ECE297 Quick Start Guide
GNU Readline

I'm not a very good writer, but I'm an excellent rewriter.
–James Michener

1 Introduction

GNU Readline[1] is an open source library useful for building command parsers. Unlike `iostream`'s which are part of the C++ standard library, Readline includes support for advanced features including:

- Interactive line editing, which allows changes to a line as it is typed (backspace, arrow-keys, delete etc.)
- Command history, which allows quick retrieval of previous commands using the up and down arrow-keys
- Automatic command completion, allowing partially entered text to be matched against expected values

These features can make your application behave much more like a regular command terminal, improving its usability.

This document is organized as follows:

- Section 2 illustrates some of the limitations of `iostreams`,
- Section 3 describes how to use readline in your application, and
- Section 4 describes how to build and run the demo applications.

2 The limitations of `iostreams`

Listing 1 shows the basic structure of a command parser using the C++'s `cin` stream to retrieve input from the user.

```
#include <iostream>

using namespace std;

int main (int argc, char* argv[]) {

    string buf; //Buffer of line entered by user

    cout << "prompt> "; //Prompt the user for input
    while(getline(cin, buf)) { //Get the line

        //Currently we just print out the input line
```

[1] For detailed documentation see: www.gnu.org/s/readline

```
13         cout << buf << endl;
14 
15         //And exit if the user requested it
16         if(buf == "exit" || buf == "quit") {
17             break;
18         }
19 
20         cout << "prompt> "; //Prompt the user for input
21     }
22 
23     return 0;
24 }
```

Listing 1: iostream_demo.cpp

The key operation is the call to `getline(cin, buf)` at line 10, which reads one line of a user's input into the string `buf`. The user's input is then processed in the body of the while loop, echoing the input back to `cout` (line 13), and exiting the loop if the user entered 'exit' or 'quit' (lines 16-18).

An example user session is shown in Listing 2:

```
prompt> hello world!
hello world!
prompt> echo
echo
prompt> echo ehco^?^?^?          #<Backspace> <Backspace> <Backspace>
prompt> hlelo^[[D^[[D^[[3~       #<LeftArrow> <LeftArrow> <Delete>
```

Listing 2: Session using iostreams

Using `iostreams` there is no easy way for the user to edit what they have already entered to correct typos. Luckily Readline can fix these limitations, and also add additional features.

3 Readline Tutorial

3.1 Interactive Line Editing

One of the most useful features of Readline is its support for interactive command line editing. If you make a typo you can edit your already typed command directly. You do not have to restart from scratch! This can be very useful when entering long commands.

Listing 3 illustrates the command prompt usage using readline. Unlike in Listing 2, when using Readline we can use the arrow, delete and backspace keys to correct our typos.

```
prompt> hello world!
hello world!
prompt> echo
echo
prompt> echo ehco  #Spelling mistake! <Backspace>
prompt> echo ehc   #<Backspace> <Backspace>
prompt> echo e     #<c> <h> <o>
prompt> echo echo  #<Enter>
echo echo
prompt> heoll      #Spelling mistake! <LeftArrow> <LeftArrow> <LeftArrow> <Delete>
prompt> hell       #<RightArrow> <RightArrow> <o>
prompt> hello      #<Enter>
hello
```

Listing 3: Command session using Readline.

To use readline we need to call the appropriate functions as illustrated in Listing 4.

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 
 4  #include <readline/readline.h>
 5 
 6 using namespace std;
 7 
 8 int main (int argc, char* argv[]) {
 9 
10     char* buf; //Buffer of line entered by user
11 
12     while((buf = readline("prompt> ")) != NULL) { //Prompt the user for input
13         //Currently we just print out the input line
14         cout << buf << endl;
15 
16         //And exit if the user requested it
17         if(strcmp(buf, "exit") == 0 || strcmp(buf, "quit") == 0) {
18             break;
19         }
20 
21         //readline() allocates a new string for every line,
22         //so we need to free the current one after we've finished
23         free(buf);
24         buf = NULL; //Mark it null to show we freed it
25     }
26 
27     //If the buffer wasn't freed in the main loop we need to free it now
28     // Note: if buf is NULL free does nothing
29     free(buf);
30 
31     return 0;
32 }
```

Listing 4: A Readline based command parser equivalent to the one in Listing 1.

The key difference between using `iostreams` and Readline occurs at line 12 of Listing 4, where we use the Readline library's `readline()` function instead of `getline()`.

Note that Readline is written in C (not C++) so readline returns `char*`, which we must explicitly free (lines 23-24, 29) to avoid memory leaks. You can of course convert a `char*` to a `std::string` if desired.

3.2 Enabling Command History

Many command shells (terminals) allow you to scroll through previously entered commands by using the up and down arrow keys. We can do the same thing with Readline as shown in Listing 5:

```
prompt> This is a really long command I don't want to re-type!
This is a really long command I don't want to re-type!
prompt>                                                       #<UpArrow>
prompt> This is a really long command I don't want to re-type!  #<Enter>
This is a really long command I don't want to re-type!
```

Listing 5: Using Readline's history feature.

Listing 6 shows the modifications added to enable history.

```
 1 #include <iostream>
 2 #include <cstdlib>
 3 
 4 #include <readline/readline.h>
 5 #include <readline/history.h>
 6 
 7 using namespace std;
 8 
 9 int main (int argc, char* argv[]) {
10 
11     char* buf; //Buffer of line entered by user
12 
13     while((buf = readline("prompt> ")) != NULL) { //Prompt the user for input
14         if(strcmp(buf, "") != 0) { //Only save non-empty commands
15             add_history(buf);
16         }
17 
18         //Currently we just print out the input line
19         cout << buf << endl;
20 
21         //And exit if the user requested it
22         if(strcmp(buf, "exit") == 0 || strcmp(buf, "quit") == 0) {
23             break;
24         }
25 
26         //readline generates a new buffer every time,
27         //so we need to free the current one after we've finished
28         free(buf);
29         buf = NULL; //Mark it null to show we freed it
30     }
31 
32     //If the buffer wasn't freed in the main loop we need to free it now
33     // Note: if buf is NULL free does nothing
34     free(buf);
35 
36     return 0;
37 }
```

Listing 6: Extending Listing 4 to support command history.

We simply need to include the `readline/history.h` header file, and call the `add_history()` function (line 15) on anything we want to add to the history. Note that to avoid adding blank commands in the history we check that the buffer is non-empty first (line 14).

3.3 Command Auto-Completion

Command auto-completion is very useful when a command takes arguments from a large set of potential values. A classic example is providing filenames on the command line, as shown in Listing 7.

```
> ls #List the files in the current directory
iostream_demo.cpp  Makefile  readline_autocomplete_demo.cpp  readline_demo.cpp
> ls -l r                                         #<Tab>
> ls -l readline_                                 #<Tab> <Tab>
readline_autocomplete_demo.cpp  readline_demo.cpp
> ls -l readline_a                                #<a>
> ls -l readline_autocomplete_demo.cpp            #<Tab> <Enter>
```

```
-rw-rw-r-- 1 kmurray kmurray 4303 Feb 11 17:27 readline_autocomplete_demo.cpp
```

Listing 7: Filename completion on the command line.

Using Readline we can define our own auto-completion scheme for the commands used by our application. For instance, if we had a command `find_intersection` that requires an intersection name argument we could auto-complete partially entered names with the Tab key:[2]

```
prompt> find_intersection B                                            #<Tab> <Tab>
Bay Street & Dundas Street West          Bloor Street West & Bay Street
Bay Street & Wellington Street West
prompt> find_intersection Bl                                           #<Tab>
prompt> find_intersection Bloor Street West & Bay Street
...
#We may need quotes if the intersection names have
#spaces and the potential matches are ambiguous
prompt> find_intersection "Ba                                          #<Tab>
prompt> find_intersection "Bay Street &                                #<Tab> <Tab>
Bay Street & Dundas Street West          Bay Street & Wellington Street West
prompt> find_intersection "Bay Street & D                              #<D> <Tab>
prompt> find_intersection "Bay Street & Dundas Street West"
```

Listing 8: Custom command completion using Readline.

Adding support for auto-completion requires modifications to `main()` which are highlighted in Listing 9:

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>

#include <readline/readline.h>
#include <readline/history.h>

using namespace std;

char** command_completion(const char* stem_text, int start, int end);
char* intersection_name_generator(const char* stem_text, int state);

int main (int argc, char* argv[]) {

    //Use tab for auto completion
    rl_bind_key('\t', rl_complete);
    //Use our function for auto-complete
    rl_attempted_completion_function = command_completion;
    //Tell readline to handle double and single quotes for us
    rl_completer_quote_characters = strdup("\"\'");

    char* buf; //Buffer of line entered by user
    while((buf = readline("prompt> ")) != NULL) { //Prompt the user for input
        if(strcmp(buf, "") != 0)
            add_history(buf); //Only add to history if user input not empty

        //Currently we just print out the input line
        cout << buf << endl;

```

[2]Note that this example is not exactly the same as the intersection naming convention we are using in your mapper project.

```
30         //And exit if the user requested it
31         if(strcmp(buf, "exit") == 0 || strcmp(buf, "quit") == 0) {
32             break;
33         }
34 
35         //readline generates a new buffer every time,
36         //so we need to free the current one after we've finished
37         free(buf);
38         buf = NULL; //Mark it null to show we freed it
39     }
40 
41     //If the buffer wasn't freed in the main loop we need to free it now
42     // Note: if buf is NULL free does nothing
43     free(buf);
44 
45     return 0;
46 }
```

Listing 9: Modifications to main to support command completion.

To enable command line auto-completion we first tell Readline what keyboard key should be used. In this case at line 16 we indicate that the Tab key (symbol '\t') should be used for auto-completion. Next, at line 18 we inform Readline that we want to use a custom function `command_completion()` to perform auto-completion. Finally, at line 20 we tell Readline to handle quoting for us. This allows us to perform completion even if the partially entered values contain spaces — provided the values are enclosed within single (') or double (") quotes.

For our purposes we will assume that the intersection names we want to match against for auto-completion are in the vector `intersection_names` as shown in Listing 10.

```
//An example list of intersections
vector<const char*> intersection_names = {
    "Yonge Street & Eglinton Avenue East & Eglinton Avenue West",
    "Bay Street & Dundas Street West",
    "Bay Street & Wellington Street West",
    "Yonge Street & Dundas Street West & Dundas Street East",
    "Yonge Street & College Street & Carlton Street",
    "Yonge Street & Queen Street West & Queen Street East",
    "Bloor Street West & Bay Street",
    "University Avenue & Front Street West & York Street",
    "King Street West & Bay Street",
    "St. George Street & Harbord Street & Hoskin Avenue",
    "College Street & Bay Street",
    "Queen Street West & John Street",
    "Avenue Road & Bloor Street West & Queen's Park",
    "Front Street West & John Street & Convention centre loading dock",
    "University Avenue & Wellington Street West"
};
```

Listing 10: Intersection names used for auto-completion.

In order to perform the auto-completion we must now define the `command_completion()` function. One potential implementation is shown in Listing 11. Since this is a user defined function you can use it to customize auto-completion to your liking.

```
//Given the stem 'stem_text' perform auto completion.
//It returns an array of strings that are potential completions
//
//Note:
 5 // 'start' and 'end' denote the location of 'stem_text' in the global
 6 // 'rl_line_buffer' variable which contains the users current input line.
 7 // If you need more context information to determine how to complete 'stem_text'
 8 // you could look at 'rl_line_buffer'.
 9 char** command_completion(const char* stem_text, int start, int end) {
10     char ** matches = NULL;
11 
12     if(start != 0) {
13         //Only generate completions if 'stem_text'
14         //is not the first thing in the buffer
15         matches = rl_completion_matches(stem_text, intersection_name_generator);
16     }
17 
18     return matches;
19 }
```

Listing 11: The function used to generate matching strings for auto-completion.

When the user requests auto-completion by hitting the Tab key, Readline will call our `command_completion()` function[3], and return an array of matching strings (i.e. `char**`).

As shown on line 9, `command_completion()` takes three arguments, the most important of which is `stem_text`. The `stem_text` variable contains the user's (partially) entered value[4], and is used to determine potential completions. The other arguments (`start` and `end`) indicate where in the line's buffer the `stem_text` is located.

To generate the set of potential completions (line 15) we make use of Readline's `rl_completion_matches()` function, which takes both the stem to be matched and a *generator* function as arguments. Since we are aiming to complete intersection names we provide our own `intersection_name_generator()` generator function, which is defined in Listing 12.

```
//Called repeatedly for a given 'stem_text'. Each time it returns a potential
//match.  When there are no more matches it returns NULL.
//
//The 'state' variable is zero the first time it is called with a given
//'stem_text', and positive afterwards.
char* intersection_name_generator(const char* stem_text, int state) {
    //Static here means a variable's value persists across function invocations
    static int count;

    if(state == 0) {
        //We initialize the count the first time we are called
        //with this stem_text
        count = -1;
    }

    int text_len = strlen(stem_text);

    //Search through intersection_names until we find a match
    while(count < (int) intersection_names.size()-1) {
        count++;
        if(strncmp(intersection_names[count], stem_text, text_len) == 0) {
            //Must return a duplicate, Readline will handle
            //freeing this string itself.
            return strdup(intersection_names[count]);
        }
```

[3]Since we set the `rl_attempted_completion_function` variable in Listing 9 to `command_completion`.

[4]For example, at line 4 of Listing 8, `stem_text` would be "Bl".

```
26     }
27 
28     //No more matches
29     return NULL;
30 }
```

Listing 12: Using Readline to perform command completion.

In Readline parlance a generator function is a function called multiple times with the same value of `stem_text`. Each time it is called the generator will return another matching string. When there are no more matches it returns `NULL`.

The second argument to `intersection_name_generator()`, `state` is used to indicate whether the function is being called with a specific value of `stem_text` for the first time (`state == 0`) or a subsequent time (`state > 0`). In `intersection_name_generator()` we initialize a static counting variable `count` which persists across function invocations when `state == 0`. The `count` variable is then used to mark the generators position in the `intersection_name` vector. The function then searches through the `intersection_name` vector looking for the next intersection which starts with the stem (line 21) which is then returned (line 24). This way subsequent calls with the same value of `stem_text` generate all the potential matches. Once the entire vector has been search the generator returns `NULL` indicating no more matches exist (line 29).

4 Building the Readline Demo Examples

The associated zip file contains the example code from Sections 2, 3.2 and 3.3. To build the examples run `make` from the extracted folder. You can then run the demo programs `iostream_demo`, `readline_demo`, `readline_autocomplete_demo`.

Note that to compile a program using readline there are two requirements:

1. The readline library must be installed on your system.

   This is already the case for the EECG UG systems. On Ubuntu/Debian based systems it can be installed with the command `sudo apt-get install libreadline6 libreadline6-dev`

2. You must link to the readline library when building your application.

   For example, using gcc/g++ you should provide the `-lreadline` option. If this option is not already used by your Makefile you must add to the compiler options it when the executable is linked.