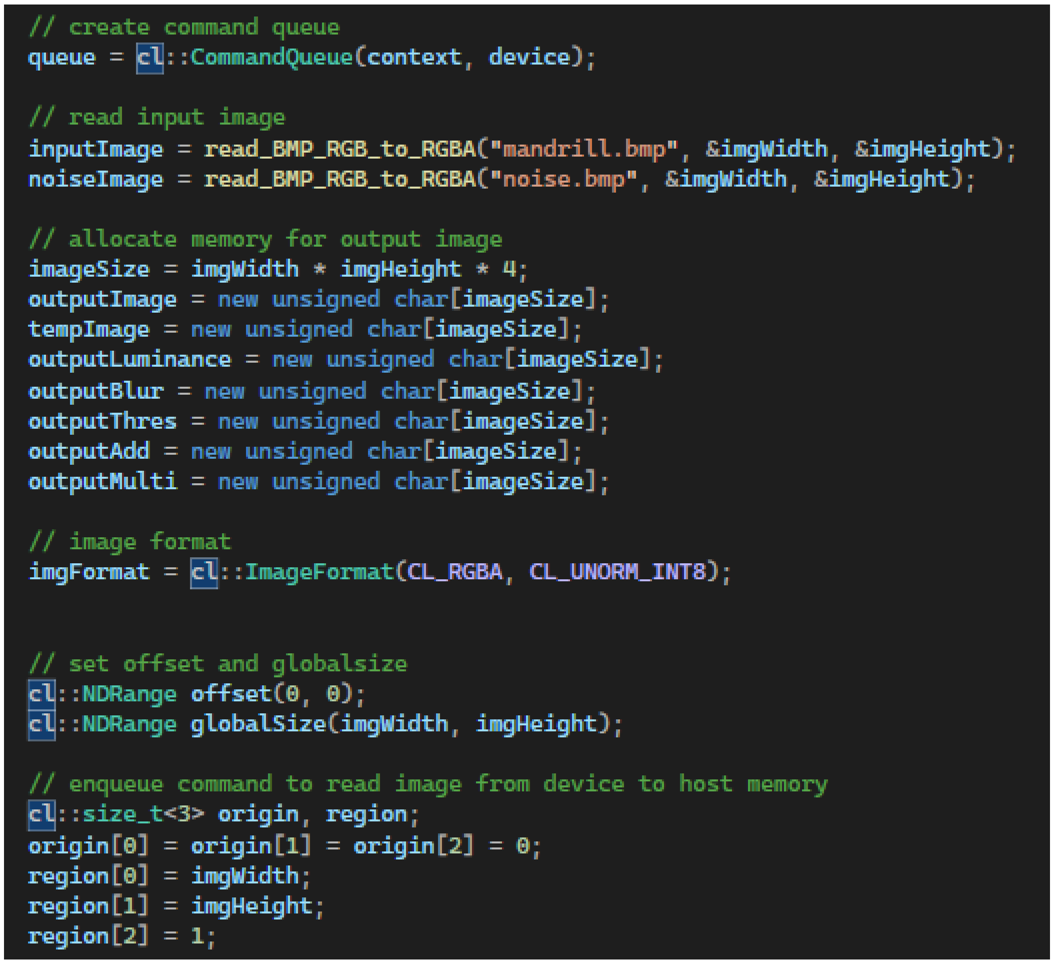
Select Device , Creating context and build program



```
// select an OpenCL device
if (!select_one_device(&platform, &device))
{
    // if no device selected
    quit_program("Device not selected.");
}

// create a context from device
context = cl::Context(device);

// build the program
if(!build_program(&program, &context, "task3c.cl"))
{
    // if OpenCL program build error
    quit_program("OpenCL program build error.");
}
```

This section of code attempts to select an OpenCL device. If no device is selected, the program will terminate with an error message. The context is constructed then the program is built using the build_program function

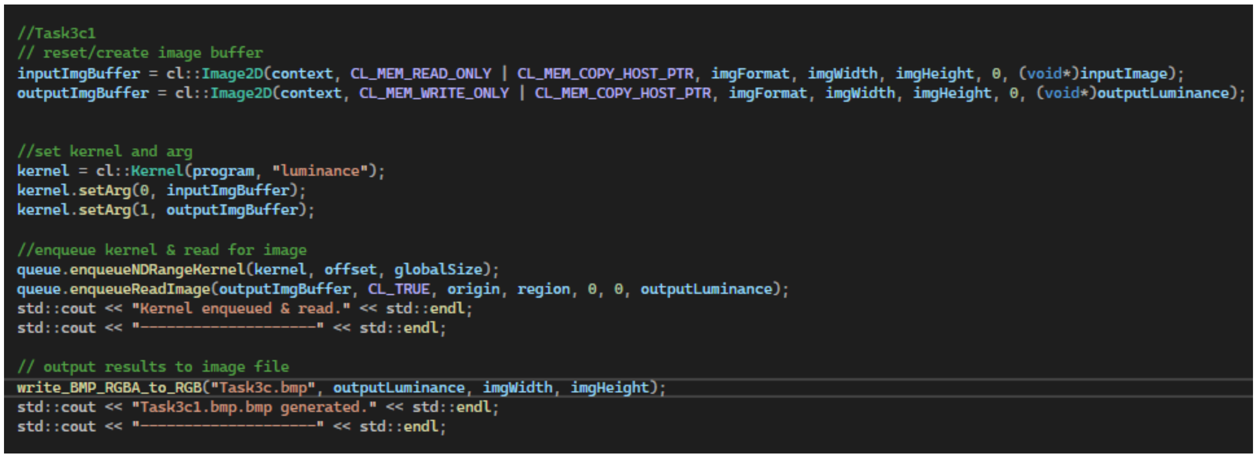
Command Queue, Read input, Allocate memory, offset,global size



```
// create command queue
queue = cl::CommandQueue(context, device);

// read input image
inputImage = read_BMP_RGB_to_RGBA("mandrill.bmp", &imgWidth, &imgHeight);
noiseImage = read_BMP_RGB_to_RGBA("noise.bmp", &imgWidth, &imgHeight);

// allocate memory for output image
imageSize = imgWidth * imgHeight * 4;
outputImage = new unsigned char[imageSize];
tempImage = new unsigned char[imageSize];
outputLuminance = new unsigned char[imageSize];
outputBlur = new unsigned char[imageSize];
outputThres = new unsigned char[imageSize];
outputAdd = new unsigned char[imageSize];
outputMulti = new unsigned char[imageSize];

// image format
imgFormat = cl::ImageFormat(CL_RGBA, CL_UNORM_INT8);


// set offset and globalsize
cl::NDRange offset(0, 0);
cl::NDRange globalSize(imgWidth, imgHeight);

// enqueue command to read image from device to host memory
cl::size_t<3> origin, region;
origin[0] = origin[1] = origin[2] = 0;
region[0] = imgWidth;
region[1] = imgHeight;
region[2] = 1;
```

- The command queue is created using the cl::CommandQueue constructor with the context and device as arguments.
- The input images (mandrill.bmp and noise.bmp) are read and converted into RGBA format using the read_BMP_RGB_to_RGBA function.
- Memory is allocated for the output image (outputImage) and several intermediate image buffers (tempImage, outputLuminance, outputBlur, outputThres, outputAdd, outputMulti), based on the image size (imgWidth * imgHeight * 4) calculated earlier.
- An image format (imgFormat) is defined as RGBA with unsigned normalized integer (8 bits per channel).
- The offset is set to (0, 0) and the global size is set to match the dimensions of the image (imgWidth, imgHeight).
- Command enqueueing is performed to read the image from the device to host memory using the specified origin and region. The region represents the size of the image.

Task3c1



```
//Task3c1
// reset/create image buffer
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)inputImage);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputLuminance);


//set kernel and arg
kernel = cl::Kernel(program, "luminance");
kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, outputImgBuffer);

//enqueue kernel & read for image
queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputLuminance);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;

// output results to image file
write_BMP_RGBA_to_RGB("Task3c.bmp", outputLuminance, imgWidth, imgHeight);
std::cout << "Task3c1.bmp.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;
```

- The image buffers inputImgBuffer and outputImgBuffer are reset or created using the OpenCL context, image format, and dimensions. inputImgBuffer is set as read-only and initialized with the inputImage data, while outputImgBuffer is set as write-only and initialized with the outputLuminance data.

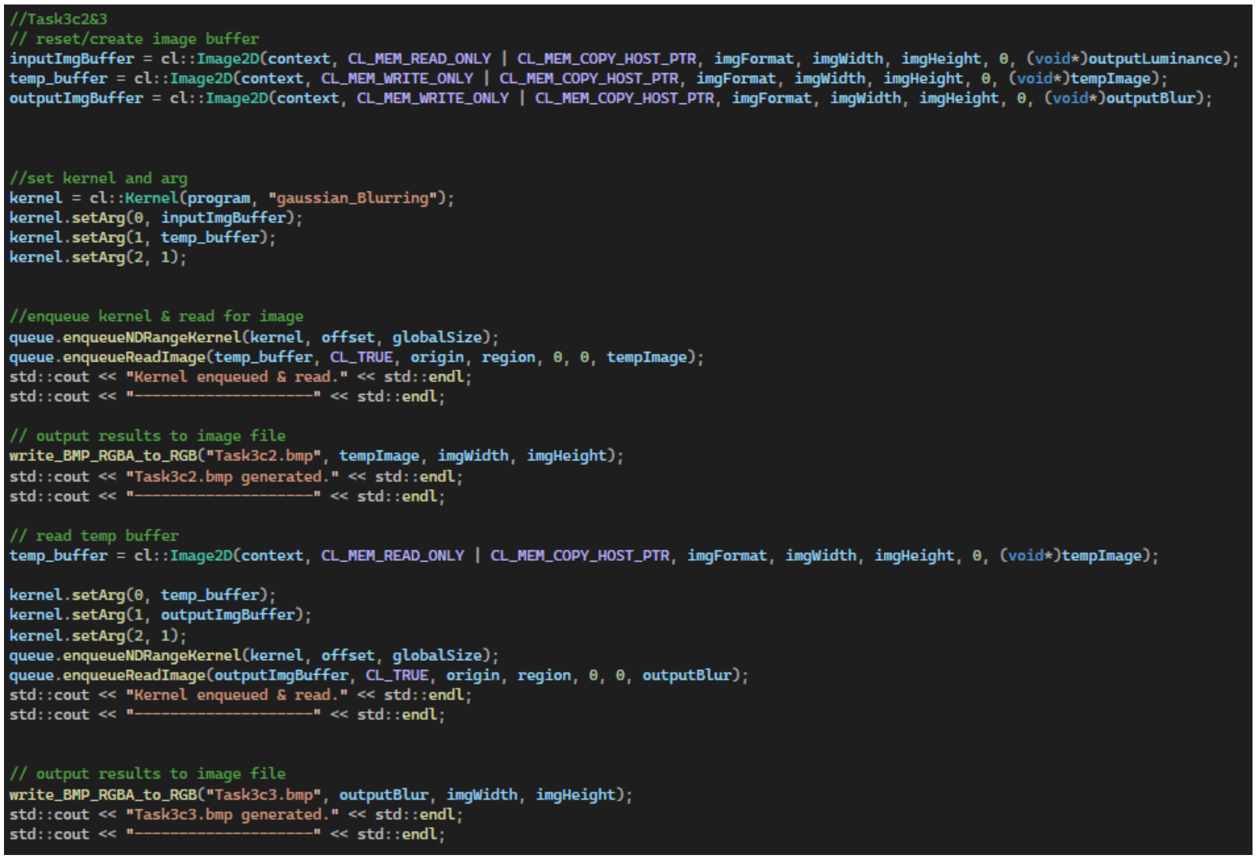
- The kernel object kernel is set to the "luminance" kernel within the program. The arguments for the kernel are then set, with inputImgBuffer assigned to argument 0 and outputImgBuffer assigned to argument 1.

- The kernel is enqueued for execution using enqueueNDRangeKernel, with the specified offset and global size.

- After the kernel execution, the output data is read from outputImgBuffer to outputLuminance using enqueueReadImage.

- The resulting outputLuminance data is then written to a BMP image file named "Task3c.bmp" using the write_BMP_RGBA_to_RGB function.
- Finally, status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.

Task3c2&3



```
//Task3c2&3
// reset/create image buffer
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputLuminance);
temp_buffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)tempImage);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputBlur);



//set kernel and arg
kernel = cl::Kernel(program, "gaussian_Blurring");
kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, temp_buffer);
kernel.setArg(2, 1);


//enqueue kernel & read for image
queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(temp_buffer, CL_TRUE, origin, region, 0, 0, tempImage);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;

// output results to image file
write_BMP_RGBA_to_RGB("Task3c2.bmp", tempImage, imgWidth, imgHeight);
std::cout << "Task3c2.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;

// read temp buffer
temp_buffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)tempImage);

kernel.setArg(0, temp_buffer);
kernel.setArg(1, outputImgBuffer);
kernel.setArg(2, 1);
queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputBlur);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;


// output results to image file
write_BMP_RGBA_to_RGB("Task3c3.bmp", outputBlur, imgWidth, imgHeight);
std::cout << "Task3c3.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;
```

- The image buffers inputImgBuffer and outputImgBuffer are reset or created using the OpenCL context, image format, and dimensions. inputImgBuffer is set as read-only and initialized with the inputImage data, while outputImgBuffer is set as write-only and initialized with the outputLuminance data.

- The kernel object kernel is set to the "luminance" kernel within the program. The arguments for the kernel are then set, with inputImgBuffer assigned to argument 0 and outputImgBuffer assigned to argument 1.

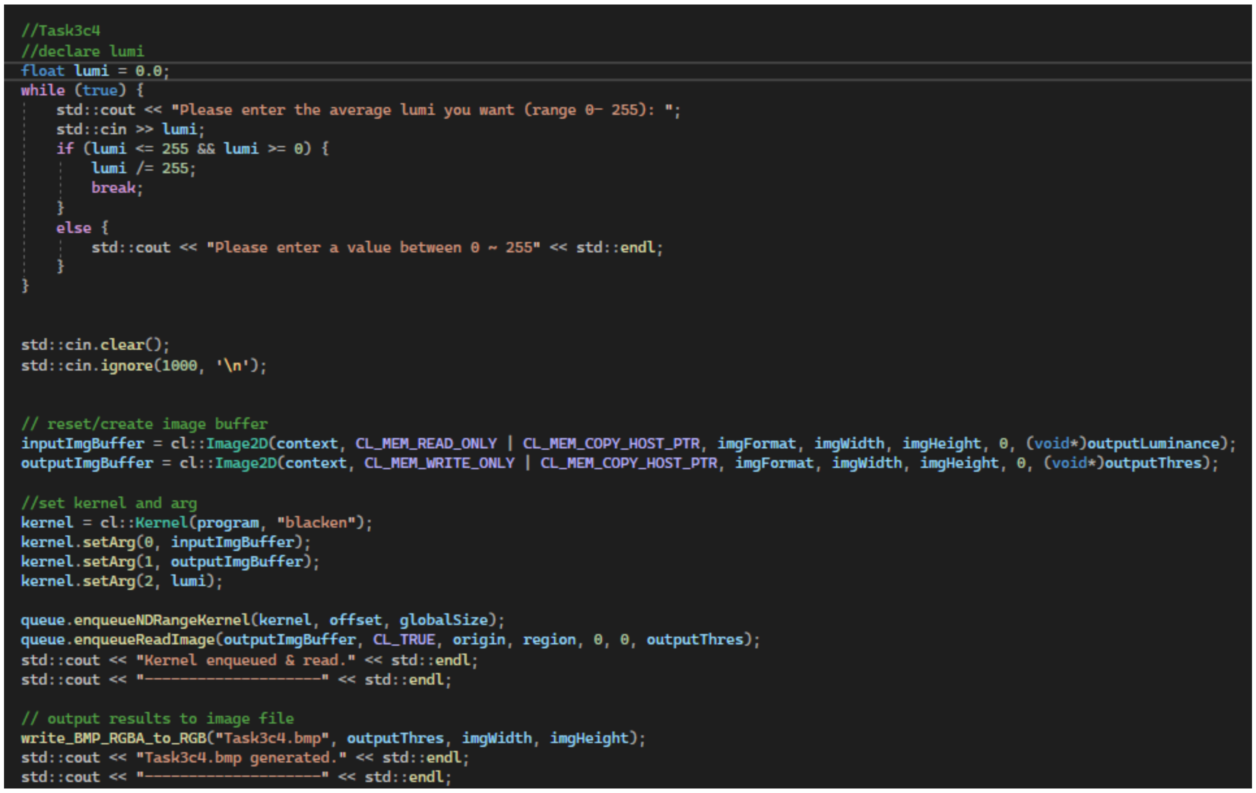
- The kernel is enqueued for execution using enqueueNDRangeKernel, with the specified offset and global size.

- After the kernel execution, the output data is read from outputImgBuffer to outputLuminance using enqueueReadImage.

- The resulting outputLuminance data is then written to a BMP image file named "Task3c.bmp" using the write_BMP_RGBA_to_RGB function.

- Finally, status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.

Task3c4



```cpp
//Task3c4
//declare lumi
float lumi = 0.0;
while (true) {
    std::cout << "Please enter the average lumi you want (range 0- 255): ";
    std::cin >> lumi;
    if (lumi <= 255 && lumi >= 0) {
        lumi /= 255;
        break;
    }
    else {
        std::cout << "Please enter a value between 0 ~ 255" << std::endl;
    }
}


std::cin.clear();
std::cin.ignore(1000, '\n');


// reset/create image buffer
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputLuminance);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputThres);

//set kernel and arg
kernel = cl::Kernel(program, "blacken");
kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, outputImgBuffer);
kernel.setArg(2, lumi);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputThres);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;

// output results to image file
write_BMP_RGBA_to_RGB("Task3c4.bmp", outputThres, imgWidth, imgHeight);
std::cout << "Task3c4.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;
```

- The code prompts the user to enter an average luminance value in the range of 0 to 255. The entered value is divided by 255 to normalize it between 0 and 1.

- The image buffers inputImgBuffer and outputImgBuffer are reset or created using the OpenCL context, image format, and dimensions. inputImgBuffer is set as read-only and initialized with the outputLuminance data. outputImgBuffer is set as write-only and initialized with the outputThres data.

- The kernel object kernel is set to the "blacken" kernel within the program. The arguments for the kernel are then set, with inputImgBuffer assigned to argument 0, outputImgBuffer assigned to argument 1, and the normalized luminance value assigned to argument 2.

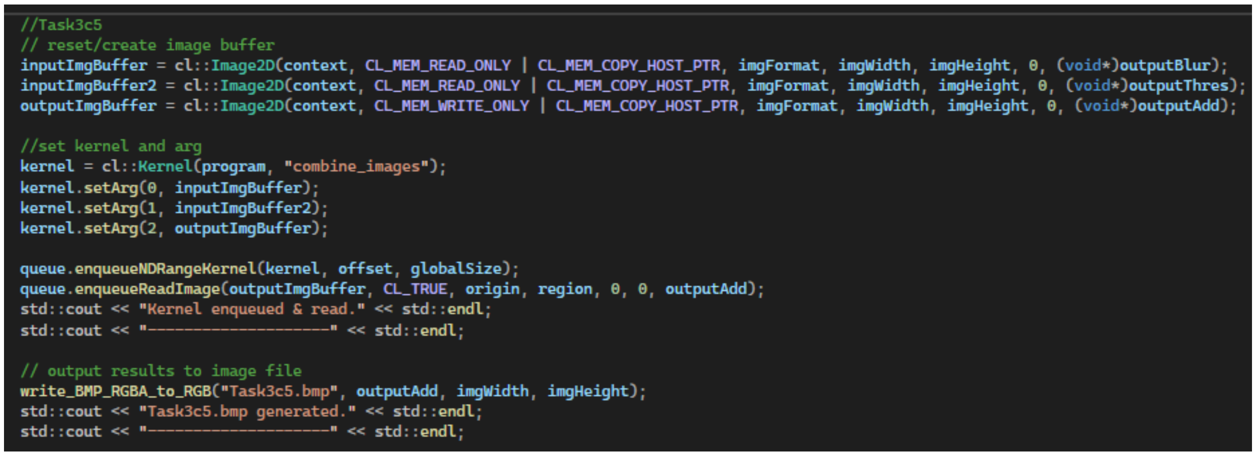
- The kernel is enqueued for execution using enqueueNDRangeKernel, with the specified offset and global size.

- After the kernel execution, the output data is read from outputImgBuffer to outputThres using enqueueReadImage.

- The resulting outputThres data is then written to a BMP image file named "Task3c4.bmp" using the write_BMP_RGBA_to_RGB function.

- Status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.

Task3c5



```
//Task3c5
// reset/create image buffer
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputBlur);
inputImgBuffer2 = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputThres);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputAdd);

//set kernel and arg
kernel = cl::Kernel(program, "combine_images");
kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, inputImgBuffer2);
kernel.setArg(2, outputImgBuffer);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputAdd);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;

// output results to image file
write_BMP_RGBA_to_RGB("Task3c5.bmp", outputAdd, imgWidth, imgHeight);
std::cout << "Task3c5.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;
```

- The image buffers inputImgBuffer, inputImgBuffer2, and outputImgBuffer are reset or created using the OpenCL context, image format, and dimensions. inputImgBuffer is set as read-only and initialized with the outputBlur data. inputImgBuffer2 is set as read-only and initialized with the outputThres data. outputImgBuffer is set as write-only and initialized with the outputAdd data.

- The kernel object kernel is set to the "combine_images" kernel within the program. The arguments for the kernel are then set, with inputImgBuffer assigned to argument 0, inputImgBuffer2 assigned to argument 1, and outputImgBuffer assigned to argument 2.

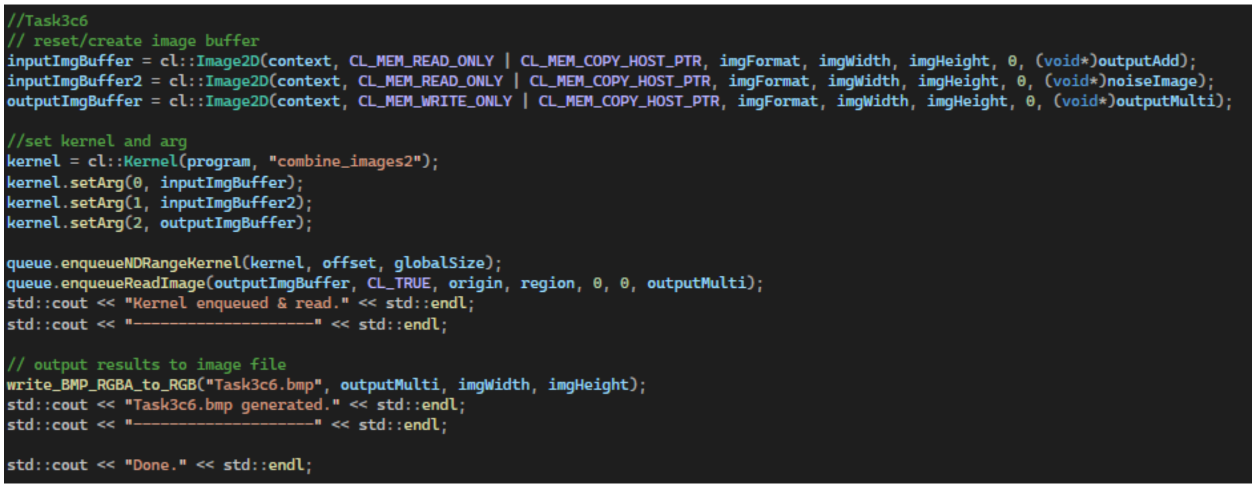
- The kernel is enqueued for execution using enqueueNDRangeKernel, with the specified offset and global size.

- After the kernel execution, the output data is read from outputImgBuffer to outputAdd using enqueueReadImage.

- The resulting outputAdd data is then written to a BMP image file named "Task3c5.bmp" using the write_BMP_RGBA_to_RGB function.

- Status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.

Task3c6

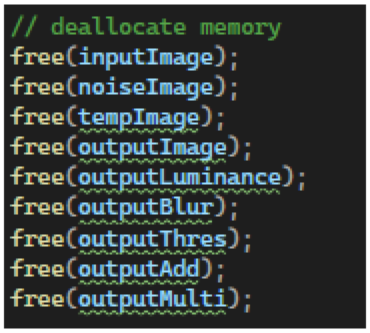


```
//Task3c6
// reset/create image buffer
inputImgBuffer = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputAdd);
inputImgBuffer2 = cl::Image2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)noiseImage);
outputImgBuffer = cl::Image2D(context, CL_MEM_WRITE_ONLY | CL_MEM_COPY_HOST_PTR, imgFormat, imgWidth, imgHeight, 0, (void*)outputMulti);

//set kernel and arg
kernel = cl::Kernel(program, "combine_images2");
kernel.setArg(0, inputImgBuffer);
kernel.setArg(1, inputImgBuffer2);
kernel.setArg(2, outputImgBuffer);

queue.enqueueNDRangeKernel(kernel, offset, globalSize);
queue.enqueueReadImage(outputImgBuffer, CL_TRUE, origin, region, 0, 0, outputMulti);
std::cout << "Kernel enqueued & read." << std::endl;
std::cout << "--------------------" << std::endl;

// output results to image file
write_BMP_RGBA_to_RGB("Task3c6.bmp", outputMulti, imgWidth, imgHeight);
std::cout << "Task3c6.bmp generated." << std::endl;
std::cout << "--------------------" << std::endl;

std::cout << "Done." << std::endl;
```

- The image buffers inputImgBuffer, inputImgBuffer2, and outputImgBuffer are reset or created using the OpenCL context, image format, and dimensions. inputImgBuffer is set as read-only and initialized with the outputAdd data. inputImgBuffer2 is set as read-only and initialized with the noiseImage data. outputImgBuffer is set as write-only and initialized with the outputMulti data.
-
- The kernel object kernel is set to the "combine_images2" kernel within the program. The arguments for the kernel are then set, with inputImgBuffer assigned to argument 0, inputImgBuffer2 assigned to argument 1, and outputImgBuffer assigned to argument 2.
-
- The kernel is enqueued for execution using enqueueNDRangeKernel, with the specified offset and global size.
-
- After the kernel execution, the output data is read from outputImgBuffer to outputMulti using enqueueReadImage.
-
- The resulting outputMulti data is then written to a BMP image file named "Task3c6.bmp" using the write_BMP_RGBA_to_RGB function.
-
- Status messages are printed to the console indicating the completion of the kernel execution and the generation of the output image file.
-

- A final message is printed to the console indicating the completion of the entire image processing operation.

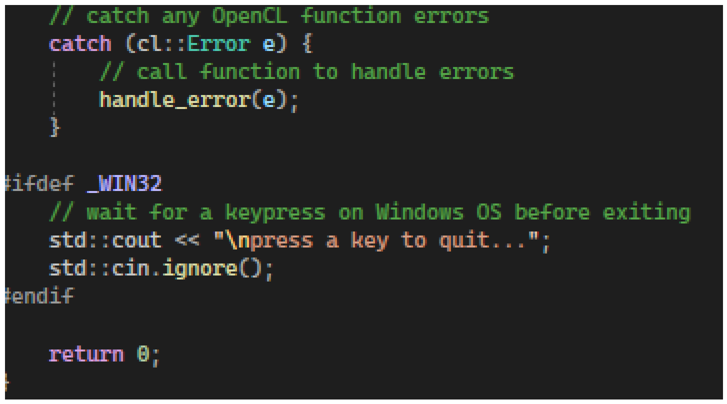
Deallocate Memory



```
// deallocate memory
free(inputImage);
free(noiseImage);
free(tempImage);
free(outputImage);
free(outputLuminance);
free(outputBlur);
free(outputThres);
free(outputAdd);
free(outputMulti);
```

Memory Deallocation:
The memory allocated for the output items are freed.

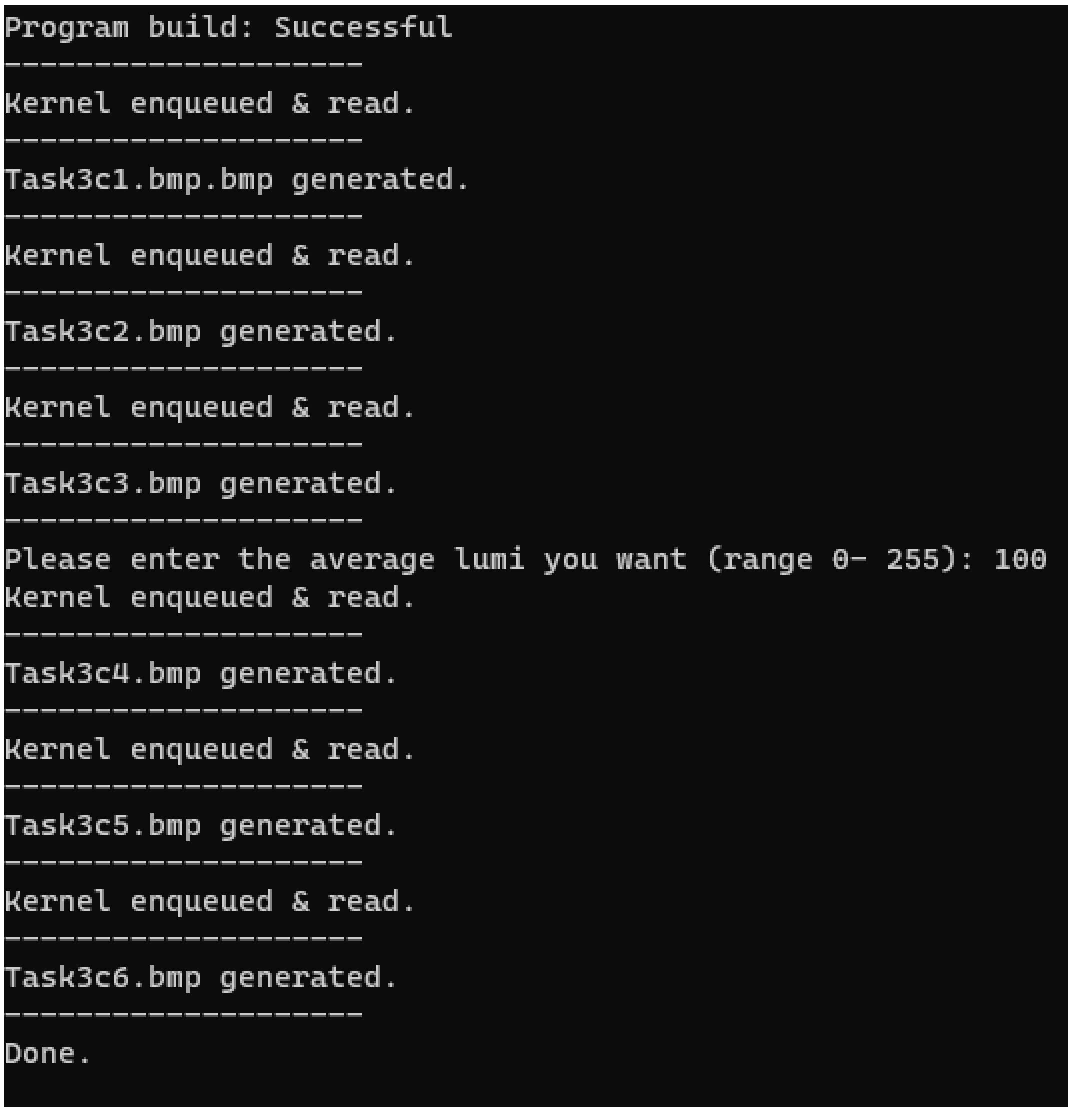
Error handling and quitting program



```
    // catch any OpenCL function errors
    catch (cl::Error e) {
        // call function to handle errors
        handle_error(e);
    }

#ifdef _WIN32
    // wait for a keypress on Windows OS before exiting
    std::cout << "\npress a key to quit...";
    std::cin.ignore();
#endif

    return 0;
}
```

If any OpenCL function errors occur, the handle_error function is called to handle them, followed by the keypress to exit program

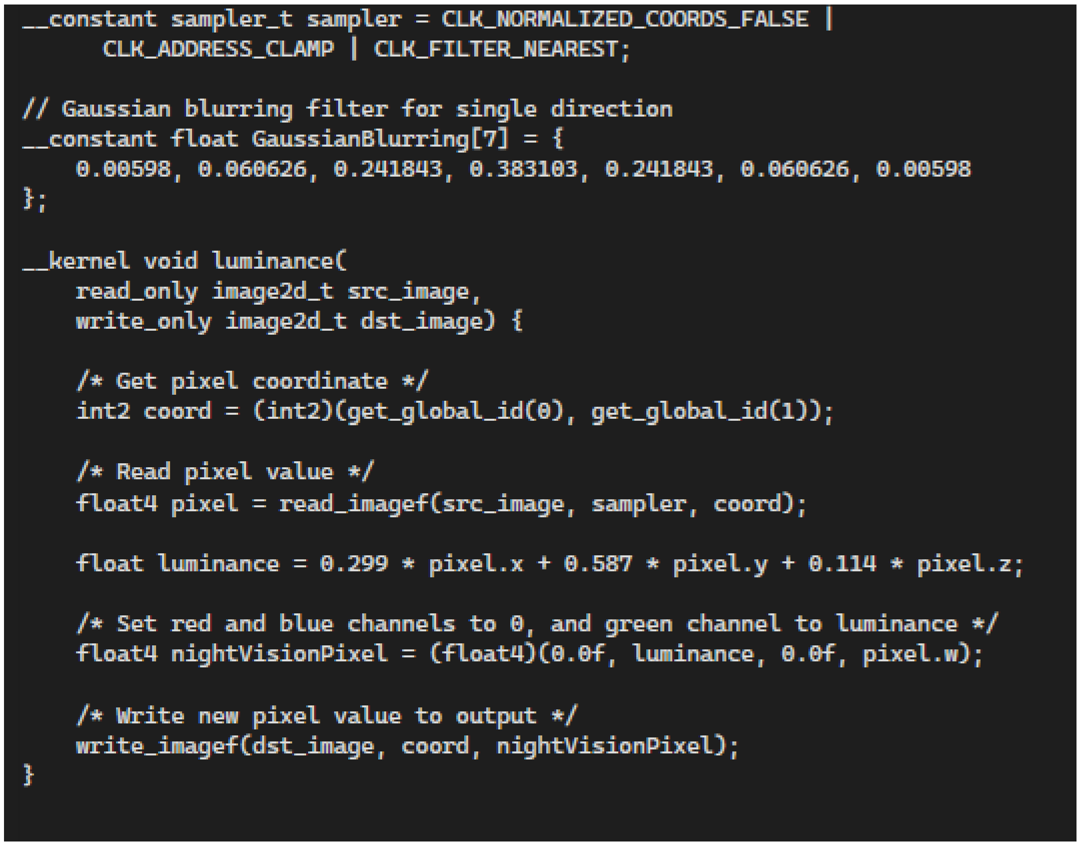
Task 3c Program screenshots



```
Program build: Successful
--------------------
Kernel enqueued & read.
--------------------
Task3c1.bmp.bmp generated.
--------------------
Kernel enqueued & read.
--------------------
Task3c2.bmp generated.
--------------------
Kernel enqueued & read.
--------------------
Task3c3.bmp generated.
--------------------
Please enter the average lumi you want (range 0- 255): 100
Kernel enqueued & read.
--------------------
Task3c4.bmp generated.
--------------------
Kernel enqueued & read.
--------------------
Task3c5.bmp generated.
--------------------
Kernel enqueued & read.
--------------------
Task3c6.bmp generated.
--------------------
Done.
```

Task3c.cl

Luminance

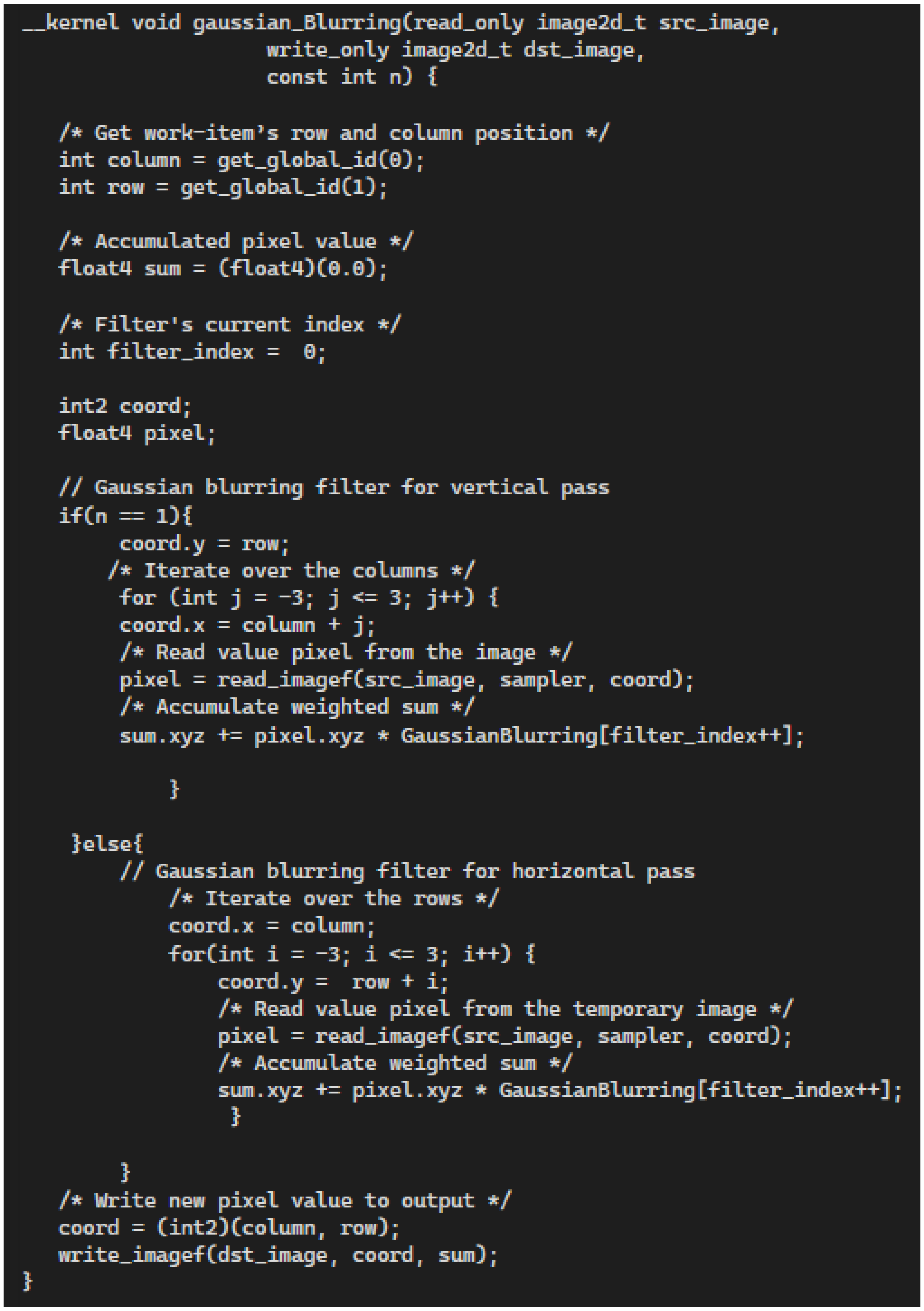


```
__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
      CLK_ADDRESS_CLAMP | CLK_FILTER_NEAREST;

// Gaussian blurring filter for single direction
__constant float GaussianBlurring[7] = {
    0.00598, 0.060626, 0.241843, 0.383103, 0.241843, 0.060626, 0.00598
};

__kernel void luminance(
    read_only image2d_t src_image,
    write_only image2d_t dst_image) {

    /* Get pixel coordinate */
    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    /* Read pixel value */
    float4 pixel = read_imagef(src_image, sampler, coord);

    float luminance = 0.299 * pixel.x + 0.587 * pixel.y + 0.114 * pixel.z;

    /* Set red and blue channels to 0, and green channel to luminance */
    float4 nightVisionPixel = (float4)(0.0f, luminance, 0.0f, pixel.w);

    /* Write new pixel value to output */
    write_imagef(dst_image, coord, nightVisionPixel);
}
```

- CLK_NORMALIZED_COORDS_FALSE: This indicates that the coordinates used for sampling are not normalized (i.e., they are in pixel coordinates).
- CLK_ADDRESS_CLAMP: This specifies that the sampler should clamp the coordinates to the edge of the image if they go beyond the valid range.
- CLK_FILTER_NEAREST: This indicates that the sampler should use the nearest neighbor filtering method to determine the pixel value at a non-integer coordinate.
-
- The array called GaussianBlurring, which represents the values of a Gaussian blur filter in a single direction
-
- The kernel takes two arguments: a read-only input image (src_image) and a write-only output image (dst_image).
- It retrieves the pixel coordinates for the current thread using get_global_id(0) and get_global_id(1).
- The pixel value at the given coordinates is read from the input image using the specified sampler (sampler) and stored in the pixel variable.

- The luminance value is computed by applying specific weightings to the RGB channels of the pixel.
- A new pixel value is created, representing a night vision effect, where the red and blue channels are set to 0, and the green channel is assigned the calculated luminance value. The alpha channel remains unchanged.
- Finally, the modified pixel value is written to the output image at the original coordinates.

Gaussian Blurring 7X1

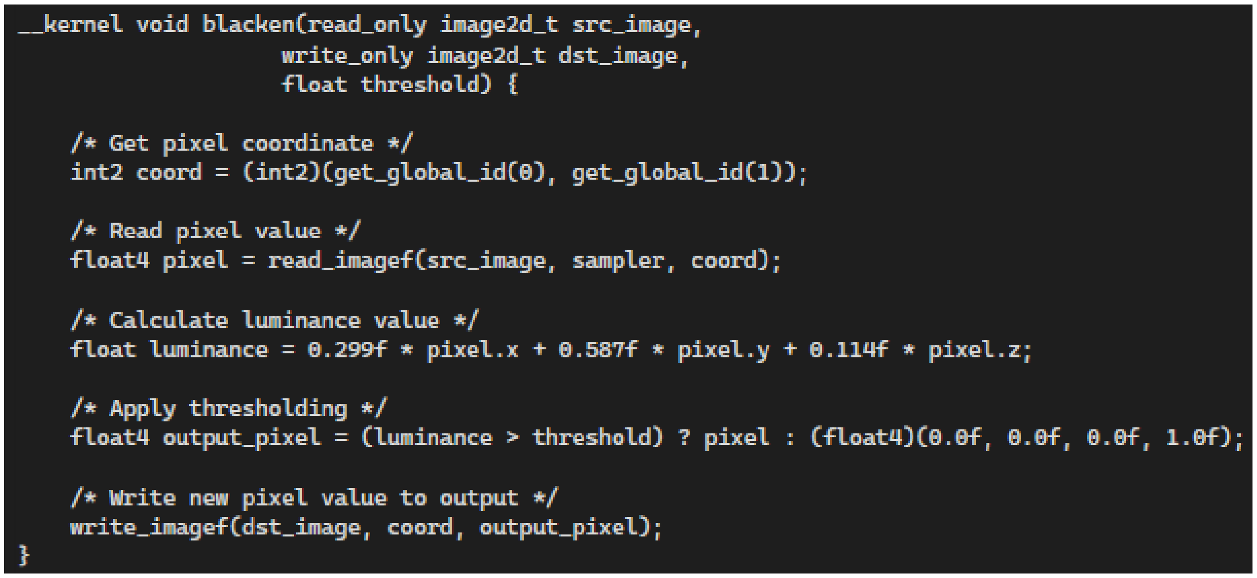


```
__kernel void gaussian_Blurring(read_only image2d_t src_image,
                    write_only image2d_t dst_image,
                    const int n) {

   /* Get work-item’s row and column position */
   int column = get_global_id(0);
   int row = get_global_id(1);

   /* Accumulated pixel value */
   float4 sum = (float4)(0.0);

   /* Filter's current index */
   int filter_index =  0;

   int2 coord;
   float4 pixel;

   // Gaussian blurring filter for vertical pass
   if(n == 1){
        coord.y = row;
       /* Iterate over the columns */
        for (int j = -3; j <= 3; j++) {
        coord.x = column + j;
        /* Read value pixel from the image */
        pixel = read_imagef(src_image, sampler, coord);
        /* Accumulate weighted sum */
        sum.xyz += pixel.xyz * GaussianBlurring[filter_index++];

            }

    }else{
        // Gaussian blurring filter for horizontal pass
            /* Iterate over the rows */
            coord.x = column;
            for(int i = -3; i <= 3; i++) {
                coord.y =  row + i;
                /* Read value pixel from the temporary image */
                pixel = read_imagef(src_image, sampler, coord);
                /* Accumulate weighted sum */
                sum.xyz += pixel.xyz * GaussianBlurring[filter_index++];
                 }

        }
   /* Write new pixel value to output */
   coord = (int2)(column, row);
   write_imagef(dst_image, coord, sum);
}
```

- The kernel function takes three arguments: a read-only input image (src_image), a write-only output image (dst_image), and an integer value (n) that determines the direction of the blur pass.
- The function retrieves the row and column positions of the current work-item using get_global_id(0) and get_global_id(1).

- An initial sum variable is initialized to (0.0) to accumulate the pixel values.
- Depending on the value of n, the function executes either the vertical or horizontal pass of the Gaussian blur.
- If n is equal to 1 (indicating the vertical pass), the function iterates over the columns with an offset of -3 to +3. It reads the pixel values from the input image at the corresponding coordinates, multiplies them with the corresponding Gaussian blur weight from the GaussianBlurring array, and accumulates the weighted sum in the sum variable.
- If n is not equal to 1 (indicating the horizontal pass), the function iterates over the rows with an offset of -3 to +3. It reads the pixel values from the input image, applies the same operations as in the vertical pass, and accumulates the weighted sum in the sum variable.
- Finally, the modified pixel value (the accumulated sum) is written to the output image at the original coordinates using the write_imagef function.

Blacken

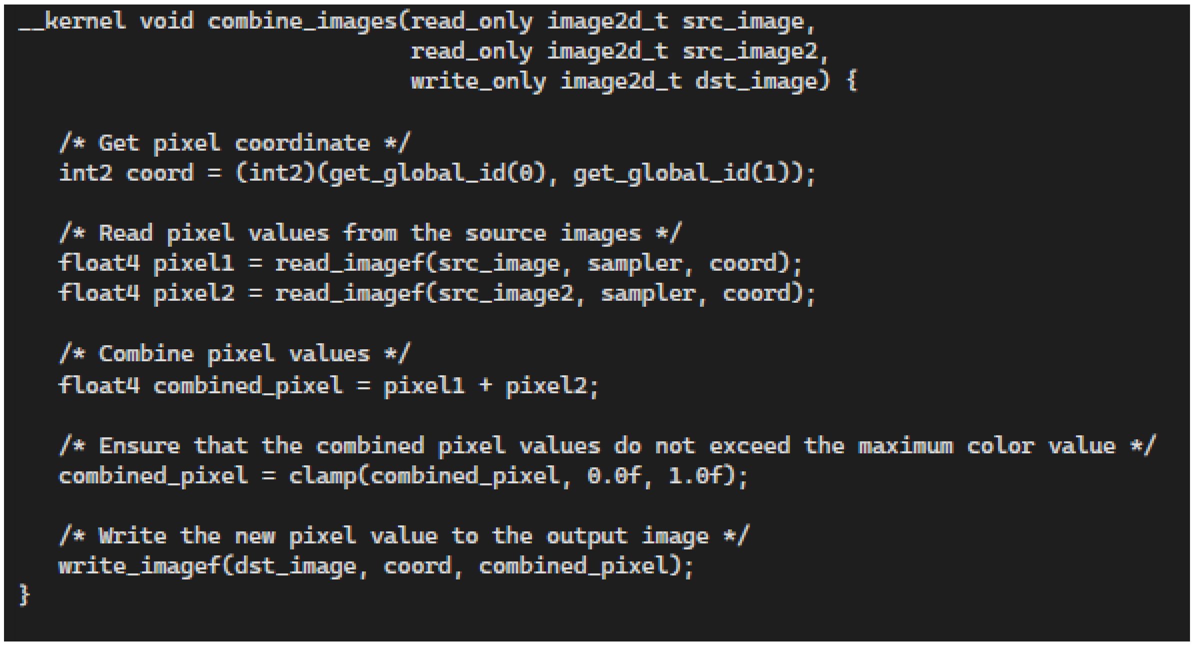


```
__kernel void blacken(read_only image2d_t src_image,
                    write_only image2d_t dst_image,
                    float threshold) {

    /* Get pixel coordinate */
    int2 coord = (int2)(get_global_id(0), get_global_id(1));

    /* Read pixel value */
    float4 pixel = read_imagef(src_image, sampler, coord);

    /* Calculate luminance value */
    float luminance = 0.299f * pixel.x + 0.587f * pixel.y + 0.114f * pixel.z;

    /* Apply thresholding */
    float4 output_pixel = (luminance > threshold) ? pixel : (float4)(0.0f, 0.0f, 0.0f, 1.0f);

    /* Write new pixel value to output */
    write_imagef(dst_image, coord, output_pixel);
}
```

- The kernel function takes three arguments: a read-only input image (src_image), a write-only output image (dst_image), and a floating-point value (threshold) that determines the luminance threshold for blackening.
- The function obtains the pixel coordinates of the current thread using get_global_id(0) and get_global_id(1).
- The pixel value at the given coordinates is read from the input image using the specified sampler (sampler).
- The luminance value is calculated by combining the color channels of the pixel with specific weights: 0.299 for the red channel (pixel.x), 0.587 for the green channel (pixel.y), and 0.114 for the blue channel (pixel.z).
- The thresholding operation is applied by comparing the calculated luminance value with the specified threshold. If the luminance value is greater than the threshold, the original

pixel value is retained; otherwise, a new pixel value with all channels set to 0 and alpha channel set to 1 is assigned.
- Finally, the modified pixel value is written to the output image at the original coordinates using the write_imagef function.

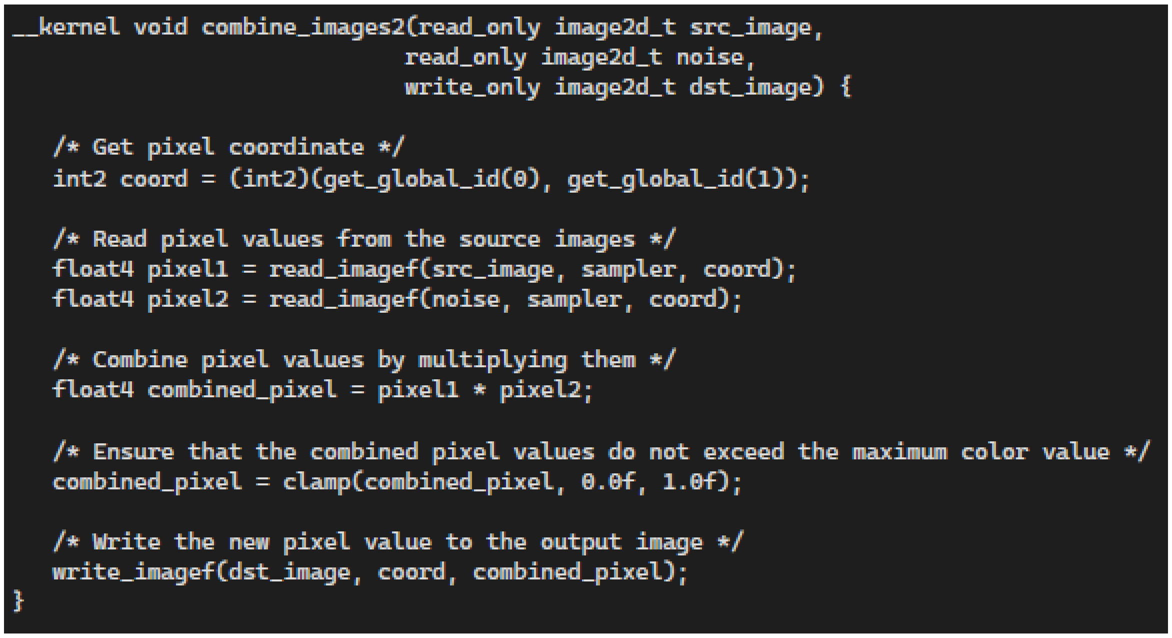
Combine Images



```
__kernel void combine_images(read_only image2d_t src_image,
                             read_only image2d_t src_image2,
                             write_only image2d_t dst_image) {

   /* Get pixel coordinate */
   int2 coord = (int2)(get_global_id(0), get_global_id(1));

   /* Read pixel values from the source images */
   float4 pixel1 = read_imagef(src_image, sampler, coord);
   float4 pixel2 = read_imagef(src_image2, sampler, coord);

   /* Combine pixel values */
   float4 combined_pixel = pixel1 + pixel2;

   /* Ensure that the combined pixel values do not exceed the maximum color value */
   combined_pixel = clamp(combined_pixel, 0.0f, 1.0f);

   /* Write the new pixel value to the output image */
   write_imagef(dst_image, coord, combined_pixel);
}
```

- The kernel function takes three arguments: two read-only input images (src_image and src_image2) and a write-only output image (dst_image).
- The function retrieves the pixel coordinates of the current thread using get_global_id(0) and get_global_id(1).
- The pixel values at the given coordinates are read from the source images using the specified sampler (sampler).
- The pixel values from the two source images are combined by adding them together, resulting in a new combined pixel value.
- To ensure that the combined pixel values do not exceed the maximum color value (1.0), the clamp function is used to limit the pixel values between 0.0 and 1.0.
- Finally, the modified pixel value is written to the output image at the original coordinates using the write_imagef function.

Combine Images2



```
__kernel void combine_images2(read_only image2d_t src_image,
                              read_only image2d_t noise,
                              write_only image2d_t dst_image) {

   /* Get pixel coordinate */
   int2 coord = (int2)(get_global_id(0), get_global_id(1));

   /* Read pixel values from the source images */
   float4 pixel1 = read_imagef(src_image, sampler, coord);
   float4 pixel2 = read_imagef(noise, sampler, coord);

   /* Combine pixel values by multiplying them */
   float4 combined_pixel = pixel1 * pixel2;

   /* Ensure that the combined pixel values do not exceed the maximum color value */
   combined_pixel = clamp(combined_pixel, 0.0f, 1.0f);

   /* Write the new pixel value to the output image */
   write_imagef(dst_image, coord, combined_pixel);
}
```

Similar to combine Images but the pixel values from the two source images are combined by multiplying them together, resulting in a new combined pixel value.