Chapter 30: Preprocessor and Macros

All preprocessor commands begins with the hash (pound) symbol `#`. A C macro is just a preprocessor command that is defined using the `#define` preprocessor directive. During the preprocessing stage, the C preprocessor (a part of the C compiler) simply substitutes the body of the macro wherever its name appears.

Section 30.1: Header Include Guards

Pretty much every header file should follow the include guard idiom:

my-header-file.h

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H

// Code body for header file

#endif
```

This ensures that when you `#include "my-header-file.h"` in multiple places, you don't get duplicate declarations of functions, variables, etc. Imagine the following hierarchy of files:

header-1.h

```
typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This code has a serious problem: the detailed contents of `MyStruct` is defined twice, which is not allowed. This would result in a compilation error that can be difficult to track down, since one header file includes another. If you instead did it with header guards:

header-1.h

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
```

```
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

This would then expand to:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
```

```
}
```

When the compiler reaches the second inclusion of **header-1.h**, `HEADER_1_H` was already defined by the previous inclusion. Ergo, it boils down to the following:

```
#define HEADER_1_H

typedef struct {
    ...
} MyStruct;

int myFunction(MyStruct *value);

#define HEADER_2_H

int myFunction2(MyStruct *value);

int main() {
    // do something
}
```

And thus there is no compilation error.

Note: There are multiple different conventions for naming the header guards. Some people like to name it `HEADER_2_H_`, some include the project name like `MY_PROJECT_HEADER_2_H`. The important thing is to ensure that the convention you follow makes it so that each file in your project has a unique header guard.

If the structure details were not included in the header, the type declared would be incomplete or an opaque type. Such types can be useful, hiding implementation details from users of the functions. For many purposes, the `FILE` type in the standard C library can be regarded as an opaque type (though it usually isn't opaque so that macro implementations of the standard I/O functions can make use of the internals of the structure). In that case, the `header-1.h` could contain:

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct MyStruct MyStruct;

int myFunction(MyStruct *value);

#endif
```

Note that the structure must have a tag name (here `MyStruct` — that's in the tags namespace, separate from the ordinary identifiers namespace of the typedef name `MyStruct`), and that the `{ ... }` is omitted. This says "there is a structure type `struct MyStruct` and there is an alias for it `MyStruct`".

In the implementation file, the details of the structure can be defined to make the type complete:

```
struct MyStruct {
    ...
};
```

If you are using C11, you could repeat the `typedef struct MyStruct MyStruct;` declaration without causing a compilation error, but earlier versions of C would complain. Consequently, it is still best to use the include guard

idiom, even though in this example, it would be optional if the code was only ever compiled with compilers that supported C11.

Many compilers support the `#pragma once` directive, which has the same results:

my-header-file.h

```
#pragma once

// Code for header file
```

However, `#pragma once` is not part of the C standard, so the code is less portable if you use it.

A few headers do not use the include guard idiom. One specific example is the standard **`<assert.h>`** header. It may be included multiple times in a single translation unit, and the effect of doing so depends on whether the macro `NDEBUG` is defined each time the header is included. You may occasionally have an analogous requirement; such cases will be few and far between. Ordinarily, your headers should be protected by the include guard idiom.

Section 30.2: #if 0 to block out code sections

If there are sections of code that you are considering removing or want to temporarily disable, you can comment it out with a block comment.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;
    return i;
}
*/
```

However, if the source code you have surrounded with a block comment has block style comments in the source, the ending */ of the existing block comments can cause your new block comment to be invalid and cause compilation problems.

```
/* Block comment around whole function to keep it from getting used.
 * What's even the purpose of this function?
int myUnusedFunction(void)
{
    int i = 5;

    /* Return 5 */
    return i;
}
*/
```

In the previous example, the last two lines of the function and the last '*/' are seen by the compiler, so it would compile with errors. A safer method is to use an `#if 0` directive around the code you want to block out.

```
#if 0
/* #if 0 evaluates to false, so everything between here and the #endif are
 * removed by the preprocessor. */
int myUnusedFunction(void)
{
```

```
    int i = 5;
    return i;
}
#endif
```

A benefit with this is that when you want to go back and find the code, it's much easier to do a search for "#if 0" than searching all your comments.

Another very important benefit is that you can nest commenting out code with `#if 0`. This cannot be done with comments.

An alternative to using `#if 0` is to use a name that will not be `#defined` but is more descriptive of why the code is being blocked out. For instance if there is a function that seems to be useless dead code you might use `#if defined(POSSIBLE_DEAD_CODE)` or `#if defined(FUTURE_CODE_REL_020201)` for code needed once other functionality is in place or something similar. Then when going back through to remove or enable that source, those sections of source are easy to find.

Section 30.3: Function-like macros

Function-like macros are similar to **`inline`** functions, these are useful in some cases, such as temporary debug log:

```
#ifdef DEBUG
# define LOGFILENAME "/tmp/logfile.log"

# define LOG(str) do {                              \
  FILE *fp = fopen(LOGFILENAME, "a");              \
  if (fp) {                                         \
    fprintf(fp, "%s:%d %s\n", __FILE__, __LINE__, \
                /* don't print null pointer */     \
                str ?str :"<null>");                \
    fclose(fp);                                     \
  }                                                 \
  else {                                            \
    perror("Opening '" LOGFILENAME "' failed");     \
  }                                                 \
} while (0)
#else
  /* Make it a NOOP if DEBUG is not defined. */
# define LOG(LINE) (void)0
#endif


#include <stdio.h>

int main(int argc, char* argv[])
{
    if (argc > 1)
        LOG("There are command line arguments");
    else
        LOG("No command line arguments");
    return 0;
}
```

Here in both cases (with `DEBUG` or not) the call behaves the same way as a function with `void` return type. This ensures that the `if/else` conditionals are interpreted as expected.

In the `DEBUG` case this is implemented through a `do { ... } while(0)` construct. In the other case, `(void)0` is a statement with no side effect that is just ignored.

An alternative for the latter would be

```
#define LOG(LINE) do { /* empty */ } while (0)
```

such that it is in all cases syntactically equivalent to the first.

If you use GCC, you can also implement a function-like macro that returns result using a non-standard GNU extension — statement expressions. For example:

```
#include <stdio.h>

#define POW(X, Y) \
({ \
        int i, r = 1; \
        for (i = 0; i < Y; ++i) \
                r *= X; \
        r; \ // returned value is result of last operation
})

int main(void)
{
        int result;

        result = POW(2, 3);
        printf("Result: %d\n", result);
}
```

Section 30.4: Source file inclusion

The most common uses of `#include` preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myheader.h"
```

`#include` replaces the statement with the contents of the file referred to. Angle brackets (<>) refer to header files installed on the system, while quotation marks ("") are for user-supplied files.

Macros themselves can expand other macros once, as this example illustrates:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"
    /*  and so on */
#else
    #define INCFILE  "versN.h"
#endif
/* ... */
#include INCFILE
```

Section 30.5: Conditional inclusion and conditional function signature modification

To conditionally include a block of code, the preprocessor has several directives (e.g `#if`, `#ifdef`, `#else`, `#endif`, etc).

```
/* Defines a conditional `printf` macro, which only prints if `DEBUG`
```

```
 * has been defined
 */
#ifdef DEBUG
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Normal C relational operators may be used for the `#if` condition

```
#if __STDC_VERSION__ >= 201112L
/* Do stuff for C11 or higher */
#elif __STDC_VERSION__ >= 199901L
/* Do stuff for C99 */
#else
/* Do stuff for pre C99 */
#endif
```

The `#if` directives behaves similar to the C `if` statement, it shall only contain integral constant expressions, and no casts. It supports one additional unary operator, `defined( identifier )`, which returns `1` if the identifier is defined, and `0` otherwise.

```
#if defined(DEBUG) && !defined(QUIET)
#define DLOG(x) (printf(x))
#else
#define DLOG(x)
#endif
```

Conditional Function Signature Modification

In most cases a release build of an application is expected to have as little overhead as possible. However during testing of an interim build, additional logs and information about problems found can be helpful.

For example assume there is some function `SHORT SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)` which when doing a test build it is desired will generate a log about its use. However this function is used in multiple places and it is desired that when generating the log, part of the information is to know where is the function being called from.

So using conditional compilation you can have something like the following in the include file declaring the function. This replaces the standard version of the function with a debug version of the function. The preprocessor is used to replace calls to the function `SerOpPluAllRead()` with calls to the function `SerOpPluAllRead_Debug()` with two additional arguments, the name of the file and the line number of where the function is used.

Conditional compilation is used to choose whether to override the standard function with a debug version or not.

```
#if 0
// function declaration and prototype for our debug version of the function.
SHORT   SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo);

// macro definition to replace function call using old name with debug function with additional
arguments.
#define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead_Debug(pPif,usLock,__FILE__,__LINE__)
#else
// standard function declaration that is normally used with builds.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd);
#endif
```

This allows you to override the standard version of the function `SerOpPluAllRead()` with a version that will provide the name of the file and line number in the file of where the function is called.

There is one important consideration: *any file using this function must include the header file where this approach is used in order for the preprocessor to modify the function. Otherwise you will see a linker error.*

The definition of the function would look something like the following. What this source does is to request that the preprocessor rename the function `SerOpPluAllRead()` to be `SerOpPluAllRead_Debug()` and to modify the argument list to include two additional arguments, a pointer to the name of the file where the function was called and the line number in the file at which the function is used.

```
#if defined(SerOpPluAllRead)
// forward declare the replacement function which we will call once we create our log.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd);

SHORT    SerOpPluAllRead_Debug(PLUIF *pPif, USHORT usLockHnd, char *aszFilePath, int nLineNo)
{
    int iLen = 0;
    char  xBuffer[256];

    // only print the last 30 characters of the file name to shorten the logs.
    iLen = strlen (aszFilePath);
    if (iLen > 30) {
        iLen = iLen - 30;
    }
    else {
        iLen = 0;
    }

    sprintf (xBuffer, "SerOpPluAllRead_Debug(): husHandle = %d, File %s, lineno = %d",
pPif->husHandle, aszFilePath + iLen, nLineNo);
    IssueDebugLog(xBuffer);

    // now that we have issued the log, continue with standard processing.
    return SerOpPluAllRead_Special(pPif, usLockHnd);
}

// our special replacement function name for when we are generating logs.
SHORT    SerOpPluAllRead_Special(PLUIF *pPif, USHORT usLockHnd)
#else
// standard, normal function name (signature) that is replaced with our debug version.
SHORT   SerOpPluAllRead(PLUIF *pPif, USHORT usLockHnd)
#endif
{
    if (STUB_SELF == SstReadAsMaster()) {
        return OpPluAllRead(pPif, usLockHnd);
    }
    return OP_NOT_MASTER;
}
```

Section 30.6: __cplusplus for using C externals in C++ code compiled with C++ - name mangling

There are times when an include file has to generate different output from the preprocessor depending on whether the compiler is a C compiler or a C++ compiler due to language differences.

For example a function or other external is defined in a C source file but is used in a C++ source file. Since C++ uses name mangling (or name decoration) in order to generate unique function names based on function argument

types, a C function declaration used in a C++ source file will cause link errors. The C++ compiler will modify the specified external name for the compiler output using the name mangling rules for C++. The result is link errors due to externals not found when the C++ compiler output is linked with the C compiler output.

Since C compilers do not do name mangling but C++ compilers do for all external labels (function names or variable names) generated by the C++ compiler, a predefined preprocessor macro, `__cplusplus`, was introduced to allow for compiler detection.

In order to work around this problem of incompatible compiler output for external names between C and C++, the macro `__cplusplus` is defined in the C++ Preprocessor and is not defined in the C Preprocessor. This macro name can be used with the conditional preprocessor `#ifdef` directive or `#if` with the `defined()` operator to tell whether a source code or include file is being compiled as C++ or C.

```
#ifdef __cplusplus
printf("C++\n");
#else
printf("C\n");
#endif
```

Or you could use

```
#if defined(__cplusplus)
printf("C++\n");
#else
printf("C\n");
#endif
```

In order to specify the correct function name of a function from a C source file compiled with the C compiler that is being used in a C++ source file you could check for the `__cplusplus` defined constant in order to cause the `extern "C" { /* ... */ };` to be used to declare C externals when the header file is included in a C++ source file. However when compiled with a C compiler, the `extern "C" { */ ... */ };` is not used. This conditional compilation is needed because `extern "C" { /* ... */ };` is valid in C++ but not in C.

```
#ifdef __cplusplus
// if we are being compiled with a C++ compiler then declare the
// following functions as C functions to prevent name mangling.
extern "C" {
#endif

// exported C function list.
int foo (void);

#ifdef __cplusplus
// if this is a C++ compiler, we need to close off the extern declaration.
};
#endif
```

Section 30.7: Token pasting

Token pasting allows one to glue together two macro arguments. For example, `front##back` yields `frontback`. A famous example is Win32's **<TCHAR.H>** header. In standard C, one can write `L"string"` to declare a wide character string. However, Windows API allows one to convert between wide character strings and narrow character strings simply by `#define`ing `UNICODE`. In order to implement the string literals, `TCHAR.H` uses this

```
#ifdef UNICODE
#define TEXT(x) L##x
```

```
#endif
```

Whenever a user writes `TEXT("hello, world")`, and UNICODE is defined, the C preprocessor concatenates `L` and the macro argument. `L` concatenated with `"hello, world"` gives `L"hello, world"`.

Section 30.8: Predefined Macros

A predefined macro is a macro that is already understood by the C pre processor without the program needing to define it. Examples include

Mandatory Pre-Defined Macros

- `__FILE__`, which gives the file name of the current source file (a string literal),
- `__LINE__` for the current line number (an integer constant),
- `__DATE__` for the compilation date (a string literal),
- `__TIME__` for the compilation time (a string literal).

There's also a related predefined identifier, `__func__` (ISO/IEC 9899:2011 §6.4.2.2), which is *not* a macro:

> The identifier `__func__` shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration:
>
> ```
> static const char __func__[] = "function-name";
> ```
>
> appeared, where *function-name* is the name of the lexically-enclosing function.

`__FILE__`, `__LINE__` and `__func__` are especially useful for debugging purposes. For example:

```
fprintf(stderr, "%s: %s: %d: Denominator is 0", __FILE__, __func__, __LINE__);
```

Pre-C99 compilers, may or may not support `__func__` or may have a macro that acts the same that is named differently. For example, gcc used `__FUNCTION__` in C89 mode.

The below macros allow to ask for detail on the implementation:

- `__STDC_VERSION__` The version of the C Standard implemented. This is a constant integer using the format `yyyymmL` (the value `201112L` for C11, the value `199901L` for C99; it wasn't defined for C89/C90)
- `__STDC_HOSTED__` `1` if it's a hosted implementation, else `0`.
- `__STDC__` If `1`, the implementation conforms to the C Standard.

Other Pre-Defined Macros (non mandatory)

ISO/IEC 9899:2011 §6.10.9.2 Environment macros:

> - `__STDC_ISO_10646__` An integer constant of the form `yyyymmL` (for example, 199712L). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The Unicode required set consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
> - `__STDC_MB_MIGHT_NEQ_WC__` The integer constant 1, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when

> used as the lone character in an integer character constant.
>
> - `__STDC_UTF_16__` The integer constant 1, intended to indicate that values of type `char16_t` are UTF−16 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.
> - `__STDC_UTF_32__` The integer constant 1, intended to indicate that values of type `char32_t` are UTF−32 encoded. If some other encoding is used, the macro shall not be defined and the actual encoding used is implementation-defined.

ISO/IEC 9899:2011 §6.10.8.3 Conditional feature macros

> - `__STDC_ANALYZABLE__` The integer constant 1, intended to indicate conformance to the specifications in annex L (Analyzability).
> - `__STDC_IEC_559__` The integer constant 1, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).
> - `__STDC_IEC_559_COMPLEX__` The integer constant 1, intended to indicate adherence to the specifications in annex G (IEC 60559 compatible complex arithmetic).
> - `__STDC_LIB_EXT1__` The integer constant `201112L`, intended to indicate support for the extensions defined in annex K (Bounds-checking interfaces).
> - `__STDC_NO_ATOMICS__` The integer constant 1, intended to indicate that the implementation does not support atomic types (including the `_Atomic` type qualifier) and the **`<stdatomic.h>`** header.
> - `__STDC_NO_COMPLEX__` The integer constant 1, intended to indicate that the implementation does not support complex types or the **`<complex.h>`** header.
> - `__STDC_NO_THREADS__` The integer constant 1, intended to indicate that the implementation does not support the **`<threads.h>`** header.
> - `__STDC_NO_VLA__` The integer constant 1, intended to indicate that the implementation does not support variable length arrays or variably modified types.

Section 30.9: Variadic arguments macro

`Version ≥ C99`

Macros with variadic args:

Let's say you want to create some print-macro for debugging your code, let's take this macro as an example:

```
#define debug_print(msg) printf("%s:%d %s", __FILE__, __LINE__, msg)
```

Some examples of usage:

The function `somefunc()` returns -1 if failed and 0 if succeeded, and it is called from plenty different places within the code:

```
int retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}

/* some other code */
```

```
 retVal = somefunc();

if(retVal == -1)
{
    debug_printf("somefunc() has failed");
}
```

What happens if the implementation of `somefunc()` changes, and it now returns different values matching different possible error types? You still want use the debug macro and print the error value.

```
debug_printf(retVal);      /* this would obviously fail */
debug_printf("%d",retVal); /* this would also fail */
```

To solve this problem the `__VA_ARGS__` macro was introduced. This macro allows multiple parameters X-macro's:

Example:

```
#define debug_print(msg, ...) printf(msg, __VA_ARGS__) \
                              printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
int retVal = somefunc();

debug_print("retVal of somefunc() is-> %d", retVal);
```

This macro allows you to pass multiple parameters and print them, but now it forbids you from sending any parameters at all.

```
debug_print("Hey");
```

This would raise some syntax error as the macro expects at least one more argument and the pre-processor would not ignore the lack of comma in the `debug_print()` macro. Also `debug_print("Hey",);` would raise a syntax error as you cant keep the argument passed to macro empty.

To solve this, `##__VA_ARGS__` macro was introduced, this macro states that if no variable arguments exist, the comma is deleted by the pre-processor from code.

Example:

```
#define debug_print(msg, ...) printf(msg, ##__VA_ARGS__) \
                              printf("\nError occurred in file:line (%s:%d)\n", __FILE__, __LINE)
```

Usage:

```
debug_print("Ret val of somefunc()?");
debug_print("%d",somefunc());
```

Section 30.10: Macro Replacement

The simplest form of macro replacement is to define a `manifest constant`, as in

```
#define ARRSIZE 100
int array[ARRSIZE];
```

This defines a *function-like* macro that multiplies a variable by `10` and stores the new value:

```
#define TIMES10(A) ((A) *= 10)

double b = 34;
int c = 23;

TIMES10(b);   // good: ((b) *= 10);
TIMES10(c);   // good: ((c) *= 10);
TIMES10(5);   // bad:  ((5) *= 10);
```

The replacement is done before any other interpretation of the program text. In the first call to `TIMES10` the name `A` from the definition is replaced by `b` and the so expanded text is then put in place of the call. Note that this definition of `TIMES10` is not equivalent to

```
#define TIMES10(A) ((A) = (A) * 10)
```

because this could evaluate the replacement of `A`, twice, which can have unwanted side effects.

The following defines a function-like macro which value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and of generating more code than a function if invoked several times.

```
#define max(a, b) ((a) > (b) ? (a) : (b))

int maxVal = max(11, 43);              /* 43 */
int maxValExpr = max(11 + 36, 51 - 7); /* 47 */

/* Should not be done, due to expression being evaluated twice */
int j = 0, i = 0;
int sideEffect = max(++i, ++j);       /* i == 4 */
```

Because of this, such macros that evaluate their arguments multiple times are usually avoided in production code. Since C11 there is the `_Generic` feature that allows to avoid such multiple invocations.

The abundant parentheses in the macro expansions (right hand side of the definition) ensure that the arguments and the resulting expression are bound properly and fit well into the context in which the macro is called.

Section 30.11: Error directive

If the preprocessor encounters an `#error` directive, compilation is halted and the diagnostic message included is printed.

```
#define DEBUG

#ifdef DEBUG
#error "Debug Builds Not Supported"
#endif

int main(void) {
    return 0;
}
```

Possible output:

```
$ gcc error.c
error.c: error: #error "Debug Builds Not Supported"
```

Section 30.12: FOREACH implementation

We can also use macros for making code easier to read and write. For example we can implement macros for implementing the `foreach` construct in C for some data structures like singly- and doubly-linked lists, queues, etc.

Here is a small example.

```
#include <stdio.h>
#include <stdlib.h>

struct LinkedListNode
{
    int data;
    struct LinkedListNode *next;
};

#define FOREACH_LIST(node, list) \
     for (node=list; node; node=node->next)

/* Usage */
int main(void)
{
    struct LinkedListNode *list, **plist = &list, *node;
    int i;

    for (i=0; i<10; i++)
    {
         *plist = malloc(sizeof(struct LinkedListNode));
         (*plist)->data = i;
         (*plist)->next = NULL;
         plist          = &(*plist)->next;
    }

    /* printing the elements here */
    FOREACH_LIST(node, list)
    {
        printf("%d\n", node->data);
    }
}
```

You can make a standard interface for such data-structures and write a generic implementation of `FOREACH` as:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct CollectionItem_
{
    int data;
    struct CollectionItem_ *next;
} CollectionItem;

typedef struct Collection_
{
    /* interface functions */
    void* (*first)(void *coll);
    void* (*last) (void *coll);
    void* (*next) (void *coll, CollectionItem *currItem);
```

```
    CollectionItem *collectionHead;
    /* Other fields */
} Collection;

/* must implement */
void *first(void *coll)
{
    return ((Collection*)coll)->collectionHead;
}

/* must implement */
void *last(void *coll)
{
    return NULL;
}

/* must implement */
void *next(void *coll, CollectionItem *curr)
{
    return curr->next;
}

CollectionItem *new_CollectionItem(int data)
{
    CollectionItem *item = malloc(sizeof(CollectionItem));
    item->data = data;
    item->next = NULL;
    return item;
}

void Add_Collection(Collection *coll, int data)
{
    CollectionItem **item = &coll->collectionHead;
    while(*item)
        item = &(*item)->next;
    (*item) = new_CollectionItem(data);
}

Collection *new_Collection()
{
    Collection *nc = malloc(sizeof(Collection));
    nc->first = first;
    nc->last  = last;
    nc->next  = next;
    return nc;
}

/* generic implementation */
#define FOREACH(node, collection)                    \
    for (node  = (collection)->first(collection);    \
         node != (collection)->last(collection);     \
         node  = (collection)->next(collection, node))

int main(void)
{
    Collection *coll = new_Collection();
    CollectionItem *node;
    int i;

    for(i=0; i<10; i++)
    {
         Add_Collection(coll, i);
```

```
    }

    /* printing the elements here */
    FOREACH(node, coll)
    {
        printf("%d\n", node->data);
    }
}
```

To use this generic implementation just implement these functions for your data structure.

```
1.  void* (*first)(void *coll);
2.  void* (*last) (void *coll);
3.  void* (*next) (void *coll, CollectionItem *currItem);
```