main.cxx

```cpp
#include <cxxopts.hpp> // command line argument parser library written by jarro2783 (https://github.com/jarro2783/cxxopts)
#include <ui.hxx>

auto main(int argc, char** argv) -> int{
    // flags for program arguments
    bool upAlphaFlag = false;
    bool lowAlphaFlag = false;
    bool numFlag = false;
    bool specialCharFlag = false;
    bool guiFlag = false;

    // other variable change by the flags
    unsigned int length = 0;
    int arg = -1;
    int state = 0; // execution state (0 for success)
    bool flagEnabled = false;

    // parse and logics for command line options
    cxxopts::Options options("APCSP Create Task", "Random Password Generator");
    options.add_options()
        ("A,upper", "Include upper case alphabets")
        ("a,lower", "Include lower case alphabets")
        ("n,number","Include numbers")
        ("s,special", "Include special characters")
        ("g,gui", "Run the GUI version of program, other passed arguments are ignored")
        ("h,help", "Print help")
        ("l,length", "Set the length of password", cxxopts::value<int>())
    ;

    cxxopts::ParseResult result;

    try {
        result = options.parse(argc, argv);
    } catch (const cxxopts::exceptions::parsing &e) {
        printHelp();
        std::cerr << e.what() << "\n";
        return 1;
    }

    if (result.count("help")) {
        printHelp();
        return 0;
    }
    if (result.count("upper")) upAlphaFlag = true;
    if (result.count("lower")) lowAlphaFlag = true;
    if (result.count("number")) numFlag = true;
    if (result.count("special")) specialCharFlag = true;
    flagEnabled = checkFlags(upAlphaFlag, lowAlphaFlag, numFlag, specialCharFlag);
    if (result.count("gui")) guiFlag = true;
    if (result.count("length") && !guiFlag && flagEnabled) {
        length = result["length"].as<int>();
    } else if (!guiFlag && flagEnabled) {
        printHelp();
        printLine("Error: no length argument");
        return 1;
    }

    // run GUI version if -g option is present
    if (guiFlag) {
        state = rungui(argc, argv);
    } else if (length > 0 && flagEnabled){ // otherwise run CUI version
        state = runcui(length, upAlphaFlag, lowAlphaFlag, numFlag, specialCharFlag);
    } else { // if no argument specified, report error and set execution state 1 (failure)
        printHelp();
        printLine("Error: No argument specified");
        state = 1;
    }
    return state;
}
```

passgen.hxx

```cpp
#ifndef PASSGEN_HXX
#define PASSGEN_HXX

namespace PassGen {
    // declare functions (not defined yet)
    /**
    * @brief Get the list of upper alphabet letters
    * @return char* - List of upper alphabet letters
    */
    auto getUpperAlpha() -> char*;
    /**
    * @brief Get the list of lower alphabet letters
    * @return char* - List of lower alphabet letters
    */
    auto getLowerAlpha() -> char*;
    /**
    * @brief Get the list of numbers
    * @return char* - List of numbers
    */
    auto getNumber() -> char*;
    /**
    * @brief Get the list of special characters
    * @return char* - List of special characters
    */
    auto getSpecialChars() -> char*;
    /**
    * @brief generate the password from list of characters and specified length
    * @param charList list of characters (pure C string)
    * @param len length of password to be generated (int)
    * @return (char*) - generated password
    */
    auto passGen(const char *charList, const unsigned int& len) ->
    char*;
}

#endif // PASSGEN_HXX
```

passgen.cxx

```
#include <cstdlib>
#include <ctime>
#include <utils.hxx>
#include <passgen.hxx>

auto PassGen::getLowerAlpha() -> char* {
    const int numOfLetters = 26; // 26 letters
    char* output = nullptr; // initialize pointer
    output = (char*)malloc(numOfLetters * sizeof(char) + 1); //
    allocate memory for 26 letters and a terminate character
    if (output == nullptr) {return nullptr;} // check if memory
    allocation is failed
    const int offset = 97; // 97th letter in ASCII (a)
    // adds 26 letters (a-z)
    for (int i = 0; i < numOfLetters; i++) {
        output[i] = (char)(offset + i);
    }
    output[numOfLetters] = '\0'; // add terminate character at end
    return output;
}

auto PassGen::getUpperAlpha() -> char* {
    const int numOfLetters = 26; // 26 letters
    char* output = nullptr; // initialize pointer
    output = (char*)malloc(numOfLetters * sizeof(char) + 1); //
    allocate memory for 26 letters + terminate character
    if (output == nullptr) {return nullptr;} // check if memory
    allocation is failed
    const int offset = 65; // 65th letter in ASCII (A)
    // adds 26 letters (A-Z)
    for (int i = 0; i < numOfLetters; i++) {
        output[i] = (char)(offset + i);
    }
    output[numOfLetters] = '\0'; // add terminate character at end
    return output;
}

auto PassGen::getNumber() -> char* {
    const int numOfLetters = 10; // 10 letters
    char* output = nullptr; // initialize pointer
    output = (char*)malloc((numOfLetters) * sizeof(char) + 1);
    if (output == nullptr) {return nullptr;} // check if memory
    allocation is failed
    const int offset = 48; // 48th letter in ASCII (0)
    // adds 10 letters (0-9)
    for (int i = 0; i < numOfLetters; i++) {
        output[i] = (char)(offset + i);
    }
    output[numOfLetters] = '\0'; // add terminate character at end
    return output;

}

auto PassGen::getSpecialChars() -> char* {
    const int numOfLetters = 31; // 31 symbols
    char* output = nullptr; // initialize pointer
    output = (char*)malloc((numOfLetters) * sizeof(char) + 1);
    if (output == nullptr) {return nullptr;} // check if memory
    allocation is failed
    int ind = 0;
    // loop config and range exclusion config
    const int start = 33;
    const int end = 127;
    const int numStart = 48;
    const int numEnd = 57;
    const int upperStart = 65;
    const int upperEnd = 90;
    const int lowerStart = 97;
    const int lowerEnd = 122;
    // adds special characters to output
    for (int i = start; i < end; i++) {
        if ((numStart <= i && i <= numEnd) || (upperStart <= i && i
     <= upperEnd) || (lowerStart <= i && i <= lowerEnd)) {
            continue; // skip at the number, upper case and lower
    case alphabets
        }
        output[ind] = (char)(i);
        ind++;
    }
    output[numOfLetters] = '\0'; // add terminate character at end
    return output;
}

auto PassGen::passGen(const char *charList, const unsigned int& len
    ) -> char* {
    if (strSize(charList) == 0) {return nullptr;}

    std::srand(time(nullptr)); // set random seed to current time

    unsigned int randomCharPos = 0; // position of charList which
    will be randomly selected
    const char termChar = '\0';
    const char backSlash = '\\';
    const int charListSize = strSize(charList);
    char currentLetter = 0;
    char previousLetter = 0;

    char* output = new char[len+1]; // length of password + 1
    terminating char
    if (output == nullptr) {return nullptr;} // return 0 on the
    failiure of memory allocation

    while (strSize(output) != len) { // to make sure output is in
    desired length
    for (int i = 0; i <= len; i++) {
        if (i == len) {output[i] = termChar;} // ends with
    terminating char
        else { // adds other chars otherwise

        // adds random character from charList
        randomCharPos = std::rand()%charListSize;
        output[i] = charList[randomCharPos];

        // set current and previous letter for check
        currentLetter = output[i];
        previousLetter = output[i-1];

        // checks escape sequences which causes issues
        while (previousLetter == backSlash &&
        ((currentLetter == 'a') || // '\a'
        (currentLetter == 'b') || // '\b'
        (currentLetter == 'c') || // '\c'
        (currentLetter == 'n') || // '\n'
        (currentLetter == 'f') || // '\f'
        (currentLetter == 'r') || // '\r'
        (currentLetter == 't') || // '\t'
        (currentLetter == 'U') || // '\U'
        (currentLetter == 'u') || // '\u'
        (currentLetter == 'v') || // '\v'
        (currentLetter == 'x') // '\x'
        )) {
            // regenerate random letter
            randomCharPos = std::rand()%charListSize;
            output[i] = charList[randomCharPos];

            // re-set current letter
            currentLetter = output[i];
        }}
    }}
    return output;
}
```

utils.hxx

```
#ifndef UTILS_HXX
#define UTILS_HXX

#include <iostream>

/**
* @brief return the size(length) of string (pure C char list)
* @param str pure C string to be measured
* @return (int) - size(length) of string
*/
inline auto strSize(const char *str) -> int {
    int out = 0;
    int index = 0;
    while (str[index] != 0) {
        index++;
        out++;
    }
    return out;
}
```

```cpp

/**
* @brief print the passed in argument
* @param object takes any type of input that is able to stdout to
    the console
* @return (void) - console output the object
*/
template<class T>
inline auto printLine(T object) -> void {
    std::cout << object << std::endl;
}

/**
* @brief prints the help for the console application
* @return (void) - console output of help document
*/
inline auto printHelp() -> void {
    printLine("APCSPCreateTask - Random Password Generator\n");
    printLine("[Usage]: APCSPCreateTask [-A -a -n -s -g] -l <length
    >\n");
    printLine("[Options]:\n");
    printLine("\t-A : include upper case alphabets in password\n");
    printLine("\t-a : include lower case alphabets in password\n");
    printLine("\t-n : include numbers in password\n");
    printLine("\t-s : include special characters in password\n");
    printLine("\t-l <number> : set the length of the password\n");
    printLine("\t-g : run in GUI regardless of the previous options
    \n");
    printLine("\t-h : print this help\n");
}

/**
* @brief checks if any flag is enabled
* @param upper boolean flag for upper case letters
* @param lower boolean flag for lower case letters
* @param num boolean flag for numbers
* @param special boolean flag for special characters
* @return (bool) - returns true if one of any flag is enabled
*/
inline auto checkFlags(bool upper, bool lower, bool num, bool
    special) -> bool {
    int count = 0;
    if (upper) {count++;}
    if (lower) {count++;}
    if (num) {count++;}
    if (special) {count++;}
    return count > 0;
}

#endif // UTILS_HXX
```

ui.hxx

```cpp
#ifndef UI_HXX
#define UI_HXX

#include <string>
```

```
#include <cstring>
#include <utils.hxx>
#include <passgen.hxx>
#include <gtkui.hxx>
using namespace PassGen;

/**
* @brief runs the CUI version of program
* @param len length of the password (int)
* @param upper boolean flag for upper case alphabets
* @param lower boolean flag for lower case alphabets
* @param num boolean flag for numbers
* @param special boolean flag for special characters
* @return (int) - execution state of program
*/
auto runcui(const unsigned int& len, const bool& upper, const bool&
     lower, const bool& num, const bool& special) -> int {
    // appends letters to input
    std::string input;
    if (upper) {input += getUpperAlpha();}
    if (lower) {input += getLowerAlpha();}
    if (num) {input += getNumber();}
    if (special) {input += getSpecialChars();}

    // converts to std::string to pure C string
    char *cInput = new char[input.length() + 1];
    strcpy(cInput, input.c_str());

    // generate, prints password, and exit with status of 0 (
    success)
    char *out = passGen(cInput, len);
    std::cout << "Generated Password: " << out << std::endl;
    return 0;
}

/**
* @brief runs the GUI version of program
* @param argc argument count (int)
* @param argv arguments (List of pure C strings)
* @return (int) - execution state of program
*/
auto rungui(int argc, char** argv) -> int {
    // run the GTK application
    auto app = Gtk::Application::create("apcsp.passgen");
    return app->make_window_and_run<PassGenUI>(0,nullptr); // run
    GTK app with no arguments
}

#endif // UI_HXX
```

gtkui.hxx

```
#ifndef GTKUI_HXX
#define GTKUI_HXX

#include <gtkmm.h>
#include <passgen.hxx>
```

```

class PassGenUI : public Gtk::Window
{
    public:
        PassGenUI(); // constructor
        ~PassGenUI() override;// destructor

        /**
        * @brief The button event for m_generate_button
        * @return (void) Generate password, set to m_output_buffer,
     and show to the user
        */
        auto on_generate_button_clicked() -> void; // button event
    private:
        const int winHeight = 480;
        const int winWidth = 640;
        const int widgetMargin = 10;
        const int maxLength = 8192;
        Gtk::CheckButton m_upper_check, m_lower_check, m_num_check,
     m_special_chars_check; // checkboxes
        Gtk::Box m_char_checks, m_output_box, m_main_box; // boxes
    (containers)
        Gtk::Label m_title;
        Gtk::Button m_generate_button;
        Gtk::SpinButton m_num_input; // length input
        Glib::RefPtr<Gtk::Adjustment> m_num_input_adj = Gtk::
    Adjustment::create(0, 0, maxLength); // sets range for
    m_num_input (0-maxLength)
        Glib::RefPtr<Gtk::CssProvider> m_output_style = Gtk::
    CssProvider::create();
        Gtk::ScrolledWindow m_output_scroll;
        Gtk::TextView m_output;
        Glib::RefPtr<Gtk::TextBuffer> m_output_buffer = Gtk::
    TextBuffer::create(); // text buffer for m_output
};

#endif // GTKUI_HXX
```

gtkui.cxx

```
#include <gtkui.hxx>

PassGenUI::PassGenUI():
// initialize widgets
m_generate_button("Generate"),
m_main_box(Gtk::Orientation::VERTICAL,widgetMargin),
m_char_checks(Gtk::Orientation::VERTICAL,widgetMargin),
m_output_box(Gtk::Orientation::VERTICAL, widgetMargin),
m_upper_check("Include Upper Case Letters"),
m_lower_check("Include Lower Case Letters"),
m_num_check("Include Numbers"),
m_special_chars_check("Include Special Characters"),
m_title("Password Generator")
{
    // set window props
    set_title("AP CSP Create Task - Password Generator");
    set_default_size(winWidth,winHeight);
    // link the button event to the function
    m_generate_button.signal_clicked().connect(sigc::mem_fun(*this,
     &PassGenUI::on_generate_button_clicked));

    // populate the widgets and other boxes in main box
    set_child(m_main_box);
    m_main_box.set_margin(widgetMargin);
    m_num_input.set_adjustment(m_num_input_adj);
    m_main_box.append(m_title);
    m_main_box.append(m_char_checks);
    m_main_box.append(m_num_input);
    m_main_box.append(m_generate_button);
    m_main_box.append(m_output_box);

    // populate the widgets in letter configuration section
    m_char_checks.append(m_upper_check);
    m_char_checks.append(m_lower_check);
    m_char_checks.append(m_num_check);
    m_char_checks.append(m_special_chars_check);

    // populate the widgets in ouput section and configure widgets
    m_output_scroll.set_child(m_output);
    m_output_scroll.set_expand();
    m_output.set_editable(false);
    m_output.set_monospace(true);
    m_output.set_cursor_visible(false);
    m_output_style->load_from_data("#m_output {font-size: 14pt;}");
    m_output.set_name("m_output");
    m_output.get_style_context()->add_provider(m_output_style, 1);
    m_output.set_wrap_mode(Gtk::WrapMode::CHAR);
    m_output_box.append(m_output_scroll);
    m_output.set_buffer(m_output_buffer);
}

PassGenUI::~PassGenUI() = default;

auto PassGenUI::on_generate_button_clicked() -> void {
    // appends letters to input according to the flags
    std::string input;
    if (m_upper_check.get_active()) {input += PassGen::
    getUpperAlpha();}
    if (m_lower_check.get_active()) {input += PassGen::
    getLowerAlpha();}
    if (m_num_check.get_active()) {input += PassGen::getNumber();}
    if (m_special_chars_check.get_active()) {input += PassGen::
    getSpecialChars();}

    // convert std::string to pure C string
    char* cInput = new char[input.length() + 1];
    strcpy(cInput, input.c_str());

    // get the length of password to be generated
    int len = m_num_input.get_value_as_int();

    // generate password
    char* passwd = PassGen::passGen(cInput, len);

    // Show error alert dialog when passwd returns nullptr and stop execution of function
    if (passwd == nullptr) {
        Glib::RefPtr<Gtk::AlertDialog> m_Alert = Gtk::AlertDialog::create();
        m_Alert->set_message("Error!");
        m_Alert->set_detail("Please check the form");
        m_Alert->show(*this);
        return;
    }

    // Show the passwd by setting text and buffer
    Glib::ustring output = Glib::convert(passwd, "UTF-8", "ISO-8859-1"); // convert to appropriate type and encoding of text
    m_output_buffer->set_text(output);
    m_output.set_buffer(m_output_buffer);
}
```