Chapter 3
Command Line Parsing

3.1 `argc` and `argv`

When running a program, you may need to pass in parameters via the command line to dictate how your program will run. These parameters are known as **command line arguments**. For example, consider the following command line input:

```
./game --level 25 --difficulty 10 --multiplayer
```

Here, the first term `"./game"` runs an executable named `game` in the current directory. The subsequent arguments that begin with a dash are known as **command line options**, and they can be used to specify certain features of the program. These options follow the executable name on the command line. For example, the above command may have been used to start the game at level 25, with a difficulty level of 10, and with multiplayer mode on. In this chapter, we will discuss mechanisms that allow you to handle command line options in your program.

If you want your program to support command line options, you should pass the following arguments into your `main()` function.

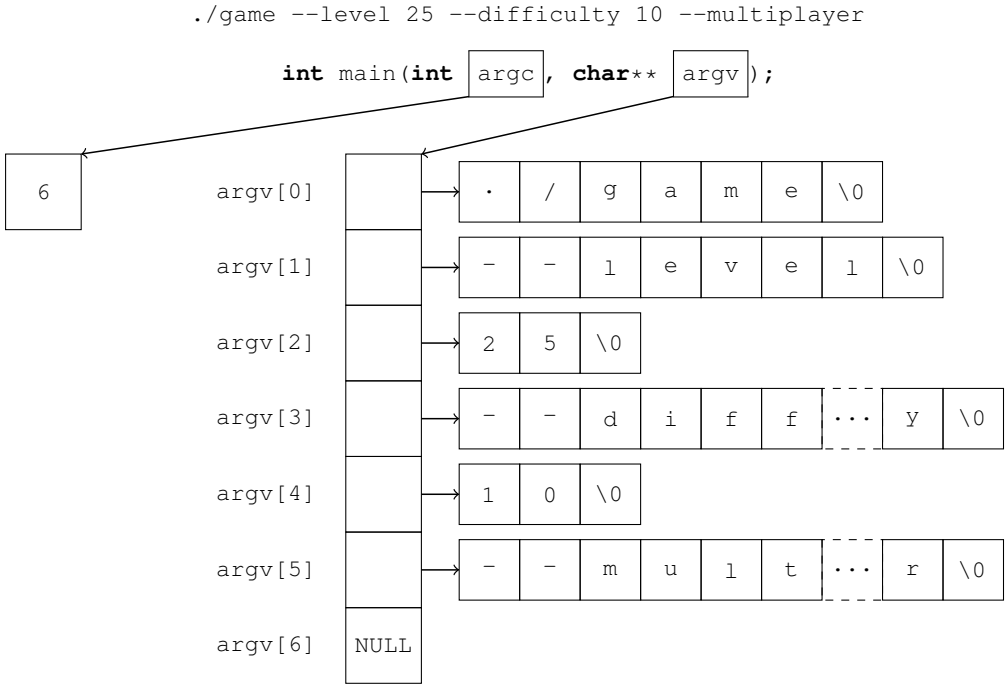
```
int main(int argc, char* argv[]);
```

— or —

```
int main(int argc, char** argv);
```

The first argument, `argc`, stores the number of arguments that are passed into the program from the command line. In the `game` example, there are six arguments: `"./game"`, `"--level"`, `"25"`, `"--difficulty"`, `"10"`, and `"--multiplayer"`. The second argument, `argv`, is an array of size `argc` that holds pointers to character arrays (also known as C-strings), where each C-string stores the contents of a command line argument. The name of the program itself is included as the first argument, and it is located at index 0 of the `argv` array.

A visualization of `argc` and `argv` for the example command is shown below.



All the arguments in `argv`, including the numbers, are stored as C-strings. A C-string is a character array that ends with a null character, or `'\0'` (this will be covered in more detail in chapter 16). Thus, if you want to retrieve the integer value of an argument in `argv`, you must first convert it from a C-string to an `int`. This can be done using the `atoi()` function, which accepts a C-string and returns the integer it represents (if the C-string cannot be interpreted as an integer, `atoi()` returns 0). For example, `atoi(argv[2])` returns `25` as an `int`.

Command line arguments related to input redirection are *not* included as a part of `argc` and `argv`. Consider the following example, which runs the same command as before, but with input and output files specified:

```
./game --level 25 --difficulty 10 --multiplayer < input.txt > output.txt
```

In this example, the arguments pertaining to input and output redirection are not considered as a part of `argc` and `argv`. The value of `argc` is still `6`, and the `argv` array will *not* store "`<`", "`input.txt`", "`>`", or "`output.txt`". Simply put, the values of `argc` and `argv` will be the same as it was before, when redirection was not specified.

3.2 Switch Statements and Enumerated Types

※ 3.2.1 *Switch Statements*

Before we move forward, we will go over the **switch statement** in C++, which will be used to parse command line options in the next section (and is a programming concept that is quite useful in general). Switch statements are essentially an alternative if-else structure for primitive types. A switch statement is not only neater than a long if-else chain, but depending on the cases involved, the compiler may also be able to optimize a switch statement to improve performance. The basic structure of a switch statement is shown below:

```
1  switch (variable_to_test)
2  {
3    case potential_value_1:
4    {
5      /* code to run */
6      break;
7    }
8    case potential_value_2:
9    {
10     /* code to run */
11     break;
12   }
13   ...
14   default:
15   {
16     /* code to run */
17   }
18 }
```

The variable you want to check goes into the switch statement. Just like the argument of a traditional if statement, the value of this variable dictates what code gets run.

These are potential values that the variable you want to test can take on. If the value of the variable is `potential_value_1`, for example, the code under the first case statement runs.

The `break` statements terminates the switch statement. If not included, the program falls through and runs code in the next case.

You can have any number of cases, or values that the variable you're testing can take on.

If the value of the variable matches none of the provided cases, the code associated with the `default` case label runs (this is kind of like the else case of an if-else statement).

To write a switch statement, pass in the variable whose value should be checked. This variable will be tested for equality against all the values specified in the subsequent case statements. If the value of the variable matches a case, the code associated with that case runs.

You can have as many cases as you want in a switch statement, as long as

1. the variable in the switch matches the data type of the cases
2. the cases are constant expressions that can be interpreted as integer values

Each case statement starts with the `case` keyword, followed by the value that the switch variable should be compared to, followed by a colon. The order of cases in a switch statement does not matter. This is different from an if-else chain, where the order of checks may influence the program's performance (note: for if-else chains, it would be preferable to list the most likely branches first).

When the code associated with a case runs, the case will execute until a `break` statement is reached. When a `break` statement is reached, the switch statement terminates and the program continues to the code after the switch. However, if no `break` is encountered at the end of a case, the program *falls through* to subsequent cases until a `break` is found — in other words, if the value of the switch variable matches a case, the program will execute that case, as well *all cases* that come after the match until a `break` statement is reached.

Remark: C++17 introduced the `[[fallthrough]]` attribute, which can be used to explicitly indicate that a fallthrough is intentional. This is because fallthroughs are typically not expected behavior, and they are often caused by forgetting the `break` statement at the end of a case. As a result, certain compilers will issue a warning if a fallthrough case is encountered in a switch statement. The inclusion of `[[fallthrough]]` suppresses this error, and it is also good style if fallthrough behavior is actually intended.

A switch statement may also have an optional `default` case at the very end. This case covers the condition where the switch variable being checked does not match with any of the provided cases. Since the `default` case is placed at the very end, there is no need for a `break` statement because there are no cases that follow it (and thus no risk of unintended fallthrough).

※ 3.2.2 *Enums*

One disadvantage of a switch statement is that they can only be applied on variables that can be interpreted as an integer. As a result, something like this would *not* work, since it is impossible to interpret a `std::string` object as an integer:

```
1   std::string day_of_week;
2   std::cin >> day_of_week;
3   switch (day_of_week)
4   {
5     case "Monday":
6     {
7       std::cout << "It's Monday!\n";
8       break;
9     }
10    case "Tuesday":
11    {
12      std::cout << "It's Tuesday!\n";
13      break;
14    }
      ...

35    case "Sunday":
36    {
37      std::cout << "It's Sunday!\n";
38      break;
39    }
40    default:
41    {
42      std::cout << "Invalid day!\n";
43    }
44  }
```

One way to get around this is to use an integer to represent each string. For example, we could let 1 represent Monday, 2 represent Tuesday, …, and 7 represent Sunday:

```
1   int32_t day_of_week;
2   std::cin >> day_of_week;
3   switch (day_of_week)
4   {
5     case 1:
6     {
7       std::cout << "It's Monday!\n";
8       break;
9     }
10    case 2:
11    {
12      std::cout << "It's Tuesday!\n";
13      break;
14    }
      ...

35    case 7:
36    {
37      std::cout << "It's Sunday!\n";
38      break;
39    }
40    default:
41    {
42      std::cout << "Invalid day!\n";
43    }
44  }
```

However, this approach is complicated, messy, and prone to errors. For example, does the number 1 represent Monday or Sunday? Should Sunday be represented as 0 or 7? Not only is the code ambiguous, it is also unreadable. For complicated switch statements with many cases, other programmers may have no idea what the integers refer to and can easily mess something up.

A better approach would be to use something known as an **enumerated type**, or **enum**. An enum is a user-defined type that is restricted to a certain set of values, and it allows you to use a condition's "name" in the switch statement. For example, an enum would allow you to describe the switch cases using the actual names of the days of the week (Monday, Tuesday, etc.) instead of integers (1, 2, etc.). This is allowed because an enum associates each "name" with an integer that is used for the switch cases under the hood. To define an enum, use the following syntax:

```
enum class <NAME OF ENUM> { ... VALUES OF ENUM ... }
```

For example, the following would define an enum called `Day` that takes on the values of Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday:

```
enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

This allows you do things like this:

```
Day day_of_week = Day::Friday;
if (day_of_week == Day::Friday) {
  std::cout << "It's Friday!\n";
} // if
```

Here, `day_of_week` is an integer behind the scenes, and not a string. However, the readability of the program is maintained since the full name of each weekday is used instead of an arbitrary number.

What integer does each weekday actually represent? If we were to print out the value of `Day::Friday`, for example, the number 4 is printed out. Why is `Day::Friday` equal to 4 upon initialization? It turns out that the construction of an enum follows two rules:

- If the first enum value is not explicitly assigned a value, it is assigned an integer value of 0.
- For any enum that follows, if it is not explicitly assigned a value, its value is set to a value one greater than the value of the previous enum.

Thus, in the following enum definition:

```
enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

Monday gets assigned the integer 0, Tuesday gets assigned the integer 1, Wednesday gets assigned the integer 2, Thursday gets assigned the integer 3, Friday gets assigned the integer 4, Saturday gets assigned the integer 5, and Sunday gets assigned the integer 6.

However, this is customizable: you can explicitly assign the value of an enum using an initializer in its definition. For instance, if you wanted Monday to be assigned the integer 1, Tuesday to be assigned the integer 2, and so on, you can explicitly assign these values as follows:

```
enum class Day { Monday = 1, Tuesday = 2, Wednesday = 3,
                 Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7 };
```

However, via the second rule of enum initialization, only Monday needs to be explicitly set to 1 if you want the above configuration. This is because each unassigned enum is assigned the value of the previous enum plus one (so Tuesday gets automatically assigned to $1 + 1 = 2$, Wednesday to $2 + 1 = 3$, etc.). The definition below does the same thing as the one above:

```
enum class Day { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

It is possible for two enum values can be assigned to the same integer, and they can be compared just like integers.

You can also change the underlying integer type of an enum. For instance, the following would store the days of the week as integers of type `int8_t` (and would take up 1 byte of memory instead of 4 bytes):

```
enum class Day : int8_t { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

Since an object of type `char` is actually a 1-byte number behind the scenes, you can assign each enum to a character and use them in the cases of switch statements as well.

```
enum class Day : char { Monday = 'm', Tuesday = 't', Wednesday = 'w',
                        Thursday = 'h', Friday = 'f', Saturday = 's', Sunday = 'u' };
```

Putting this all together, you can use enumerated types to build a cleaner, more readable switch statement, such as the one below.

```
enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
Day day_of_week = ...;
switch (day_of_week)
{
  case Day::Monday:
  {
    std::cout << "It's Monday!\n";
    break;
  }
  case Day::Tuesday:
  {
    std::cout << "It's Tuesday!\n";
    break;
  }
  ...
  case Day::Sunday:
  {
    std::cout << "It's Sunday!\n";
    break;
  }
  default:
  {
    std::cout << "Invalid day!\n";
  }
}
```

As mentioned, switch cases allow individual cases to fall through if a `break` statement is not specified in a case. This can be useful if you want to group different cases together, such as with the following example.

```
1  enum class Suit { Diamonds, Hearts, Clubs, Spades };
2  Suit current_suit = ...;  // assign value to suit
3  switch (current_suit)
4  {
5    case Suit::Diamonds:
6    {
7      [[fallthrough]];
8    }
9    case Suit::Hearts:
10   {
11     std::cout << "You got a red card!\n";
12     break;
13   }
14   case Suit::Clubs:
15   {
16     [[fallthrough]];
17   }
18   case Suit::Spades:
19   {
20     std::cout << "You got a black card!\n";
21     break;
22   }
23   default:
24   {
25     std::cout << "I don't know what your card is.\n";
26   }
27 }
```

Here, if the suit ends up being `Diamonds`, the code associated with the `Diamonds` case runs. Since there is no `break` statement in the `Diamonds` case, the program falls through to the next case and runs the code associated with the `Hearts` case as well (printing "You got a red card!"). Only then does the code encounter a `break` statement, so it exits the switch without running the code for `Clubs` or `Spades`.

Here's another example of a switch statement, which gives a prognosis of the current day's weather conditions when given one of four possible weather values.

```
1  enum class Weather { partly_cloudy, sunny, rainy, overcast };
2  Weather current_weather = ...;  // assign value to current weather
3  switch (current_weather)
4  {
5    case Weather::partly_cloudy:
6    {
7      std::cout << "There will be clouds in the sky.\n";
8      [[fallthrough]];
9    }
10   case Weather::sunny:
11   {
12     std::cout << "The sun will be visible in the sky.\n";
13     break;
14   }
15   case Weather::rainy:
16   {
17     std::cout << "It will rain.\n";
18     [[fallthrough]];
19   }
20   case Weather::overcast:
21   {
22     std::cout << "The sun will likely not be visible in the sky.\n";
23     break;
24   }
25   default:
26   {
27     std::cout << "Cannot identify weather\n";
28   }
29 }
```

If `current_weather` were set to `partly_cloudy`, the switch statement would run code from the `partly_cloudy` case (line 5) to the first `break` encountered (line 12). Thus, the following would be printed:

```
There will be clouds in the sky.
The sun will be visible in the sky.
```

If `current_weather` were set to `sunny`, the switch statement would run code from line 9 to the first `break` encountered, or line 12. The following would be printed:

```
The sun will be visible in the sky.
```

If `current_weather` were set to `rainy`, the switch statement would run code from line 14 to the first `break` encountered, or line 21. The following would be printed:

```
It will rain.
The sun will likely not be visible in the sky.
```

If `current_weather` were set to `overcast`, the switch statement would run code from line 18 to the first `break` encountered, or line 21. The following would be printed:

```
The sun will likely not be visible in the sky.
```

Lastly, it should be mentioned that curly braces are not always necessary for each case. The following two versions of code do the same thing and would both work:

```
1  case Day::Monday:
2  {
3    std::cout << "It's Monday!\n";
4    break;
5  }
6  case Day::Tuesday:
7    ...
```

```
1  case Day::Monday:
2    std::cout << "It's Monday!\n";
3    break;
4  case Day::Tuesday:
5    ...
```

However, the version without the curly braces does *not* allow you to initialize variables within the code for a case! Of the two implementations below, only the one of the left would work — the one on the right would throw an error.

```
1  case Day::Monday:
2  {
3    int num_classes = 5; // OK
4    std::cout << "It's Monday!\n";
5    break;
6  }
7  case Day::Tuesday:
8    ...
```

```
1  case Day::Monday:
2    int num_classes = 5; // ERROR
3    std::cout << "It's Monday!\n";
4    break;
5  case Day::Tuesday:
6    ...
```

Remark: There is a difference between an `enum class` and a traditional `enum`. Technically, both of the following approaches can be used to define a brand new enum:

```
enum class <NAME OF ENUM> { ... VALUES OF ENUM ... }
enum <NAME OF ENUM> { .. VALUES OF ENUM ... }
```

However, you should always include the `class` keyword when creating a new enum. This is because enum classes address several bug prone issues that traditional enums do not:

1. Traditional enums implicitly convert to integers, which may not be desired by the programmer.

The compiler is not the programmer, so it should not be making decisions on the programmer's behalf (similar to why the `explicit` keyword is necessary). Thus, you would want to prevent implicit conversions between an enum type and its underlying integer value, which is a protection that only enum classes provide.

```
1  enum Color { Red, Green, Blue, Yellow };        // traditional enum
2
3  int main() {
4    Color enum_value = Color::Blue;
5    if (enum_value == 2) {                        // implicit conversion, bad!
6      std::cout << "blue" << std::endl;
7    } // if
8  } // main()
```

```
1  enum class Color { Red, Green, Blue, Yellow };  // enum class
2
3  int main() {
4    Color enum_value = Color::Blue;
5    if (enum_value == 2) {                        // compiler error, good!
6      std::cout << "blue" << std::endl;
7    } // if
8  } // main()
```

```
1  enum class Color { Red, Green, Blue, Yellow };  // enum class
2
3  int main() {
4    Color enum_value = Color::Blue;
5    if (static_cast<int32_t>(enum_value) == 2) {  // okay, because explicitly cast to integer
6      std::cout << "blue" << std::endl;
7    } // if
8  } // main()
```

2. Traditional enums export their names to the surrounding scope, which could cause clashes with other variable names.

Unlike enum classes, the enum names used in a traditional enum cannot be used in the same scope as the enum, or in any other enum. This is not ideal behavior, as shown below:

```
enum Color { Red, Green, Blue, Yellow };     // traditional enum
enum Color2 { Red, Orange, Pink, Purple };   // not allowed, "Red" already used by 1st Color enum

int main() {
  enum Color3 { Brown, Indigo, Cyan, Teal };
  std::string Brown = "brown";               // not allowed, "Brown" already used by enum
} // main()
```

```
enum class Color { Red, Green, Blue, Yellow };     // enum class
enum class Color2 { Red, Orange, Pink, Purple };   // okay

int main() {
  enum class Color3 { Brown, Indigo, Cyan, Teal };
  std::string Brown = "brown";                     // okay
} // main()
```

3.3 Getopt Long

※ 3.3.1 *Command Line Options*

In the first section of this chapter, we introduced `argc` and `argv`, which can be used to retrieve information from the command line. However, once we get these command line options, we need a way to parse them and adjust our program's behavior based on which options were entered. Let's return to our original command:

```
./game --level 25 --difficulty 10 --multiplayer
```

This command tells us that we should run level 25 of the game, with difficulty set to 10, and multiplayer mode on. A tempting (but incorrect) approach would be to retrieve the level number from `argv[2]`, the difficulty from `argv[4]`, and check `argc` to determine whether multiplayer mode is turned on. However, this would fail if the options were given in a different order!

```
./game --multiplayer --level 25 --difficulty 10
```

A better approach would be to go through the `argv` array and check for each of the options. If `--level` is specified, check the following index for the number of the level. If `--multiplayer` is specified, turn on multiplayer. This approach is more likely to produce the desired behavior, but it also has its faults. What if the user forgot to input the level number after the `--level` option? What if the `--multiplayer` option could be followed by a number that represents the number of players?

To further complicate things, command lines typically support short form options as well. A *short form option* is an abbreviated form of an option, and it is preceded with a single dash rather than two dashes (the fully typed out options we currently have, such as `--level` and `--difficulty`, are known as *long form options*). For example, the short form option of `--difficulty` may be `-d`, and the short form option of `--multiplayer` may be `-m`. In other words, the command

```
./game --level 25 --difficulty 10 --multiplayer
```

can be rewritten like this for the same outcome:

```
./game -l 25 -d 10 -m
```

In most programs, short forms and long forms can be mixed together, and short forms may be combined. As long as the difficulty follows the long form `--difficulty` or the short form `-d`, and the level follows the long form `--level` or the short form `-l`, the command should be valid. For instance, the following commands should all exhibit the same behavior:

```
./game --level 25 --difficulty 10 --multiplayer
          ./game -l 25 -d 10 -m
      ./game --difficulty 10 -m -l 25
         ./game -md 10 --level 25
```

As you can see here, command line parsing can be get quite complicated if you are forced to do so by hand! While it might be feasible to account for all possible combinations and orderings of options when the number of possible options are low, this becomes much harder as the number of options increases. The total number of possible orderings also makes it difficult to handle errors in the command line.

※ 3.3.2 *Using Getopt Long*

To simplify the process of parsing command line options, we can use the GNU's `getopt_long()` function, found in the `<getopt.h>` library. A possible implementation for the `game` example is shown below:

```
#include <iostream>
#include <getopt.h>
using namespace std;

// sets the difficulty of the game
set_game_difficulty(int32_t difficulty);
// sets the level of the game
set_game_level(int32_t level);
// turns multiplayer mode on
set_multiplayer();

int main(int argc, char** argv) {
  int opt;
  int opt_index = 0;
  static struct option long_opts[] = {
    { "difficulty", required_argument, nullptr, 'd' },
    { "level", required_argument, nullptr, 'l' },
    { "multiplayer", no_argument, nullptr, 'm' },
    { nullptr, 0, nullptr, '\0' }
  };

  while ((opt = getopt_long(argc, argv, "d:l:m", long_opts, &opt_index)) != -1) {
    switch (opt)
    {
      case 'd':
      {
        // runs if 'd' or "difficulty" is specified on the command line
        std::cout << "difficulty set to " << atoi(optarg) << '\n';
        set_game_difficulty(atoi(optarg));
        break;
      }
      case 'l':
      {
        // runs if 'l' or "level" is specified on the command line
        std::cout << "level set to " << atoi(optarg) << '\n';
        set_game_level(atoi(optarg));
        break;
      }
      case 'm':
      {
        // runs if 'm' or "multiplayer" is specified on the command line
        std::cout << "multiplayer mode turned on\n";
        set_multiplayer();
        break;
      }
      default:
      {
        std::cout << "unrecognized option" << std::endl;
        exit(1);
      }
    } // switch
  } // while
} // main()
```

There is quite a bit of information to break down here. In the following pages, we will break this code down into segments and explain the details of `getopt_long()` one segment at a time. To start off, let's look at the `long_opts[]` array:

```
static struct option long_opts[] = {
  { "level", required_argument, nullptr, 'l' },
  { "difficulty", required_argument, nullptr, 'd' },
  { "multiplayer", no_argument, nullptr, 'm' },
  { nullptr, 0, nullptr, '\0' }
};
```

The `long_opts[]` array is an array of `option` structs, where each `option` is defined as follows:

```
struct option {
  const char* name;
  int has_arg;
  int* flag;
  int val;
};
```

Each valid option that the user can enter into the command line (e.g., `-d`, `-l`, etc.) is treated as a separate `option` object. As shown above, each `option` has four member variables. Consider the first `option` struct, defined on line 16.

```
{ "difficulty", required_argument, nullptr, 'd' }
```

The `name` member variable stores the long version of the option as a C-string. For example, this variable would store the word "difficulty" as a C-string for the `--difficulty` command line option (as shown above).

```
{ "difficulty", required_argument, nullptr, 'd' }
```

The `has_arg` member variable stores an integer that represents whether the option needs to be followed by an argument. There are three types of options: ones that have a required argument, ones that have no argument, and ones that have an optional argument.

- An option with a *required argument* must be followed by an additional argument if it is included in the command line. For example, `--level` would be an option with a required argument, since it must be followed by the level you want to go to (e.g., `--level 25`).
- An option with *no argument* must be run on its own if specified on the command line. For example, `--multiplayer` requires no argument, since there is no need to include anything along with it.
- An option with an *optional argument* may or may not be followed by an additional argument if specified on the command line. Unlike options with required arguments, this additional argument is *not necessary* for the command to be valid. For example, if the `--multiplayer` option defaults to 2 players, but the number of players could be customized with an additional argument (e.g., `--multiplayer 3` for 3 players), then this option would have an optional argument. For optional arguments, it is important to note that *the argument following the option is the thing that is optional, and not the actual option itself!*

The value of `has_arg` can take on three possible values, `no_argument`, `required_argument`, or `optional_argument`. All three are enums that correspond to the integers 0, 1, and 2, respectively. *Make sure you specify the correct option type for each option, as mistakes here can be difficult to track down!* Because these are enums, you can simply type out the word `required_argument` or `no_argument` in the initializer list, as shown in the provided code.

```
{ "difficulty", required_argument, nullptr, 'd' }
```

Next, we have the `flag` and `val` member variables of the `option` object. There are two different behaviors that can happen, depending on the value assigned to `flag`. If you want a command line option to set the value of an integer variable, you can set `flag` to the address of this integer variable and `val` to the value you want to set the integer to. Consider the following example:

```
1  static int num_players = 1;
2  int main(int argc, char** argv) {
3    ...
4    static struct option long_opts[] = {
5      { "multiplayer", no_argument, &num_players, 2 },
6      ...
7    };
8    ...
9  }
```

Here, the number of players is initially set to 1. However, because the value of `flag` for the `--multiplayer` option was set to the address of `num_players` and `val` was set to `2`, the value of `num_players` will be changed to `2` if the `--multiplayer` option is ever seen on the command line. ***You will not be using this method in EECS 281.***

In this class, you will be using the second behavior, which allows you to specify a short form for each option. To accomplish this, set the value of `flag` to `nullptr` and the value of `val` to the character that should be used as the option's short form. For example, the following indicates that the option `"difficulty"` can be represented in short form using the letter `'d'`.

```
{ "difficulty", required_argument, nullptr, 'd' }
```

The last item in the array must be an `option` with all of its members set to zero. This item tells `getopt_long()` that there are no more option choices remaining:

```
{ nullptr, 0, nullptr, '\0'}
```

You can also initialize this last `option` using `{0, 0, 0, 0}`, which does the same thing.

Now, let's look at the `while` loop after the array definition.

```
22  while ((opt = getopt_long(argc, argv, "d:l:m", long_opts, &opt_index)) != -1) {
```

Here, the `getopt_long()` function parses the command line and assigns the short form of the parameter into the `opt` variable (declared on line 13). If there are no more command line arguments to read, this function would return `-1`, and the `while` loop would terminate.

The third argument of the `getopt_long()` function is the *short options string*, which tells the function whether it should expect an additional argument after the option. In the example, we had to tell the `game` program what level we wanted to go to if we specify `--level` on the command line. `--level` in this case is our option, but the `25` that follows is an additional argument that must always follow `--level`.

In the short options string, you would use a colon to indicate whether a short option should be followed by an argument. If an option has a required argument, its short option must be followed by a colon in the short options string. If an option has no argument, its short option is followed by nothing. If an option has an optional argument, its short option must be followed by two colons in the short options string.

The example in the sample code, `"d:l:m"`, indicates that the

- The `d` (`difficulty`) option has a required argument (since `d` is followed by a colon).
- The `l` (`level`) option has a required argument (since `l` is followed by a colon).
- The `m` (`multiplayer`) option has no argument (since `m` is followed by nothing).

The order of the options in the short options string does not matter. The short option strings `"l:d:m"`, `"md:l:"`, `"ml:d:"`, and `"l:md:"` all work as alternatives. You just need to ensure that `d` and `l` are followed by a colon, since these options require an additional argument.

```
23  switch (opt)
24  {
25    case 'd':
26    {
27      // runs if 'd' or "difficulty" is specified on the command line
28      std::cout << "difficulty set to " << atoi(optarg) << '\n';
29      set_game_difficulty(atoi(optarg));
30      break;
31    }
32    case 'l':
33    {
34      // runs if 'l' or "level" is specified on the command line
35      std::cout << "level set to " << atoi(optarg) << '\n';
36      set_game_level(atoi(optarg));
37      break;
38    }
39    case 'm':
40    {
41      // runs if 'm' or "multiplayer" is specified on the command line
42      std::cout << "multiplayer mode turned on\n";
43      set_multiplayer();
44      break;
45    }
46    default:
47    {
48      std::cout << "unrecognized option" << std::endl;
49      exit(1);
50    }
51  }
```

In the body of the `while` loop, you will have a switch statement that determines what your program does when each of the options are encountered on the command line. Each case denotes the short option of a command (as shown on lines 25, 32, and 39), and the body of each case implements the actions that are taken when the option is seen.

In the case where there is an additional argument after the option, the argument is stored in a global C-string called `optarg`. For example, if `--level 25` is seen on the command line, the `25` is stored in a variable called `optarg`, which can be used in your program (an example is shown on line 28). It is important to note that `optarg` is a C-string, so you will have to convert `optarg` to the correct type before you use it.

For error handling, it may be useful to add an additional `default` case at the end of the switch statement. The code in the `default` section is run when the option encountered does not match any of the options that are defined in the `long_opts[]` array. In the example, the `default` case is included on line 46, and exits the program if an unrecognized option is provided on the command line.

Lastly, as mentioned before, the `getopt_long()` function returns `-1` if there are no more options to read from the command line. However, what happens if there are more arguments that need to be read after the final option? Consider the following command:

```
./game --multiplayer --level 25 --difficulty 10 input.txt
```

Here, we added an input file, `input.txt`, that needs to be read in by the program. However, `input.txt` is not associated with a command line option, so `getopt_long()` would return `-1` before processing this last term.

How do we know where the input file is located? In the above example, it is located at `argv[6]`, so we would want to start processing any additional arguments starting at index 6 of the `argv` array. Finding this final index though is not trivial; the command below is equally valid, and the input file for this command is located at `argv[5]`.

```
./game -ml 25 -d 10 input.txt
```

Luckily, `getopt_long()` keeps track of this information for us. There is a global variable called `optind` that stores the index of the next element in the `argv` array that needs to be processed. Once `getopt_long()` finishes processing all of the option terms, the value of `optind` can be used to determine where the remaining non-option terms begin. In the first command, `optind` would have a value of 6 after `getopt_long()` finishes running, since the first non-option term, `input.txt`, is located at `argv[6]`. In the second command, `optind` would have a value of 5 since the first non-option term is located at `argv[5]`.

In general, to check if there are non-option terms that need to be processed after running `getopt_long()`, you can compare the value of `optind` with the value of `argc`. If `(argc > optind)` evaluates to `true`, then there are additional terms that you will need to parse.

3.4 Boost Program Options (⁎)

※ 3.4.1 *Using Boost Program Options (⁎)*

If you ever do C++ development in industry, there is a good chance that you will not be using `getopt_long()` to handle command line arguments. Instead, a popular alternative is `boost::program_options` in the `boost` library. The `boost` library, however, is banned in the class, *so you will **not** be responsible for knowing this material*. That being said, this section is included because `boost::program_options` is a useful resource, and it would not hurt to be aware of its existence as a developer.

A short snippet of command line parsing code using `boost::program_options` is shown below, with the same game example covered in the previous section. You do not need to understand everything in this code for now; we will go over the details shortly.

```
1   #include <iostream>
2
3   #include <boost/program_options.hpp>
4   #include <boost/program_options/errors.hpp>
5   #include <boost/program_options/options_description.hpp>
6   #include <boost/program_options/parsers.hpp>
7   #include <boost/program_options/positional_options.hpp>
8   #include <boost/program_options/value_semantic.hpp>
9   #include <boost/program_options/variables_map.hpp>
10
11  int main(int argc, char** argv) {
12    boost::program_options::options_description command_line_options{"Program Options"};
13    command_line_options.add_options()
14      ("difficulty,d", boost::program_options::value<int32_t>(), "Game difficulty")
15      ("level,l", boost::program_options::value<int32_t>(), "Starting level")
16      ("name,n", boost::program_options::value<std::string>(), "Player name")
17      ("multiplayer,m", "Starts game in multiplayer mode")
18      ("help,h", "Prints out help message");
19
20    boost::program_options::variables_map options_map;
21    int32_t difficulty{};
22    int32_t level{};
23    std::string name;
24    bool is_multiplayer = false;
25
26    try {
27      boost::program_options::parsed_options parsed_result =
28        boost::program_options::parse_command_line(argc, argv, command_line_options);
29      boost::program_options::store(parsed_result, options_map);
30
31      if (options_map.count("help")) {
32        std::cout << command_line_options << '\n'; // prints out options in pretty format
33        return 0;
34      } // if help
35      if (options_map.count("difficulty")) {
36        difficulty = options_map["difficulty"].as<int32_t>(); // sets difficulty to provided value
37      } // if difficulty
38      if (options_map.count("level")) {
39        level = options_map["level"].as<int32_t>(); // sets level to provided value
40      } // if level
41      if (options_map.count("name")) {
42        name = options_map["name"].as<std::string>(); // sets name to provided value
43      } // if name
44      if (options_map.count("multiplayer")) {
45        is_multiplayer = true;
46      } // if multiplayer
47
48      boost::program_options::notify(options_map);
49    } // try
50    catch (boost::program_options::error& e) {
51      std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
52      // error handling here
53    } // catch
54  } // main()
```

The `boost::program_options::options_description` class (created on line 12) can be used to register the command line options that your program can handle. Lines 14-18 register the command line options of `difficulty`, `level`, and `multiplayer` (like in the example at the end of section 3.3), as well as an option to print a help message and specify a player's name.

On line 13, the `add_options()` member of the `options_description` class can be used to define the available options for a program. If you want to specify a short option for a command line option, add a comma and the letter you want to use after the long name of the option (i.e., `"difficulty,d"` to allow `--difficulty` and `-d` to be used interchangeably). A short option must only consist of one letter.

Notice that some of the options listed above involve three parameters (`difficulty` and `level`), while others involve two (`multiplayer` and `help`). For the latter two, this means that no argument can be set to each of these flags (i.e., the `no_argument` case with `getopt_long()`). On the other hand, options that require an argument can be specified using `boost::program_options::value<>` with the expected type for the argument (as shown with `difficulty` and `level` on lines 14 and 15, which both expect an argument of type `int32_t`).

On line 12, we created an `boost::program_options::options_description` instance using the string `"Program Options"` in the constructor. This string is known as a *caption*, and it is printed out if you try to print the contents of an `options_description` object (which is done on line 32). The output from line 32 is shown below:

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --help
```

Output:

```
Program Options:
  -d [ --difficulty ] arg     Game difficulty
  -l [ --level ] arg          Starting level
  -n [ --name ] arg           Player name
  -m [ --multiplayer ]        Starts game in multiplayer mode
  -h [ --help ]               Prints out help message
```

The caption is optional, and you can omit it while constructing an `options_description` object. If no caption is provided, printing out the `options_description` would only print out the list of valid commands, as shown:

```
11  int main(int argc, char** argv) {
12    boost::program_options::options_description command_line_options; // no caption
13    command_line_options.add_options()
14      ("difficulty,d", boost::program_options::value<int32_t>(), "Game difficulty")
15      ("level,l", boost::program_options::value<int32_t>(), "Starting level")
16      ("name,n", boost::program_options::value<std::string>(), "Player name")
17      ("multiplayer,m", "Starts game in multiplayer mode")
18      ("help,h", "Prints out help message");
```

Output:

```
-d [ --difficulty ] arg    Game difficulty
-l [ --level ] arg         Starting level
-n [ --name ] arg          Player name
-m [ --multiplayer ]       Starts game in multiplayer mode
-h [ --help ]              Prints out help message
```

After you are done specifying which arguments your program accepts, you will need to parse the arguments that are actually provided on the command line. This is done using a `boost::program_options::variables_map`, which stores the options that are supplied on the command line with their associated values (similar to `optarg` in the case of `getopt_long()`). An example is shown on lines 27-29:

```
27  boost::program_options::parsed_options parsed_result =
28    boost::program_options::parse_command_line(argc, argv, command_line_options);
29  boost::program_options::store(parsed_result, options_map);
```

The `parse_command_line()` method takes in `argc`, `argv`, and an `options_description` object that holds the list of valid command line arguments. Then, the return value of this method is passed into the `store()` method, which stores the parsed results in a provided `variables_map` (in this case, the options map that was initialized on line 20).

After line 29, we have a series of `if` checks that determine whether an option was provided on the command line. To access the argument of a provided option (i.e., the `optarg` value in the context of `getopt_long()`), we can use square brackets on our `options_map` in the format shown on lines 36, 39, and 42.

```
31  if (options_map.count("help")) {
32    std::cout << command_line_options << '\n'; // prints out options in pretty format
33    return 0;
34  } // if help
35  if (options_map.count("difficulty")) {
36    difficulty = options_map["difficulty"].as<int32_t>(); // sets difficulty to provided value
37  } // if difficulty
38  if (options_map.count("level")) {
39    level = options_map["level"].as<int32_t>(); // sets level to provided value
40  } // if level
41  if (options_map.count("name")) {
42    name = options_map["name"].as<std::string>(); // sets name to provided value
43  } // if name
44  if (options_map.count("multiplayer")) {
45    is_multiplayer = true;
46  } // if multiplayer
```

Previously, we initialized variables on lines 21-24 and assigned them while looping through our `options_map` on lines 31-46. However, this is not the only way to assign variables to values provided on the command line. You can also perform an assignment by passing a pointer to the variable you want to assign into the value argument of the initial `options_description` object. This is shown below on lines 19-21:

```
1  #include <iostream>
2
3  #include <boost/program_options.hpp>
4  #include <boost/program_options/errors.hpp>
5  #include <boost/program_options/options_description.hpp>
6  #include <boost/program_options/parsers.hpp>
7  #include <boost/program_options/positional_options.hpp>
8  #include <boost/program_options/value_semantic.hpp>
9  #include <boost/program_options/variables_map.hpp>
10
11 int main(int argc, char** argv) {
12   int32_t difficulty{};
13   int32_t level{};
14   std::string name;
15   bool is_multiplayer = false;
16
17   boost::program_options::options_description command_line_options{"Program Options"};
18   command_line_options.add_options()
19     ("difficulty,d", boost::program_options::value<int32_t>(&difficulty), "Game difficulty")
20     ("level,l", boost::program_options::value<int32_t>(&level), "Starting level")
21     ("name,n", boost::program_options::value<std::string>(&name), "Player name")
22     ("multiplayer,m", "Starts game in multiplayer mode")
23     ("help,h", "Prints out help message");
24
25   boost::program_options::variables_map options_map;
26
27   try {
28     boost::program_options::parsed_options parsed_result =
29       boost::program_options::parse_command_line(argc, argv, command_line_options);
30     boost::program_options::store(parsed_result, options_map);
31
32     if (options_map.count("help")) {
33       std::cout << command_line_options << '\n'; // prints out options in pretty format
34       return 0;
35     } // if help
36     if (options_map.count("multiplayer")) {
37       is_multiplayer = true;
38     } // if multiplayer
39
40     boost::program_options::notify(options_map);
41
42     std::cout << "Difficulty: " << difficulty << '\n';
43     std::cout << "Level: " << level << '\n';
44     std::cout << "Name: " << name << '\n';
45   } // try
46   catch (boost::program_options::error& e) {
47     std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
48     // error handling here
49   } // catch
50 } // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --difficulty 10 --level 25 --name Dario
```

Output:

```
Difficulty: 10
Level: 25
Name: Dario
```

※ 3.4.2 *Switch Arguments* (⁎)

We can also use the previous process to assign the `is_multiplayer` Boolean. This can be done by converting the `multiplayer` option into a *switch argument*. A switch argument is a command line argument that takes in no value, but can be used to switch on or off some functionality if specified. The format for specifying a switch argument is shown below on line 22:

```
17  boost::program_options::options_description command_line_options{"Program Options"};
18  command_line_options.add_options()
19    ("difficulty,d", boost::program_options::value<int32_t>(&difficulty), "Game difficulty")
20    ("level,l", boost::program_options::value<int32_t>(&level), "Starting level")
21    ("name,n", boost::program_options::value<std::string>(&name), "Player name")
22    ("multiplayer,m", boost::program_options::bool_switch(&is_multiplayer)->default_value(false),
23     "Starts game in multiplayer mode")
24    ("help,h", "Prints out help message");
```

Adding this change to our previous code, we would get the following:

```
1   #include <iostream>
2
3   #include <boost/program_options.hpp>
4   #include <boost/program_options/errors.hpp>
5   #include <boost/program_options/options_description.hpp>
6   #include <boost/program_options/parsers.hpp>
7   #include <boost/program_options/positional_options.hpp>
8   #include <boost/program_options/value_semantic.hpp>
9   #include <boost/program_options/variables_map.hpp>
10
11  int main(int argc, char** argv) {
12    int32_t difficulty{};
13    int32_t level{};
14    std::string name;
15    bool is_multiplayer;
16
17    boost::program_options::options_description command_line_options{"Program Options"};
18    command_line_options.add_options()
19      ("difficulty,d", boost::program_options::value<int32_t>(&difficulty), "Game difficulty")
20      ("level,l", boost::program_options::value<int32_t>(&level), "Starting level")
21      ("name,n", boost::program_options::value<std::string>(&name), "Player name")
22      ("multiplayer,m", boost::program_options::bool_switch(&is_multiplayer)->default_value(false),
23       "Starts game in multiplayer mode")
24      ("help,h", "Prints out help message");
25
26    boost::program_options::variables_map options_map;
27
28    try {
29      boost::program_options::parsed_options parsed_result =
30        boost::program_options::parse_command_line(argc, argv, command_line_options);
31      boost::program_options::store(parsed_result, options_map);
32
33      if (options_map.count("help")) {
34        std::cout << command_line_options << '\n'; // prints out options in pretty format
35        return 0;
36      } // if
37
38      boost::program_options::notify(options_map);
39
40      std::cout << "Difficulty: " << difficulty << '\n';
41      std::cout << "Level: " << level << '\n';
42      std::cout << "Name: " << name << '\n';
43      std::cout << "Multiplayer: " << std::boolalpha << is_multiplayer << '\n';
44    } // try
45    catch (boost::program_options::error& e) {
46      std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
47      // error handling here
48    } // catch
49  } // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --difficulty 10 --level 25 --name Dario --multiplayer
```

Output:

```
Difficulty: 10
Level: 25
Name: Dario
Multiplayer: true
```

※ 3.4.3 *Notifiers* (✻)

Notifiers are an additional feature that you can use when parsing program options. A notifier allows you to call a function whenever you encounter a specific command line option. To use a notifier, add the `notifier()` method to the value type of an option's description and pass in the function you want to invoke when that option is encountered. For example, the following prints out `"Hello <NAME>!"` when `<NAME>` is passed into the command line with the `--name` option:

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

void greet_user(const std::string& name) {
  std::cout << "Hello " << name << "!\n";
} // greet_user()

int main(int argc, char** argv) {
  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("name,n", boost::program_options::value<std::string>()->notifier(greet_user), "Player name");

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::parse_command_line(argc, argv, command_line_options);
    boost::program_options::store(parsed_result, options_map);

    boost::program_options::notify(options_map);
  } // try
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  } // catch
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --name Dario
```

Output:

```
Hello Dario!
```

Notifiers make it easy to use lambda functions to set the values of variables. The following reads in configs in the command line and stores them in a configuration class (you do not need to know what lambda functions are for now, but we will cover them in chapter 11).

```
1  #include <iostream>
2
3  #include <boost/program_options.hpp>
4  #include <boost/program_options/errors.hpp>
5  #include <boost/program_options/options_description.hpp>
6  #include <boost/program_options/parsers.hpp>
7  #include <boost/program_options/positional_options.hpp>
8  #include <boost/program_options/value_semantic.hpp>
9  #include <boost/program_options/variables_map.hpp>
10
11 class GameConfig {
12   std::string name;
13   int32_t difficulty{};
14 public:
15   std::string get_name() {
16     return name;
17   } // get_name()
18
19   void set_name(const std::string& name_in) {
20     name = name_in;
21   } // set_name()
22
23   int32_t get_difficulty() {
24     return difficulty;
25   } // get_difficulty()
26
27   void set_difficulty(int32_t difficulty_in) {
28     difficulty = difficulty_in;
29   } // set_difficulty()
30 };
31
32 int main(int argc, char** argv) {
33   GameConfig config;
34   boost::program_options::options_description command_line_options{"Program Options"};
35   command_line_options.add_options()
36     ("difficulty,d", boost::program_options::value<int32_t>()->notifier(
37       [&config] (int32_t provided_difficulty) { config.set_difficulty(provided_difficulty); }
38     ), "Game difficulty")
39     ("name,n", boost::program_options::value<std::string>()->notifier(
40       [&config] (const std::string& provided_name) { config.set_name(provided_name); }
41     ), "Player name");
42
43   boost::program_options::variables_map options_map;
44
45   try {
46     boost::program_options::parsed_options parsed_result =
47       boost::program_options::parse_command_line(argc, argv, command_line_options);
48     boost::program_options::store(parsed_result, options_map);
49
50     boost::program_options::notify(options_map);
51
52     std::cout << "Name: " << config.get_name() << '\n';
53     std::cout << "Difficulty: " << config.get_difficulty() << '\n';
54   } // try
55   catch (boost::program_options::error& e) {
56     std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
57     // error handling here
58   } // catch
59 } // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --name Dario --difficulty 10
```

Output:

```
Name: Dario
Difficulty: 10
```

In this code, lines 37 and 40 are lambda functions that initialize the contents of a `GameConfig` object directly from the program arguments.

If you paid close attention to the example code provided so far in this section, you may noticed that there is always a call to `notify()` on the constructed `options_map` (line 50 of the above code). *This call is very important to have after you are done parsing the command line arguments!* By calling `notify()`, you trigger any actions that are required after the value of an option is determined. This includes notifier functions like in the example above, as well as any variable initializations within the `value<>` object of the options description (i.e., if `value<int32_t>(&difficulty)` is specified, `difficulty` is only assigned to its command line value *after* `notify()` is invoked).

※ 3.4.4 *Required Arguments* (✻)

If you want to require an argument to be specified on the command line, you can use the `required()` method, as shown. In this code, the user must specify a difficulty, level, and name. Failure to specify any of these arguments would result in an error, which is issued when the `notify()` function is invoked.

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

int main(int argc, char** argv) {
  int32_t difficulty{};
  int32_t level{};
  std::string name;

  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("difficulty,d", boost::program_options::value<int32_t>(&difficulty)->required(),
     "Game difficulty")
    ("level,l", boost::program_options::value<int32_t>(&level)->required(), "Starting level")
    ("name,n", boost::program_options::value<std::string>(&name)->required(), "Player name")
    ("help,h", "Prints out help message");

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::parse_command_line(argc, argv, command_line_options);
    boost::program_options::store(parsed_result, options_map);

    if (options_map.count("help")) {
      std::cout << command_line_options << '\n';
      return 0;
    } // if

    boost::program_options::notify(options_map); // issues error if required arguments missing
  } // try
  catch (boost::program_options::required_option& req) {
    std::cout << "Missing required argument: " << req.get_option_name() << '\n';
    // error handling here
  } // catch
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  } // catch
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --name Dario --difficulty 10
```

Output:

```
Missing required argument: --level
```

※ 3.4.5 *Default and Implicit Values* (✻)

The `default_value()` method can be used to specify a default value if a given option is not provided on the command line. In the following code, `difficulty` has a default value of 10 if not specified, and `level` has a default value of 1 if not specified.

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

int main(int argc, char** argv) {
  int32_t difficulty{};
  int32_t level{};
  std::string name;

  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("difficulty,d", boost::program_options::value<int32_t>(&difficulty)->default_value(10),
     "Game difficulty")
    ("level,l", boost::program_options::value<int32_t>(&level)->default_value(1), "Starting level")
    ("name,n", boost::program_options::value<std::string>(&name), "Player name")
    ("help,h", "Prints out help message");

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::parse_command_line(argc, argv, command_line_options);
    boost::program_options::store(parsed_result, options_map);

    if (options_map.count("help")) {
      std::cout << command_line_options << '\n';
      return 0;
    } // if

    boost::program_options::notify(options_map);

    std::cout << "Difficulty: " << difficulty << '\n';
    std::cout << "Level: " << level << '\n';
    std::cout << "Name: " << name << '\n';
  } // try
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  } // catch
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --name Dario
```

Output:

```
Difficulty: 10
Level: 1
Name: Dario
```

Boost program options also support optional arguments, which occur if an option is specified on the command line without a value (for example, `--difficulty` or `-d` without a number after it). If you want to set a value for an option that accepts a value but is not supplied one, you can use the `implicit_value()` method. In the following code, if `difficulty` is specified on the command line without a value, then it is implicitly set to value of 15. Similarly, if `level` is specified on the command line without a value, it is implicitly set to value of 5.

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

int main(int argc, char** argv) {
  int32_t difficulty{};
  int32_t level{};
  std::string name;

  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("difficulty,d", boost::program_options::value<int32_t>(&difficulty)->implicit_value(15),
     "Game difficulty")
    ("level,l", boost::program_options::value<int32_t>(&level)->implicit_value(5), "Starting level")
    ("name,n", boost::program_options::value<std::string>(&name), "Player name")
    ("help,h", "Prints out help message");

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::parse_command_line(argc, argv, command_line_options);
    boost::program_options::store(parsed_result, options_map);

    if (options_map.count("help")) {
      std::cout << command_line_options << '\n';
      return 0;
    } // if

    boost::program_options::notify(options_map);

    std::cout << "Difficulty: " << difficulty << '\n';
    std::cout << "Level: " << level << '\n';
    std::cout << "Name: " << name << '\n';
  } // try
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  } // catch
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --name Dario --difficulty --level
```

Output:

```
Difficulty: 15
Level: 5
Name: Dario
```

You can also chain multiple methods together. For instance, the following option description indicates that `difficulty` is set to 10 if it is not specified on the command line at all, and 15 if it is specified but without a value.

```
(
  "difficulty,d",
  boost::program_options::value<int32_t>()->default_value(10)->implicit_value(15),
  "Game difficulty"
)
```

※ 3.4.6 *Multitoken and Composing Arguments (⁂)*

The `multitoken()` method can be used to specify a command line option that can take in multiple arguments. For instance, suppose you wanted to specify a list of levels that you want to play for your game. One potential approach for doing this is to pass in all the levels after the `--level` option, as shown:

```
./game --level 3 10 12 19
```

To handle this using `boost::program_options`, you can use the `multitoken()` method on the value type when defining your command line options. When parsing an option of this type, the corresponding arguments are returned in the form of a `std::vector<>`.

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

int main(int argc, char** argv) {
  std::vector<int32_t> levels_to_play;

  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("level,l", boost::program_options::value<std::vector<int32_t>>(&levels_to_play)->multitoken(),
     "Levels to play")
    ("help,h", "Prints out help message");

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::parse_command_line(argc, argv, command_line_options);
    boost::program_options::store(parsed_result, options_map);

    if (options_map.count("help")) {
      std::cout << command_line_options << '\n';
      return 0;
    } // if

    boost::program_options::notify(options_map);

    for (int32_t level : levels_to_play) {
      std::cout << level << " ";
    } // for level
  } // try
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  }
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --level 3 10 12 19
```

Output:

```
3 10 12 19
```

Another option would be to allow an option to be specified more than once:

```
./game --level 3 --level 10 --level 12 --level 19
```

To handle this type of option, you can use the `composing()` method, which also parses all of the option's arguments into a vector. If we replace lines 16 and 17 of the previous code to use `composing()` instead of `multitoken()`, we would get the exact same output if the `--level` option was specified more than once.

```
14  boost::program_options::options_description command_line_options{"Program Options"};
15  command_line_options.add_options()
16    ("level,l", boost::program_options::value<std::vector<int32_t>>(&levels_to_play)->composing(),
17     "Levels to play")
18    ("help,h", "Prints out help message");
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --level 3 --level 10 --level 12 --level 19
```

Output:

```
3 10 12 19
```

Again, multiple attributes can be chained together, so it is perfectly valid for an option to support both multitoken and composing arguments.

```
1  (
2    "level,l",
3    boost::program_options::value<std::vector<int32_t>>()->multitoken()->composing(),
4    "Levels to play"
5  )
```

Lastly, the `zero_tokens()` method can be used to specify that no arguments need to be included with a multitoken option (for example, if the user wanted to play no levels at all, they could specify the `--level` option with no arguments if `zero_tokens()` is used).

※ 3.4.7 *Positional Arguments* (⁂)

Positional arguments are options on the command line that are not associated with a name. For example, consider the following command:

```
./game input1.txt input2.txt input3.txt
```

The input files at the end of the command line are positional arguments. To handle these, you should first add an entry (with a name) into the `options_description` object, much like any other argument type (you still need a name, since this is how positional arguments are categorized into groups). The type of the value should be a vector of the positional arguments that you want to read in. An example is shown below:

```
1  boost::program_options::options_description command_line_options;
2  command_line_options.add_options()
3    ("input-files", boost::program_options::value<std::vector<std::string>>(), "Input files")
```

However, this option type must also be added to an object of type `boost::program_options::position_options_description` to specify that it is a positional argument. This is shown below:

```
1  boost::program_options::positional_options_description positional_options;
2  positional_options.add("input-files", -1);
```

The `-1` specifies that an unlimited number of options on the command line should be considered as input files. You can change this value to adjust the number of positional arguments that you want to group within a given category. For instance, calling `positional_options.add("input-files", 5)` would indicate to the program that the next five positional arguments should be considered as input files, but not anything else after it. This allows you to easily group positional arguments into individual categories:

```
1  boost::program_options::positional_options_description positional_options;
2  positional_options.add("player-names", 2); // two player names
3  positional_options.add("input-files", -1); // rest of positional arguments are input files
```

After specifying the positional arguments, you should include them while parsing the command line as shown. Note that we are using the more advanced `command_line_parser()` method to parse our options instead of `parse_command_line()`, which we used before. This is because `parse_command_line()` is a simplified method that does not support positional arguments.

```
1  boost::program_options::variables_map options_map;
2  boost::program_options::parsed_options parsed_result =
3    boost::program_options::command_line_parser(argc, argv)
4      .options(command_line_options)
5      .positional(positional_options)
6      .run();
7  boost::program_options::store(parsed_result, options_map);
```

A full example using positional arguments is shown below:

```
#include <iostream>

#include <boost/program_options.hpp>
#include <boost/program_options/errors.hpp>
#include <boost/program_options/options_description.hpp>
#include <boost/program_options/parsers.hpp>
#include <boost/program_options/positional_options.hpp>
#include <boost/program_options/value_semantic.hpp>
#include <boost/program_options/variables_map.hpp>

int main(int argc, char** argv) {
  int32_t difficulty{};
  int32_t level{};
  std::vector<std::string> player_names;
  std::vector<std::string> input_files;

  boost::program_options::options_description command_line_options{"Program Options"};
  command_line_options.add_options()
    ("difficulty,d", boost::program_options::value<int32_t>(&difficulty), "Game difficulty")
    ("level,l", boost::program_options::value<int32_t>(&level), "Starting level")
    ("player-names", boost::program_options::value<std::vector<std::string>>(&player_names),
     "Player names")
    ("input-files", boost::program_options::value<std::vector<std::string>>(&input_files),
     "Input files")
    ("help,h", "Prints out help message");

  boost::program_options::positional_options_description positional_options;
  positional_options.add("player-names", 2); // two player names
  positional_options.add("input-files", -1); // rest of positional arguments are input files

  boost::program_options::variables_map options_map;

  try {
    boost::program_options::parsed_options parsed_result =
      boost::program_options::command_line_parser(argc, argv)
        .options(command_line_options)
        .positional(positional_options)
        .run();
    boost::program_options::store(parsed_result, options_map);

    if (options_map.count("help")) {
      std::cout << command_line_options << '\n';
      return 0;
    } // if

    boost::program_options::notify(options_map);

    std::cout << "Difficulty: " << difficulty << '\n';
    std::cout << "Level: " << level << '\n';

    std::cout << "Player Names:\n";
    for (const std::string& name : player_names) {
      std::cout << name << '\n';
    } // for name
    std::cout << '\n';

    std::cout << "Input Files:\n";
    for (const std::string& file : input_files) {
      std::cout << file << '\n';
    } // for file
    std::cout << '\n';
  } // try
  catch (boost::program_options::error& e) {
    std::cout << "Error while parsing command line; Message=" << e.what() << '\n';
    // error handling here
  } // catch
} // main()
```

Compile and run program:

```
g++ -std=c++1z -O3 -Wall -pedantic -pthread main.cpp -lboost_program_options -o game
./game --difficulty 10 --level 25 Dario Paoluigi input1.txt input2.txt input3.txt
```

Output:

```
Difficulty: 10
Level: 25
Player Names: Dario Paoluigi
Input Files: input1.txt input2.txt input3.txt
```