Lecture Notes for Lecture 14 of CS 5001 (Foundations of CS) for the Fall, 2018 session at the Northeastern University Silicon Valley Campus.

# *Processing Arguments*

Philip Gust,
Clinical Instructor
Department of Computer Science

# Lecture 13 Review

- A unit test tests the functionality of a group of related or inter-dependent functions in isolation, to ensure that code functions as expected during development, integration, and maintenance.
- Unit test frameworks provide a way to construct unit tests that can be run either manually or as part of an automated build, test, and release management system.
- Tests are organized into test functions that use unit test library functions to call program functions and compare the actual results against expected values
- Test functions can be further organized into suites that test all the code in a functional unit such as a data type.
- Unit tests generate results that are captured by the framework, and used to generate results in several forms, including to the console and in structured formats for further processing

# Processing Arguments

- So far, we have only used functions with a fixed number of arguments. In C and many other languages, it is possible to create functions with variable numbers of arguments.
- We have already seen an example of a function with a variable number of arguments: *printf()*, which takes a format string and additional arguments that provide values to the format string.
- We will look at how to process variable numbers of arguments and several examples of how this technique is used.
- A programs can also be passed arguments that can be used to provide data, and to customize the program behavior .
- Values can be passed when running an application from a command line like “bash” in Unix, or “cmd.com” or “powershell” in Windows., or from within an IDE such as Eclipse.
- We will look at several ways to process and use program arguments.

# Processing Arguments

Variable Length Function Arguments

- In certain situations, it is useful to write a function that can take a variable number of arguments.
- An example is the printf() function, which takes a formatting string, and additional arguments that are determined by the format specifies in the format string.
- Here are some examples of calling printf with different numbers and types of arguments.

```
printf("Hello world!");  // requires no additional arguments
printf("Today is %s\n", "Tuesday");   // requires an additional string argument
printf("Today is %s %d, %d\n", "Oct.", 9, 2018);  // requires three more arguments
```

- How can a variable number of arguments be processed by a function?

# Processing Arguments

Variable Length Function Arguments

- The C programming language provides a solution for defining a *variadic function* that can accept a variable number of parameters based on your requirements.
- You specify a function with one or more fixed parameters, and then a "…" specifier indicating zero or more arguments follow.
- The function determines the additional arguments using an explicit count, a sentinel value, or other information from the fix parameters.

# Processing Arguments

Variable Length Function Arguments

- vaMin determines minimum argument from an explicit count.

```
/**
  * Determine the minimum from explicit count
  * @param count count of list size
  * @param  first first of count additional parameters
  * @return number of chars printed
  */
int vaMin(unsigned count, int first, ... ) {
    // process first and additional count-1 parameters
}

int main() {
    vaMin(1, 3);            // one value
    vaMin(2, 3, -1);              // two values
    vaMin(3, 3, -1, -7);  // three values
}
```

# Processing Arguments

Variable Length Function Arguments

- vaStrlen determines total length of all strings in a null-terminated argument list.

```
/**
  * Determine the total length of all strings in null terminated argument list.
  * @param  first first string parameter
  * @return total length of all strings in list
  */
int vaStrlen(const char* firststr, ... ) {
    // process first and additional count-1 parameters
}

int main() {
    vaStrlen("a", NULL);              // one value
    vaStrlen("a", "b", NULL);                 // two values
    vaStrlen("a", "b", "c", NULL);   // three values
}
```

# Processing Arguments

Variable Length Function Arguments

- vaPrintf determines the number and and type of additional parameters from the format specifiers in the format string.

```
/**
 * Prints the values using the format string.
 * @param  fmtstr the format string
 * @return number of characters printed
 */
int vaPrintf(const char* fmtstr, ... ) {
    // process additional args based on format specifiers
}

int main() {
    vaPrintf("Hello world! ");  // requires no additional arguments
    vaPrintf("Today is %s\n", "Tuesday");   // requires an additional string argument
    vaPrintf("Today is %s %d, %d\n", "Oct.", 9, 2018);  // requires three more arguments
}
```

# Processing Arguments

Variable Length Function Arguments

- C provides a set of C preprocessor functions and a defined type for processing the argument list within a variadic function. These functions and the type are defined in the ‘stdarg.h’ header file.

| Type name | Description |
|---|---|
| va_list | type for iterating function arguments |

| Function name | Description |
|---|---|
| va_start | start iterating arguments with a va_list |
| va_arg | retrieve an argument |
| va_end | free a va_list |
| va_copy | copy contents of one va_list to another |

# Processing Arguments

Variable Length Function Arguments

- Example: vaMin()

```
/**
  * Determine the minimum from among list of ints.
  * @param count count of list size
  * @param  first first of count additional parameters
  * @return number of chars printed
  */
int vaMin(unsigned count, int first, ... ) {
    int minval = first;      // initial min value

    va_list args;
    va_start(args, first);
    for (int i = 1; i< count; i++) {
        int val = va_arg(args, int);
        if (val < minval) {
            minval = val;
        }
    }
    va_end(args);
    return minval;
}
```

# Processing Arguments

Variable Length Function Arguments

- Example: vaStrlen()

```
/** Determine the total length of all strings in null terminated argument list.
  * @param  first first string parameter
  * @return total length of all strings in list
  */
int vaStrlen(const char* firststr, ... ) {
    int len = 0;       // total length of strings

    va_list args;
    va_start(args, firststr);

    for (const char *nextstr = firststr;
            nextstr != NULL; nextstr = va_arg(args, const char*)) {
        len += strlen(nextstr);
    }

    // free va_list
    va_end(args);
    return len;
}
```

# Processing Arguments

Variable Length Function Arguments

- Example: vaPrintf()

```
/**
 * Prints the values using the format string.
  * @param  fmtstr the format string
  * @return number of characters printed
  */
int vaPrintf(const char* fmtstr, ... ) {
    int len = 0;

    va_list args;
    va_start(args, fmtstr);

    for (const char* curp = fmtstr; *curp != 0; curp++) {
        if (*curp != '%') {
            len += printf("%c", *curp);
        }
```

# Processing Arguments

Variable Length Function Arguments

- Example: vaPrintf()

```
            else {
                switch (*++curp) {
                case 'd':    // int
                    len += printf("%d", va_arg(args, int));
                    break;
                case 's':    // string
                    len +=printf("%s", va_arg(args, const char*));
                    break;
                default:    // unknown fmt specifier
                    len += printf("%%%c", *curp);
                    break;
                }
            }
        }
        // free va_list
        va_end(args);
        return len;
    }
```

# Processing Arguments

Program Arguments

- Arguments passed to a program at runtime can be used to customize the program behavior each time the program is run, rather than when the program was developed.
- Values can be passed when running an application from a command shell like “bash” in Unix, or “cmd.com” or “powershell” in Windows. They can also be set in the Eclipse Run or Debug configuration dialog.
- The arguments are made available to the *main()* function in C through an alternate function signature.

```
/** Main function with alternate signature.
  * @param argc number of argument strings
  * @param argv array of argument strings
  * @return completion status: EXIT_SUCCESS for success, EXIT_FAILURE for general failure
  */
int main(int argc, char *argv[argc]) {
    …
}
```

# Processing Arguments

Program Arguments

- In C, the first argument is always a string representing how the program was run. Subsequent strings represent options that the program can use. Here is how to process just the first argument.

```
/** This function echoes its first argument string.
 * @param argv array of argument strings
 * @param argc number of argument strings
 * @return status: EXIT_SUCCESS for success, EXIT_FAILURE for general failure
 */
int main(int argc, char *argv[argc]) {
    if (argc != 1) {
        printf("unexpected number of arguments: %d\n", argc);
        return EXIT_FAILURE;
    }

    printf("%s\n", argv[0]);
    return EXIT_SUCCESS;
}
```

# Processing Arguments

Program Arguments

- One way to process arguments is positionally. In this example, a string argument is required and a second numeric argument is optional.

```
/** This function echoes its argument strings a specified number of times (default: 1).
 *
 * @param argv array of argument strings
 * @param argc number of argument strings
 * @return status: EXIT_SUCCESS for success, EXIT_FAILURE for general failure
 *
 */
int main(int argc, char *argv[argc]) {
    if (argc != 2 && argc != 3) {      // check expected number of arguments
            printf("Usage: program <string> [<count>]\n");
            return EXIT_FAILURE;
    }
```

# Processing Arguments

## Program Arguments

```
    int count = 1;
    if (argc == 3) {  // process numeric count argument
        if (sscanf(argv[2], "%d", &count) != 1) {
            printf("Usage: program <string> [<count>]\n");
            return EXIT_FAILURE;
        }
    }

    // print first argument number of time specified by second argument.
    for (int i = 0; i < count; i++) {
        printf("%s\n", argv[1]);
    }
    return EXIT_SUCCESS;
}
```

# Processing Arguments

Program Arguments

- Another way to structure arguments is as named options followed by values. The advantage is that options can often be specified in any order, or left off if not used.
- The simplest ”short” option parameter is a leading ‘-’ and a single letter specifying the option, followed by a parameter used as its value.

```
/** This function echoes its argument string a specified number of times (default: 1).
 *
 * @param argv array of argument strings
 * @param argc number of argument strings
 * @return status: EXIT_SUCCESS for success, EXIT_FAILURE for general failure
 *
 */
int main(int argc, char *argv[argc]) {
    if (argc != 2 && argc != 4) { // check expected number of arguments
        printf("Usage: program [-n <count>] <string>\n");
        return EXIT_FAILURE;
    }
```

# Processing Arguments

## Program Arguments

```
    char *echostr = argv[1];
    int count = 1;
    if (argc == 4) {
        if (strcmp(argv[1], "-n") != 0) { // process numeric count argument
            printf("Usage: program [-n <count>] <string>\n");
            return EXIT_FAILURE;
        }
        if (sscanf(argv[2], "%d", &count) != 1) {
            printf("Usage: program [-n <count>] <string> \n");
            return EXIT_FAILURE;
        }
        echostr = argv[3];
    }

    // print first argument number of time specified by second argument.
    for (int i = 0; i < count; i++) {
        printf("%s\n", echostr);
    }
    return EXIT_SUCCESS;
}
```

# Processing Arguments

Program Arguments

- In another form of option processing, a multi-character “long” option name begins with ‘—’ and the value is part of the same parameter, separated by a ‘=“.

```
/** This function echoes its argument string a specified number of times (default: 1).
 *
 * @param argv array of argument strings
 * @param argc number of argument strings
 * @return status: EXIT_SUCCESS for success, EXIT_FAILURE for general failure
 *
 */
int main(int argc, char *argv[argc]) {
    if (argc != 2 && argc != 3) { // check expected number of arguments
        printf("Usage: program [--nreps=<count>] <string>\n");
        return EXIT_FAILURE;
    }
```

# Processing Arguments

## Program Arguments

```
    char *echostr = argv[1];
    int count = 1;
    if (argc == 3) {
        if (strncmp(argv[1], "—nreps=",8) != 0) { // process numeric count argument
            printf("Usage: program [--nreps=<count>] <string>\n");
            return EXIT_FAILURE;
        }
        if (sscanf(argv[1]+8, "%d", &count) != 1) {
            printf("Usage: program [--nreps=<count>] <string> \n");
            return EXIT_FAILURE;
        }
        echostr = argv[2];
    }

    // print first argument number of time specified by second argument.
    for (int i = 0; i < count; i++) {
        printf("%s\n", echostr);
    }
    return EXIT_SUCCESS;
}
```

# Processing Arguments

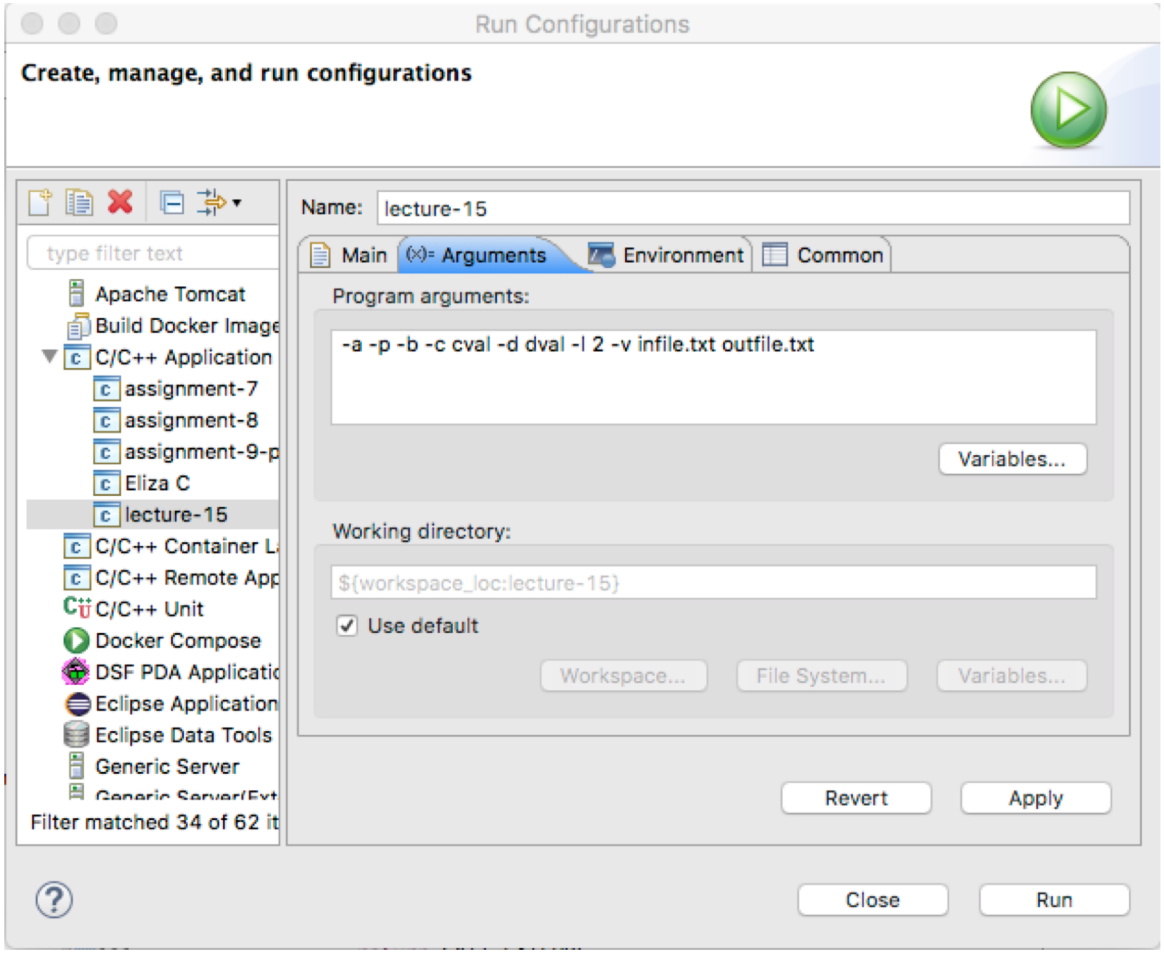
Example: Passing short program arguments

***progname*** –a –p –b [-c *cval*] [-d *dval*] [-l [*level*]] [–v] [*in* [*out*]]

- Program ***progname*** accepts 1-character argument names and optional argument values:
  - -a add
  - -p append
  - -b brief mode
  - -c *cval* create with *cval* as value
  - -d *dval* delete value *dval*
  - -l *level* logging with optional level *level*
  - -v verbose mode
  - in optional input file
  - out optional output file

# Processing Arguments

Example: Passing Arguments From Eclipse Run/Debug Configuration



# Processing Arguments

Method 1: DIY Short Argument Processing

- Loop over program arguments
- Use switch statement to process short arguments with leading ‘-’.
- Pick up required or optional value parameter values for arguments.
- Record options chosen for program to use.

```
/** Structure for holding program arguments */
typedef struct {
        bool add_opt;                   // Add values to the container
        bool append_opt;                // Append values to container
        char* delete_value;             // Delete an item from the container
        char* create_value;             // Create an item in the container
        int logging_level;              // Logging level (default=0)
        OutLevel output_level;          // Output level (normal, brief, verbose)
        char* input;                    // name of input file or '-' for stdin
        char* output;                   // name of output file, or '-' for stdout
}  AppArgs;
```

# Processing Arguments

## Method 1: DIY Short Argument Processing

```
/*
 * Initialize application arguments
 */
static AppArgs app_args = {false, false, NULL, NULL, normal, 0};

size_t optind;     // index of current opt in argv[]
for (optind = 1; optind < argc && argv[optind][0] == '-'; optind++) {
        switch (argv[optind][1]) {
        case  'a':    / /   add
              app_args.add_opt  =  true;
              printf  ("option  a\n");
              break;
        case 'b':  // brief
              app_args.output_level = brief;
              break;
```

# Processing Arguments

## Method 1: DIY Short Argument Processing

```
case 'c':  // create
        // get required create parameter
        if ((optind >= argc) || (argv[optind+1][0] == '-')) {
                // no next argument or cvalue missing (next is another flag)
                fprintf (stderr, "Option c requires an argument.\n");
                return EXIT_FAILURE;
        }
        app_args.create_value = argv[++optind];
        printf ("option c with value `%s'\n", app_args.create_value);
        break;
case 'd':  // delete
        if (optarg == (char*)NULL) {
                fprintf (stderr, "Option d requires an argument.\n");
                return EXIT_FAILURE;
        }
        // get argument as a string value
        app_args.delete_value = strdup(optarg);
        printf ("option d with value `%s'\n", app_args.delete_value);
        break;
```

# Processing Arguments

## Method 1: DIY Short Argument Processing

```
case 'l':  // logging
        if (optarg == (char*)NULL) {
                // use default level 0
                app_args.logging_level = 0;
                printf ("option l default level %d\n", app_args.logging_level);
        } else {
                // get argument as numeric value
                if (sscanf(optarg,"%d", &app_args.logging_level) == 1) {
                        printf ("option l with value %d\n", app_args.logging_level);
                } else {
                        fprintf(stderr, "Logging level '%s; not a numeric value\n", optarg);
                        return EXIT_FAILURE;
                }
        }
        break;
case 'p':  // append
        app_args.append_opt = true;
        printf ("option p\n");
        break;
```

# Processing Arguments

## Method 1: DIY Short Argument Processing

```
        case 'v':  // verbose
                app_args.output_level = verbose;
                break;
        case '?': // help option or unknown option error
                if (argv[argvind][1] == '?') { // help option
                        usage(stdout, argv[0]);
                        return EXIT_SUCCESS;
                } else {  // optarg error conditions for option stored in optopt
                        if (isprint (optopt)) {  // character is printable
                                fprintf (stderr, "Unknown option `%c'.\n", optopt);
                        } else {
                                fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
                        }
                        return EXIT_FAILURE;
                }
                break;
        default:  // should not happen, so report usage and exit
                usage(stderr, argv[0]);
                return EXIT_FAILURE;
        }
}
```

# Processing Arguments

Method 2: Getopt Short Argument Processing

- Loop over program arguments
- Call getopt() function to process rgument using option specifier string.
- Use switch statement to process short arguments with leading ‘-’.
- Function sets globals for option and argument from specifier
- Record options chosen for program to use.

```
/** Structure for holding program arguments */
typedef struct {
        bool add_opt;           // Add values to the container
        bool append_opt;        // Append values to container
        char* delete_value;     // Delete an item from the container
        char* create_value;     // Create an item in the container
        int logging_level;      // Logging level (default=0)
        OutLevel output_level;  // Output level (normal, brief, verbose)
        char* input;            // name of input file or '-' for stdin
        char* output;           // name of output file, or '-' for stdout
}  AppArgs;
```

# Processing Arguments

**Method 2: Getopt Short Argument Processing**

```
// option specifier for:
// [-a] [-b] [-c cvalue] [-d dvalue] [-l [level]] [-p] [-v] [-?]
static const char* options = ":abc:d:l:pv?";optreset = 1;

optind = 1; // reset global to allow re-processing argument list
while (true) {
    int argvind = optind;        // index of next option in argv[]

    // get next option
    int optchr = getopt (argc, argv, options);
    if (optchr == -1) {
        break;  // no more options
    }

    // argument not specified if arg from next position is another option
    if (optarg != NULL && *optarg == '-') {
        optind = argvind+1;  // reset optind to after current option in argv[]
        optarg = NULL;  // set to no argument
    }
```

# Processing Arguments

**Method 2: Getopt Short Argument Processing**

```
switch (optchr) {
case 'a':  // add
        app_args.add_opt = true;
        printf ("option a\n");
        break;
case 'b':  // brief
        app_args.output_level = brief;
        break;
case 'c':  // create
        if (optarg == NULL) {
                fprintf (stderr, "Option c requires an argument.\n");
                return EXIT_FAILURE;
        }
        app_args.create_value = strdup(optarg);
        printf ("option c with value `%s'\n", app_args.create_value);
        break;
        …
```

# Processing Arguments

**Method 2: Getopt Short Argument Processing**

```
        case '?': // help option or unknown option error
                if (argv[argvind][1] == '?') { // help option
                        usage(stdout, argv[0]);
                        return EXIT_SUCCESS;
                } else {  // optarg error conditions for option stored in optopt
                        if (isprint (optopt)) {  // character is printable
                                fprintf (stderr, "Unknown option `%c'.\n", optopt);
                        } else {
                                fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
                        }
                        return EXIT_FAILURE;
                }
                break;
        default:  // should not happen, so report usage and exit
                usage(stderr, argv[0]);
                 return EXIT_FAILURE;
        }
}
```

# Processing Arguments

Example: Passing long arguments

***--add --append --brief [–create=cvalue] [--delete=dvalue] [–logging[=level]] [--verbose] [in [out]]***

- Program ***progname*** accepts n-character argument names and optional argument values:
  - --add add
  - --append append
  - --brief brief mode
  - --create *cval* create with *cval* as value
  - --delete *dval* delete value *dval*
  - --logging *level* logging with optional level *level*
  - --verbose verbose mode
  - in optional input file
  - out optional output file

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
static struct option long_options[] = {   // long command line options
        /* These options set a flag. */
        {"verbose", no_argument,      (int*)&app_args.output_level, verbose},
        {"brief",   no_argument,      (int*)&app_args.output_level, brief},
        // The following options don't set a flag.
        // They distinguish them by their indices.
        {"add",     no_argument,      NULL, 'a'},
        {"append",  no_argument,      NULL, 'p'},
        {"delete",  required_argument, NULL, 'd'},
        {"create",  required_argument, NULL, 'c'},
        {"logging", optional_argument, NULL, 'l'},
        {"help",    required_argument, NULL, '?'},
        {0, 0, 0, 0}
};
static const char* short_options = ":abc:d:l:pv?";  // short options
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
optreset = 1;
optind = 1; // reset to allow re-processing argument list
while (true) {
    // call to getopt_long stores the long option index here.
    int long_index = -1;  // index of long option in long_options[]
    int argvind = optind;  // index of next option in argv[]

    // call getopt_long to get next argument
    int optchr = getopt_long (argc, argv, short_options, long_options, &long_index);

    // signals end of the options
    if (optchr == -1) {
       break;
    }

    // string representation of long or short option
    const char* optstr =
        (long_index >= 0) ? long_options[long_index].name : (char[]){argv[argvind][1],'\0'};
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
// argument not specified if arg from next position is another option
if (optarg != (char*)NULL && *optarg == '-' && long_index < 0) { // only for short opt
        optind = argvind+1;  // reset optind to current option in argv[]
                optarg = (char*)NULL;  // set to no argument
}
switch (optchr) {
case 0:  // brief/verbose flag
        // If this option set a flag, do nothing else now. */

        if (long_options[long_index].flag != (int*)NULL) {
                break;
        }
        printf ("option %s", optstr);
        if (optarg != (char*)NULL) {
           printf (" with value %s", optstr);
        }
        printf ("\n");
        break;
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
case 0:  // brief/verbose flag
case 'a': // add
                app_args.add_opt = true;
                printf ("option %s\n", optstr);
                break;
case 'b':  // brief (short options)
                app_args.output_level = brief;
                break;
case 'c':  // create
                if (optarg == (char*)NULL) {
                                fprintf (stderr, "Option c requires an argument.\n");
                                                return EXIT_FAILURE;
                }
                app_args.create_value = strdup(optarg);
                printf ("option %s with value `%s'\n", optstr, app_args.create_value);
                break;
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
case 'd':  // delete
        if (optarg == (char*)NULL) {
                fprintf (stderr, "Option d requires an argument.\n");
                return EXIT_FAILURE;
        }
        // get argument as a string value
        app_args.delete_value = strdup(optarg);
        printf ("option %s with value `%s'\n", optstr, app_args.delete_value);
        break;
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
case 'l':  // logging
        if (optarg == (char*)NULL) {
                // use default level 0
                app_args.logging_level = 0;
                printf ("option %s default level %d\n",
                        optstr, app_args.logging_level);
        } else {
                // get argument as numeric value
                if (sscanf(optarg,"%d", &app_args.logging_level) == 1) {
                        printf ("option %s with value %d\n",
                                optstr, app_args.logging_level);
                } else {
                        fprintf(stderr, "Logging level '%s; value not numeric\n",
                                optarg);
                        return EXIT_FAILURE;
                }
        }
        break;
```

# Processing Arguments

## Method 3: Getopt Long Argument Processing

```
        case 'p':  // append
                app_args.append_opt = true;
                printf ("option %s\n",  optstr);
                break;
        case 'v':  // verbose (short option)
                app_args.output_level = verbose;
                break;
        case '?':
                if (   strcmp(optstr,"?") == 0
                   || (long_index >= 0 && long_options[long_index].val == '?')) {
                        // help long argument
                        usage(stdout, argv[0]);
                        return EXIT_SUCCESS;  // stop processing if getting help
                } else {  // getopt also uses '?' to signal error
                        fprintf (stderr, "Unknown option: '%s'\n", optstr);
                        return EXIT_FAILURE;
                }
                break;
        default:
                return EXIT_FAILURE;
        }
}
```