# Chapter 49: Jump Statements

## Section 49.1: Using return

**Returning a value**

One commonly used case: returning from `main()`

```
#include <stdlib.h> /* for EXIT_xxx macros */

int main(int argc, char ** argv)
{
  if (2 < argc)
  {
    return EXIT_FAILURE; /* The code expects one argument:
                             leave immediately skipping the rest of the function's code */
  }

  /* Do stuff. */

  return EXIT_SUCCESS;
}
```

Additional notes:

1. For a function having a return type as `void` (not including `void *` or related types), the `return` statement should not have any associated expression; i.e, the only allowed return statement would be `return;`.
2. For a function having a non-`void` return type, the `return` statement shall not appear without an expression.
3. For `main()` (and only for `main()`), an *explicit* `return` statement is not required (in C99 or later). If the execution reaches the terminating `}`, an implicit value of `0` is returned. Some people think omitting this `return` is bad practice; others actively suggest leaving it out.

**Returning nothing**

Returning from a `void` function

```
void log(const char * message_to_log)
{
  if (NULL == message_to_log)
  {
    return; /* Nothing to log, go home NOW, skip the logging. */
  }

  fprintf(stderr, "%s:%d %s\n", __FILE__, _LINE__, message_to_log);

  return; /* Optional, as this function does not return a value. */
}
```

## Section 49.2: Using goto to jump out of nested loops

Jumping out of nested loops would usually require use of a boolean variable with a check for this variable in the loops. Supposing we are iterating over `i` and `j`, it could look like this

```
size_t i,j;
for (i = 0; i < myValue && !breakout_condition; ++i) {
    for (j = 0; j < mySecondValue && !breakout_condition; ++j) {
        ... /* Do something, maybe modifying breakout_condition */
            /* When breakout_condition == true the loops end */
     }
}
```

But the C language offers the `goto` clause, which can be useful in this case. By using it with a label declared after the loops, we can easily break out of the loops.

```
size_t i,j;
for (i = 0; i < myValue; ++i) {
    for (j = 0; j < mySecondValue; ++j) {
        ...
        if(breakout_condition)
          goto final;
    }
}
final:
```

However, often when this need comes up a `return` could be better used instead. This construct is also considered "unstructured" in structural programming theory.

Another situation where `goto` might be useful is for jumping to an error-handler:

```
ptr = malloc(N *  x);
if(!ptr)
  goto out_of_memory;

/* normal processing */
free(ptr);
return SUCCESS;

out_of_memory:
 free(ptr); /* harmless, and necessary if we have further errors */
 return FAILURE;
```

Use of `goto` keeps error flow separate from normal program control flow. It is however also considered "unstructured" in the technical sense.

## Section 49.3: Using break and continue

Immediately `continue` reading on invalid input or **`break`** on user request or end-of-file:

```
#include <stdlib.h> /* for EXIT_xxx macros */
#include <stdio.h>  /* for printf() and getchar() */
#include <ctype.h> /* for isdigit() */

void flush_input_stream(FILE * fp);


int main(void)
{
  int sum = 0;
  printf("Enter digits to be summed up or 0 to exit:\n");

  do
  {
    int c = getchar();
    if (EOF == c)
    {
      printf("Read 'end-of-file', exiting!\n");

      break;
    }

    if ('\n' != c)
    {
      flush_input_stream(stdin);
    }

    if (!isdigit(c))
    {
      printf("%c is not a digit! Start over!\n", c);

      continue;
    }

    if ('0' == c)
    {
      printf("Exit requested.\n");

      break;
    }

    sum += c - '0';

    printf("The current sum is %d.\n", sum);
  } while (1);

  return EXIT_SUCCESS;
}

void flush_input_stream(FILE * fp)
{
  size_t i = 0;
  int c;
  while ((c = fgetc(fp)) != '\n' && c != EOF) /* Pull all until and including the next new-line. */
  {
    ++i;
  }

  if (0 != i)
  {
    fprintf(stderr, "Flushed %zu characters from input.\n", i);
  }
}
```