Patterns to Escape the #ifdef Hell

CHRISTOPHER PRESCHERN, B&R Industrial Automation GmbH

There are many things one can do wrong when using the C preprocessor's #ifdef statements. Such statements are often used to distinguish between hardware variants or operating system variants in the C code and when not used carefully, the C code easily becomes a mess of #ifdef cascades. This paper addresses this issue and provides best practices in form of patterns on how to organize variants in C code and on how to get rid of #ifdef statements.

INTRODUCTION

The C programming language is widespread, in particular with systems where high performance or hardware-near programming is required. With hardware-near programming comes the necessity to cope with hardware variants. Aside from hardware variants, some systems have to cope with supporting multiple operating systems. A commonly used approach to address these issues is to use #ifdef statements of the C preprocessor in order to distinguish hardware or operating system variants in the code. The C preprocessor comes with this power, but with this power also comes the responsibility to use it in a well structured way.

However, that is where the weakness of the C preprocessor with its #ifdef statements shows up. The C-preprocessor does not support any methods to enforce rules regarding its usage. That is a pity, because it can very easily be abused. It is very easy to add another hardware variant or another optional feature in the code by adding yet another #ifdef. Also, #ifdef statements can easily be abused to add quick bug-fixes which only affect a single variant. That makes the code for different variants more diverse which leads to code which more and more has to be fixed for each of the variants separately. Using #ifdef statements in such an unstructured and ad-hoc way is the certain way to hell. The code becomes unreadable and unmaintainable and that is definitely something each developer wants to avoid.

This paper presents approaches to escape from such a hell, or even better: A way to avoid getting into that hell. Even though the C language and its preprocessor exist for many decades now, there is surprisingly few literature on that topic. There is literature available on writing portable code in general or on supporting feature variants in general. There is even literature available on doing that in the C language [Hook 2005]. Such literature addresses a broader topic and covers operating system variants and hardware variants in detail, by for example giving advice for coping with byte-ordering, data-type-sizes or line-separator tokens. However, when it comes to the very specific topic on giving detailed guidance on how to handle these variants in the C code, there is very few literature available [Spencer and Collyer 1992][Malenfant 2000].

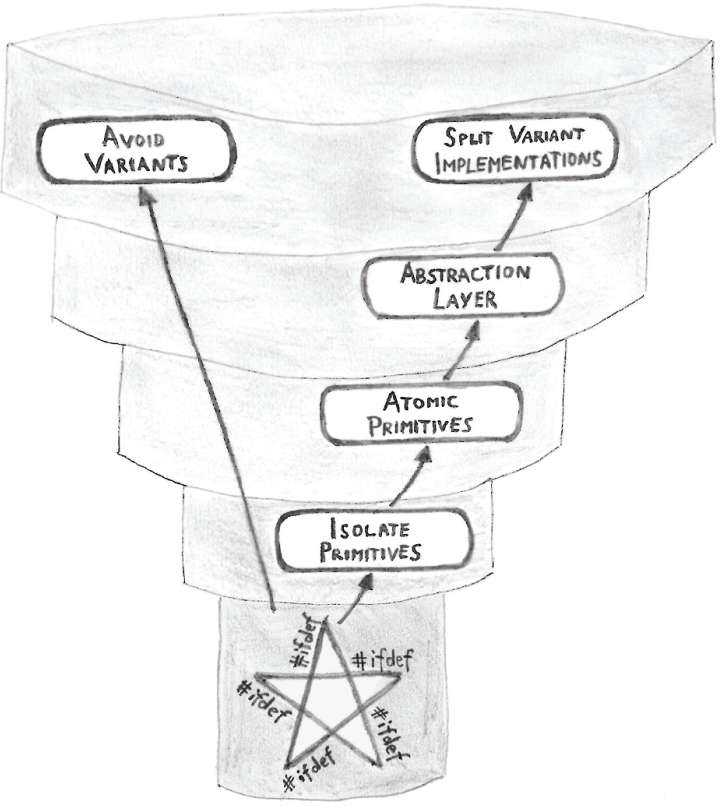
That is where this paper comes in. The paper gives detailed guidance on how to support variants, like operating system variants or hardware variants, in C code and the paper gives detailed guidance on how to use and how to get rid of #ifdef statements. The paper gives this guidance in form of patterns and, to make these patterns easier to grasp, the paper shows their application to a running example.

OVERVIEW OF PATTERNS PRESENTED IN THIS PAPER

This paper presents five patterns on how to cope with code variants and on how to organize or even get rid of #ifdef statements. The patterns can be seen as a step-by-step introduction into organizing such code or maybe as a step-by-step guide on how to refactor unstructured #ifdef code.

Take these steps to escape the #ifdef hell:

1) AVOID VARIANTS. If possible don't implement code variants for different platforms, but instead use standardized functions, such as C Library or POSIX functions.
2) ISOLATE PRIMITIVES. If you do have code variants, then do not mix program logic and these variants. Instead pack the variants into separate functions.
3) ATOMIC PRIMITIVES. The functions which handle the variants should each only handle exactly one kind of variant. Do not mix, for example, operating system variants and hardware variants in one single function.
4) ABSTRACTION LAYER. Hide the functions which handle the variants behind an API. Every user of the API should not have to cope with platform specific issues.
5) SPLIT VARIANT IMPLEMENTATIONS. Put each of the implementations into a separate c-file. That makes it possible only have #ifdef statements across whole files or to let makefiles decide what to build on which platform.



PATTERNS AND RUNNING EXAMPLE

Running example:

You want to implement the functionality to write some text into a file to be stored in a newly created directory, which, depending on a configuration flag, is either created in the user- or home-directory. To make things more complicated, your code should run on Windows systems as well as on Linux systems.

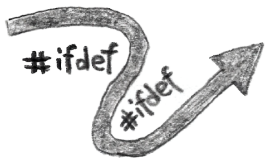
Your first attempt is to have one single implementation file which contains all the code for all configurations and operating systems. To do that, the file contains many #ifdef statements to distinguish between the code variants.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
  #include <sys/stat.h>
  #include <fcntl.h>
  #include <unistd.h>
#elif defined _WIN32
  #include <windows.h>
#endif

int main()
{
  char dirname[50];
  char filename[50];
  char* my_data = "Write this data to the file";
  #ifdef __unix__
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
      sprintf(filename, "%s%s", dirname, "newfile");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir");
      strcpy(filename, "newdir/newfile");
    #endif
    mkdir(dirname,S_IRWXU);
    int fd = open (filename, O_RDWR | O_CREAT, 0666);
    write(fd, my_data, strlen(my_data));
    close(fd);
  #elif defined _WIN32
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"), "\\newdir\\");
      sprintf(filename, "%s%s", dirname, "newfile");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir");
      strcpy(filename, "newdir\\newfile");
    #endif
    CreateDirectory (dirname, NULL);
    HANDLE hFile = CreateFile(filename, GENERIC_WRITE, 0, NULL,
                              CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
    WriteFile(hFile, my_data, strlen(my_data), NULL, NULL);
    CloseHandle(hFile);
  #endif
  return 0;
}
```

This code is chaos. The program logic is completely duplicated. This is not operating system independent code, but instead it is only two different operating system specific implementations put into one single file. Especially the orthogonal code variants of different operating systems and different places for creating the directory make the code very ugly as they lead to the nested #ifdef statements above, which are very hard to understand. When reading the code, you always have to jump between the lines. You have to skip the code from other #ifdef branches, in order to follow the program logic. Such duplicated program logic invites programmers to fix errors or to add new features only in the code variant, on which they currently work on. That makes the code pieces and the behavior for the variants drift apart which makes the code hard to maintain.

Where to start? How to clean this mess up? As a first step, if possible, you can use standardized functions in order to AVOID VARIANTS.

AVOID VARIANTS



Context:
You write portable code which should be used on multiple operating system platforms or on multiple hardware platforms. Some of the functions you call in your code are available on one platform, but are not available in exactly the same syntax and semantic on another platform. Because of that, you implement code variants - one for each platform. Now you have different pieces of code for your different platforms and you distinguish between the variants with #ifdef statements in your code.

Problem:
Using different functions for each platform makes the code harder to read and harder to write. The programmer is required to initially understand, to correctly use, and to test these multiple functions in order to achieve one single functionality across multiple platforms.

Quite often it is the aim that you implement functionality which should behave exactly the same on all platforms. However, when using platform-dependent functions, that aim is more difficult to achieve and might require writing additional code, because not only the syntax, but also the semantics of the functions might slightly differ between the platforms.

Using multiple functions for multiple platforms make the code not only more difficult to write, but also more difficult to read and to understand. Distinguishing between the different functions with #ifdef statements makes the code longer and requires the reader to jump across lines when trying to find out what the code does for one single #ifdef branch.

Solution:
Use standardized functions which are available on all platforms. In case there are no standardized functions, consider to not implement the functionality.

Good examples for standardized functions which you can use are the C Standard Library functions and the POSIX functions. If possible, these functions should be used instead of more specific functions which are only available on one of the platforms.

Not at all implementing the functionality only because there is no standardized function will not always be an option. However, if there are only platform-dependent functions available for the functionality you want to achieve, then seriously consider whether providing that functionality in your product is worth it. Maybe maintaining different code for different platforms is not worth the coding or testing effort.

However, in some cases you do have to provide functionality in your product even if there are no standardized functions available. That means that you have to use different functions across different platforms or maybe even have to implement features on one platform which are alredy available on another. To do that in a structured way, ISOLATE PRIMITIVES for your code variants and hide them behind an ABSTRACTION LAYER.

Code Example

CALLER

```
#include <standardizedApi.h>

int main()
{
  /* just one single function
     instead of multiple via
     ifdef distinguished
     functions is called*/
  somePosixFunction();
  return 0;
}
```

STANDARDIZED API

```
/* this function is available
   on all operating systems
   which adhere to the POSIX
   standard */
somePosixFunction();
```

To avoid variants, for example, use C Library file access functions like `fopen` instead of using operating system specific functions like Linux' `open` or Windows' `CreateFile` functions. Another example are the C Library time functions. Avoid using operating system specific time functions like Windows' `GetLocalTime` and Linux' `localtime_r`, but instead use the standardized `localtime` function from `time.h`.

Consequences:

The code is simple to write and to read, because one single piece of code can be used for multiple platforms. The programmer does not have to understand different functions for different platforms when writing the code and the programmer does not have to jump between #ifdef branches when reading the code.

As there is the same piece of code on all platforms, the problem does not occur that on different platforms functions for similar functionality might slightly differ in their behavior. You just use one single function which is standardized and which behaves the same on each platform which adheres to the standard.

The standardized function might not be the most efficient and the most high-performance way to achieve the required functionality on each of the platforms. Some platforms might provide other platform specific functions which, for example, use specialized hardware on that platform to achieve higher performance. Such hardware specific advantages might not be used by the standardized functions.

Known Uses:

- The code of the editor VIM uses the operating system independent functions `fopen`, `fwrite`, `fread`, and `fclose` to access files.
- The openSSL code writes the current local time to its log messages. To do that, it converts the current UTC time to local time using the operating system independent function `localtime`.
- The openSSL function `BIO_lookup_ex` looks up the node and service to connect to. This function is compiled on Windows and Linux and uses the operating system independent function `htons` to convert a value to network byte order.

Running example:

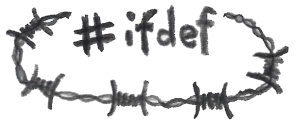
For your functionality to access files, you are in the lucky position where there are operating system independent functions available. The functions `fopen`, `fwrite`, and `fclose` are part of the C Library and are available on Windows as well as on Linux. That makes your code already a lot easier:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
  #include <sys/stat.h>
#elif defined _WIN32
  #include <windows.h>
#endif

int main()
{
  char dirname[50];
  char filename[50];
  char* my_data = "Write this data to the file";
  #ifdef __unix__
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
      sprintf(filename, "%s%s", dirname, "newfile");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir");
      strcpy(filename, "newdir/newfile");
    #endif
    mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"), "\\newdir\\");
      sprintf(filename, "%s%s", dirname, "newfile");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir");
      strcpy(filename, "newdir\\newfile");
    #endif
    CreateDirectory(dirname, NULL);
  #endif
  FILE* f = fopen(filename, "w+");
  fwrite(my_data, 1, strlen(my_data), f);
  fclose(f);
  return 0;
}
```

When looking at the last three function calls in the code above, your code already is a lot simpler. Instead of having the separate file access calls for Windows and for Linux, you now have have one common code for that. Having that common code has the major advantage that you can be sure that the calls have the same behavior for both operating systems and there is no danger that two different implementations run apart after bug-fixes or added features.

Still, your code is dominated by #ifdefs and due to that, The code is very difficult to read. Therefore, make sure that your main program logic does not get obfuscated by code variants. ISOLATE PRIMITIVES containing the code variants from the main program logic.

ISOLATE PRIMITIVES



Context:
Your code calls platform specific functions. You have different pieces of code for different platforms and you distinguish between the code variants with #ifdef statements. You cannot simply AVOID VARIANTS, because there are no standardized functions available which provide the feature you need in a uniform way on all your platforms.

Problem:
Having code variants organized with #ifdef statements makes the code unreadable. It is very difficult to follow the program flow, because it is implemented multiple times for multiple platforms.

When trying to understand the code, you usually just focus on one platform and then you have to jump between the lines in the code in order to only scan through the code variant you are interested in.

The #ifdef statements also make the code difficult to maintain. Such statements invite a programmer to only fix the code for the one platform he or she is interested in and to not touch any other code, because of the danger of breaking it. However, only fixing a bug or only introducing a new feature for one single platforms means that the behavior of the code on the different platforms drifts apart. The alternative, to fix such a bug on all platforms in different ways, requires testing the code on all platforms.

Testing code with many code variants is difficult. Each new kind of #ifdef statement doubles the testing effort as all possible combinations have to be tested. Even worse: Each such statement doubles the number of binaries that can be built and have to be tested. That brings in a logistic problem, because build times increase and the number of binaries provided to the test department and to the customer increase.

Solution:
Isolate your code variants. In your implementation file, put the code handling the variants into separate functions and call these functions from your main program logic which then only contains platform independent code.

Each of your functions should either only contain program logic or it should only cope with handling variants. None of your functions should do both. So either there is no #ifdef statement at all in a function, or there are just #ifdef statements with one single platform dependent or feature dependent function call per #ifdef branch.

When only having one single function call per #ifdef branch, then finding a good abstraction granularity for the functions handling the variants should not be a problem. Usually the granularity is exactly at the level of the available platform specific or feature specific functions to be wrapped.

If the functions which handle the variants are still complicated and contain #ifdef cascades, then making sure to only have ATOMIC VARIANTS helps.

Code Example

```
static void handlePlatformVariants()
{
  #ifdef PLATFORM_A
    /* call function of platform A */
  #elif defined PLATFORM_B
    /* call function of platform B */
  #endif
}

int main()
{
  /* program logic goes here */

  handlePlatformVariants();

  /* program logic continues */
}
```

Consequences:

The main program logic is now easy to follow, because the code variants are separated from it. When reading the main code, it is not necessary anymore to jump between the lines in order to find out what the code does on one specific platform.

For finding out, what the code does on one specific platform, you have to look at the called function which implements this variant. Having such a function has the advantage that it can be called from other places in the file as well and thus code duplications can be avoided. If the functionality is also required in other implementation files, then an ABSTRACTION LAYER has to be implemented.

As no program logic should be introduced in the functions handling the variants, it is quite difficult to make a bug-fix which only affects a single platform and it is quite easy to pinpoint bugs which do not occur on all platforms, because it is now easy to pinpoint the places in the code where the behavior of the platforms differ.

Code duplication becomes less of an issue, because as the main program logic is well separated from the variant implementations, there is no temptation to duplicate the program logic anymore.

Known Uses:

- The code of the editor VIM isolates the function `htonl2`, which converts data to network byte order. The program logic of VIM uses `htonl2`, which is defined as a macro in the implementation file. The macro is compiled differently depending on the platform endianness.
- The openSSL function `BIO_ADDR_make` copies socket information into an internal struct. The function uses #ifdef statements to handle operating system specific and feature specific variants distinguishing between Linux/Windows and IPv4/IPv6. The function isolates these variants from the main program logic.
- The function `load_rcfile` of GNUplot reads data from an initialization file and isolated operating system specific file access operations from the rest of the code.

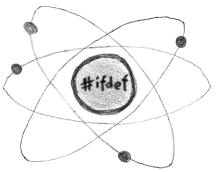
Running example:

Now that you ISOLATED PRIMITIVES, your main program logic is a lot easier to read without requiring the reader to jump between the lines only to keep the variants apart.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __unix__
  #include <sys/stat.h>
#elif defined _WIN32
  #include <windows.h>
#endif

static void getDirectoryName(char* dirname)
{
  #ifdef __unix__
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir/");
    #endif
  #elif defined _WIN32
    #ifdef STORE_IN_HOME_DIR
      sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"), "\\newdir\\");
    #elif defined STORE_IN_CWD
      strcpy(dirname, "newdir\\");
    #endif
  #endif
}

static void createNewDirectory(char* dirname)
{
  #ifdef __unix__
    mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
    CreateDirectory (dirname, NULL);
  #endif
}

int main()
{
  char dirname[50];
  char filename[50];
  char* my_data = "Write this data to the file";
  getDirectoryName(dirname);
  createNewDirectory(dirname);
  sprintf(filename, "%s%s", dirname, "newfile");
  FILE* f = fopen(filename, "w+");
  fwrite(my_data, 1, strlen(my_data), f);
  fclose(f);
  return 0;
}
```

The code variants are now quite well isolated. The program logic of the `main` function is very easy to read and to understand as there are no variants anymore in this function. However, the new function `getDirectoryName` is still dominated by #ifdefs and is not easy to comprehend. Here it could help to only have ATOMIC PRIMITIVES.

ATOMIC PRIMITIVES



Context:
You implemented variants in your code with #ifdef statements and you put these variants into separate functions in order to ISOLATE PRIMITIVES which handle these variants. The primitives separate the variants from the main program flow which makes the main program well structured and easy to comprehend.

Problem:
The function which contains the variants and which is called by the main program is still hard to comprehend, because all the complex #ifdef code was simply put into this function in order to get rid of it in the main program.

Simply handling all kinds of variants in one function becomes difficult as soon as there are many different variants to handle. If, for example, a single function distinguishes with #ifdef statements between different hardware types and different operating systems, then adding an additional operating system variant becomes difficult, because it has to be added for all hardware variants. Each variant cannot be handled in one place anymore, but instead the effort multiplies with the number of different kinds of variants. That is a problem. It should be easy to add new variants at one place in the code.

Solution:
Make your primitives atomic by only handling exactly one kind of variant per function. If you handle multiple kinds of variants, for example, operating system variants and hardware variants, then have separate functions for that.

Let one of these functions call the other which already abstracts one kind of variant. If you abstract a platform-dependence and a feature-dependence, then let the feature-dependent function be the one calling the platform-dependent function, because usually you provide features across all platforms, so platform-dependent functions should be the most atomic ones.

If there is a function which apparently has to provide some functionality across multiple kinds of variants and handles all these kind of variants, then the function scope might be wrong. Perhaps the function is too general or does more than one thing. Split the function.

Call atomic primitives in your main code containing the program logic. If you want to use the atomic primitives in other implementation files, then use an ABSTRACTION LAYER.

Code Example

```
static void handleHardwareOfFeatureX()
{
  #ifdef HARDWARE_A
   /* call function for feature X on hardware A */
  #elif defined HARDWARE_B || defined HARDWARE_C
   /* call function for feature X on hardware B and C */
  #endif
}
```

```
static void handleHardwareOfFeatureY()
{
  #ifdef HARDWARE_A
   /* call function for feature Y on hardware A */
  #elif defined HARDWARE_B
   /* call function for feature Y on hardware B */
  #elif defined HARDWARE_C
   /* call function for feature Y on hardware C */
  #endif
}

void callFeature()
{
  #ifdef FEATURE_X
    handleHardwareOfFeatureX();
  #elif defined FEATURE_Y
    handleHardwareOfFeatureY();
  #endif
}
```

Consequences:

Each function now only handles one single kind of variant. That makes each of the functions easy to understand as there are no more cascades of #ifdef statements. Each of the functions now only abstracts one kind of variant and does no more than exactly that one thing. So the functions do follow the single responsibility principle.

Having no #ifdef cascades makes it less tempting for programmers to simply handle one additional kind of variant in one single function, because starting an #ifdef cascade is less likely done compared to extending an existing cascade.

With separate functions, each kind of variant can easily be extended for an additional variant. To achieve that, only one single #ifdef branch has to be added in one single function and the functions which handle other kinds of variants do not have to be touched.

Known Uses:

- The openSSL implementation file `threads_pthread.c` contains functions for thread handling. There are separate functions to abstract operating systems and separate functions to abstract whether pthreads are available at all.
- The code of sqlite contains functions to abstract operating system specific file access (for example the `fileStat` function). The code abstracts file access related compile time features with other, separate functions.

- The function `boot_jump_linux` calls another function which performs different boot actions depending on the CPU architecture which is handled via #ifdef statements in that function. Then the function `boot_jump_linux` calls another function which uses #ifdef statements to select which configured resources (USB, network, ...) have to be cleaned up.

Running example:

With ATOMIC PRIMITIVES you now have the following code for your functions to determine the directory path:

```
static void getHomeDirectory(char* dirname)
{
  #ifdef __unix__
    sprintf(dirname, "%s%s", getenv("HOME"), "/newdir/");
  #elif defined _WIN32
    sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"), getenv("HOMEPATH"), "\\newdir\\");
  #endif
}

static void getWorkingDirectory(char* dirname)
{
  #ifdef __unix__
    strcpy(dirname, "newdir/");
  #elif defined _WIN32
    strcpy(dirname, "newdir\\");
  #endif
}

static void getDirectoryName(char* dirname)
{
  #ifdef STORE_IN_HOME_DIR
    getHomeDirectory(dirname);
  #elif defined STORE_IN_CWD
    getWorkingDirectory(dirname);
  #endif
}
```

The code variants are now very well isolated. For obtaining the directory name, instead of having one complicated function with many #ifdefs, you now have several functions which only have one #ifdef each. That makes it a lot easier to understand the code, because now each of these function only performs one single thing instead of distinguishing between several kinds of variants with #ifdef cascades.

The functions are now very simple and easy to read. However, your implementation file is still very long and the one single implementation file contains the main program logic as well as code to distinguish between variants. That makes parallel development or separate testing of the variant code next to impossible.

To improve that, best split the implementations up up by inserting an ABSTRACTION LAYER.

ABSTRACTION LAYER

Context:
You have platform variants for which you distinguish with #ifdef statements in your code. You maybe already ISOLATED PRIMITIVES to separate the variants from the program logic and you made sure that you have ATOMIC PRIMITIVES.

Problem:
You want to make it possible to use the code, which abstracts the platforms, also from other implementation files. However, you do not simply want to duplicate that code.

You don't want the caller at all having to cope with platform variants. In the caller code, it should not be necessary to know anything about the implementation details of the functionality for the different platforms. The caller should not have to use any #ifdef statements and the caller should not even have to include any platform specific headerfiles.

You want to make it possible to work on the platform specific code without requiring the caller of this code to care about that. Maybe you even want to have the platform-dependent code to be developed and tested by other programmers than those responsible for the platform-independent code.

If programmers of the platform-dependent code perform a bug-fix for one platform or if they add an additional platform, then this must not require changes to the caller's code.

Solution:
Provide an API for each functionality which requires platform specific code. Define only platform independent functions in the h-file and put all platform specific #ifdef code into the c-file. The caller of your functions only includes your h-file and does not have to include any platform specific files.

Try to design the API for the abstraction layer to be stable, because changing the API later on requires changes in your caller's code and sometimes that is not possible. However, it is very difficult to design a stable API. For platform abstractions, you can have a look at different platforms, maybe even platforms which you don't yet support. When understanding how these platforms work and what the differences are, you can create an API to abstract features for these platforms. Then you can be sure that, even when later on adding support for similar platforms, there is no need for changing the API.

Make sure to document the API well. Add comments to each function describing what the function does. Also describe on which platforms the functions are supported if that is not clearly defined elsewhere for your whole code-base.

Code Example

caller.c

```
#include "someFeature.h"

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

someFeature.h

```
/* Provides generic access to someFeature.
   Supported on platform A and platform B. */
void someFeature();
```

someFeature.c

```
void someFeature()
{
  #ifdef PLATFORM_A
    performFeaturePlatformA();
  #elif defined PLATFORM_B
    performFeaturePlatformB();
  #endif
}
```

Consequences:

The abstracted features can be used from anywhere in the code and not only from one single implementation file. In other words: Now you have distinct roles of caller and callee. The callee has to cope with platform variants and the caller can be platform independent.

On the up side, the caller does not have to cope with platform specific code. The caller simply includes the provided headerfile and does not have to include any platform specific headerfiles. However, on the down side, the caller might not be able to use platform-specific features anymore. Especially if the caller knows the platform-specific functions which are used below the abstraction layer and is used to these functions, then the caller might not be satisfied with using the abstracted functionality which might be different to use and which might not provide as much functionality as the platform-specific functions.

The platform-specific part can now be developed and even be tested separately from the other code. This now is the first time that even though you support many platforms, like multiple hardware and multiple operating systems, the testing effort is still manageable, because now you can mock the hardware-specific part in order to write simple tests for the platform-independent code..

When building up such APIs for all platform specific functions, then the sum of these functions and APIs is the platform abstraction layer for the code-base. When having such a platform abstraction layer, it is very clear which code is platform-dependent and which is platform independent. Such a platform abstraction layer also makes it clear, which parts of the code have to be touched in order to support an additional platform.

Known Uses:

- A hardware abstraction is used for the Time Triggered Ethernet protocol described in [Bunzel 2013]. The hardware abstraction layer contains functions for accessing interrupts and timers. The functions are marked as `inline` to not loose performance.
- The function `sock_addr_inet_pton` of the lighttpd web-server converts an IP address from text to binary form. The implementation distinguishes with #ifdef statements between code variants for IPv4 and IPv6. Callers of the API do not see this distinction.
- The function `getprogname` of the gzip data compression program returns the name of the invoking program. The way to obtain this name depends on the operating system and is distinguished via #ifdef statements in the implementation. The caller does not have to care on which operating system the function is called.

Running example:

Now you have a piece of code that might not anymore be considered ugly. Each of the functions only performs one single action and you hide implementation details about the variants behind APIs.

directoryNames.h

```
/* Copies the path to a new
   directory with name "newdir"
   located in the user's home
   directory into 'dirname'.
   Works on Linux and Windows. */
void getHomeDirectory(char* dirname);

/* Copies the path to a new
   directory with name "newdir"
   located in the current working
   directory into 'dirname'.
   Works on Linux and Windows. */
void getWorkingDirectory(char*
                         dirname);
```

directoryNames.c

```
#include "directoryNames.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void getHomeDirectory(char* dirname)
{
  #ifdef __unix__
    sprintf(dirname, "%s%s", getenv("HOME"),
            "/newdir/");
  #elif defined _WIN32
    sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"),
            getenv("HOMEPATH"), "\\newdir\\");
  #endif
}

void getWorkingDirectory(char* dirname)
{
  #ifdef __unix__
    strcpy(dirname, "newdir/");
  #elif defined _WIN32
    strcpy(dirname, "newdir\\");
  #endif
}
```

directorySelection.h

```
/* Copies the path to a new
   directory with name "newdir"
   into 'dirname'.
   The directory is located in the
   user's home directory, if
   STORE_IN_HOME_DIR is set
   or it is located in the current
   working directory, if
   STORE_IN_CWD is set. */
void getDirectoryName(char* dirname);
```

directorySelection.c

```
#include "directorySelection.h"
#include "directoryNames.h"

void getDirectoryName(char* dirname)
{
  #ifdef STORE_IN_HOME_DIR
    getHomeDirectory(dirname);
  #elif defined STORE_IN_CWD
    getWorkingDirectory(dirname);
  #endif
}
```

directoryHandling.h

```
/* Creates a new directory of the
   provided name ('dirname'). Works
   on Linux and Windows. */
void createNewDirectory(char*
                        dirname);
```

directoryHandling.c

```
#include "directoryHandling.h"
#ifdef __unix__
  #include <sys/stat.h>
#elif defined _WIN32
  #include <windows.h>
#endif

void createNewDirectory(char* dirname)
{
  #ifdef __unix__
    mkdir(dirname,S_IRWXU);
  #elif defined _WIN32
    CreateDirectory (dirname, NULL);
  #endif
}
```

main.c

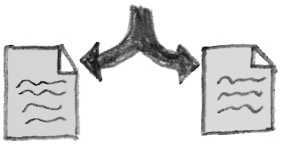
```
#include <stdio.h>
#include <string.h>
#include "directorySelection.h"
#include "directoryHandling.h"

int main()
{
  char dirname[50];
  char filename[50];
  char* my_data = "Write this data to the file";
  getDirectoryName(dirname);
  createNewDirectory(dirname);
  sprintf(filename, "%s%s", dirname, "newfile");
  FILE* f = fopen(filename, "w+");
  fwrite(my_data, 1, strlen(my_data), f);
  fclose(f);
  return 0;
}
```

Finally your file with the main program logic is completely independent from the operating system and not even operating system specific headerfiles are included here. Separating the implementation files with an ABSTRACTION LAYER on the one hand makes the files each by itself easier to comprehend and on the other hand makes it possible to reuse the functions in other parts of the code. Also development, maintenance, and testing can be split for the platform dependent and platform independent code.

When having many such ISOLATED PRIMITIVES behind an ABSTRACTION LAYER and organizing them according to the kind of variant which they abstract, then you'll end up with a hardware abstraction layer or with an operating system abstraction layer.

However, the implementations behind the APIs still contain #ifdef code for different variants. That has the disadvantage that these implementations have to be touched and grow if, for example, additional operating systems have to be supported. To avoid touching existing implementation files for adding another variant, you could could SPLIT VARIANT IMPLEMENTATIONS.

SPLIT VARIANT IMPLEMENTATIONS



Context:
You have platform variants hidden behind an ABSTRACTION LAYER. In the platform specific implementation you distinguish with #ifdef statements between the code variants.

Problem:
The platform specific implementations still contain #ifdef statements to distinguish between code variants. That makes it difficult to see and to select which part of the code should be built for which platform.

As code for different platforms is put into one single file, it is not possible to select the platform specific code on a file-basis. However, that is the approach taken by tools such as Makefiles, which are usually actually responsible to select which files should be compiled in order to come up with variants for different platforms.

When looking at the code from a high-level view, it is not possible to see which parts are platform specific and which are not. However, that would be very desirable when porting the code to another platform, to quickly see which code has to be touched.

The Open-Closed Principle **??** says that to bring in new features (or to port to a new platform), it should not be necessary to touch existing code. The code should be open for such modifications. However, having platform variants separated with #ifdef statements requires that existing implementations have to be touched for introducing a new platform, because simply another #ifdef branch has to be placed into an existing function.

Solution:
Put each variant implementation into a separate c-file and select on file-granularity what you want to compile for which platform.

Related functions of the same platform can sill be put into the same file. For example, there could be a file gathering all socket handling functions on Windows and one such file doing the same for Linux.

With separate files for each platform, #ifdef statements can still be used to determine which code is compiled on a specific platform. For example, a `fileHandlingWindows.c` file could still have on `#ifdef _WIN32` statement across the whole file. Alternatively, other platform-independent mechanisms such as Makefiles can be used to decide on a file-basis which code to compile on a specific platform.

With separate files for the platforms comes the question of where to put these files and how to name them:

- One option is to put platform-specific files per software-module next to each other and to name them in a way that makes it clear which platform they cover (e.g. `fileHandlingWindows.c`). That provides the advantage that the implementations of the software-modules are at the same place.

- Another option is to put all platform-specific files from the code-base into one directory and to have one subdirectory for each platform. That provides the advantage that all files for one platform are at the same place.

Code Example

someFeature.h

```
/* Provides generic access to
   someFeature. Supported on
   platform A and platform B. */
someFeature();
```

someFeatureWindows.c

```
#ifdef _WIN32
  someFeature()
  {
    performWindowsFeature();
  }
#endif
```

someFeatureLinux.c

```
#ifdef __unix__
  someFeature()
  {
    performLinuxFeature();
  }
#endif
```

Consequences:

You now have the option to not have any #ifdef statements at all in the code, but to instead distinguish between the variants on file-basis with tools such as Makefiles.

In each implementation file there is now just one code variant, so there is no need to jump between the lines when reading the code in order to only read the #ifdef branch you are looking for. It is much easier to read and understand the code.

When fixing a bug on one platform, no files for other platforms have to be touched. When porting to a new platform, only new files have to be added and no existing file or existing code has to be modified.

It is easy to spot which part of the code is platform-dependent and which code has to be added in order to port to a new platform. Either all platform specific files are in one single directory, or the files are named in a way that makes clear they are platform-dependent.

Known Uses:

- The Simple Audio Library [Hook 2005] uses separate implementation files to provide access to threads and mutexes for Linux and OS-X. The implementation files use #ifdef statements to only compile the code on the corresponding operating system.
- The Multi-Processing-Module of the Apache web-server, which is responsible for handling accesses to the web-server, is implemented in separate implementation files for Windows and Linux. The implementation files use #ifdef statements to only compile the code on the corresponding operating system.
- The code of the uboot bootloader puts the source code for each hardware platform it supports into a separate directory. Each of these directories contains amongst others the file `cpu.c` which contains a function to reset the CPU. A Makefile decides which directory (and which `cpu.c` file) has to be compiled - there are no #ifdef statements in these files. The main program logic of uboot calls the function to reset the CPU and does not have to care about hardware platform details at that point.

Running example:

After SPLITTING VARIANT IMPLEMENTATIONS, you'll end up with the following final code for your functionality to create a directory and to write data to a file:

directoryNames.h

```
/* Copies the path to a new
   directory with name "newdir"
   located in the user's home
   directory into 'dirname'.
   Works on Linux and Windows. */
void getHomeDirectory(char* dirname);

/* Copies the path to a new
   directory with name "newdir"
   located in the current working
   directory into 'dirname'.
   Works on Linux and Windows. */
void getWorkingDirectory(char*
                         dirname);
```

directoryNamesLinux.c

```
#ifdef __unix__
  #include "directoryNames.h"
  #include <string.h>
  #include <stdio.h>
  #include <stdlib.h>

  void getHomeDirectory(char* dirname)
  {
      sprintf(dirname, "%s%s", getenv("HOME"),
              "/newdir/");
  }

  void getWorkingDirectory(char* dirname)
  {
    strcpy(dirname, "newdir/");
  }
#endif
```

directoryNamesWindows.c

```
#ifdef _WIN32
  #include "directoryNames.h"
  #include <string.h>
  #include <stdio.h>
  #include <windows.h>

  void getHomeDirectory(char* dirname)
  {
    sprintf(dirname, "%s%s%s", getenv("HOMEDRIVE"),
            getenv("HOMEPATH"), "\\newdir\\");
  }

  void getWorkingDirectory(dirname)
  {
    strcpy(dirname, "newdir\\");
  }
#endif
```

directorySelection.h

```
/* Copies the path to a new
   directory with name "newdir"
   into 'dirname'.
   The directory is located in the
   user's home directory, if
   STORE_IN_HOME_DIR is set
   or it is located in the current
   working directory, if
   STORE_IN_CWD is set. */
void getDirectoryName(char* dirname);
```

directorySelectionHomeDir.c

```
#ifdef STORE_IN_HOME_DIR
  #include "directorySelection.h"
  #include "directoryNames.h"

  void getDirectoryName(char* dirname)
  {
    getHomeDirectory(dirname);
  }
#endif
```

directorySelectionWorkingDir.c

```
#ifdef STORE_IN_CWD
  #include "directorySelection.h"
  #include "directoryNames.h"

  void getDirectoryName(char* dirname)
  {
    return getWorkingDirectory(dirname);
  }
#endif
```

directoryHandling.h

```
/* Creates a new directory of the
   provided name ('dirname'). Works
   on Linux and Windows. */
void createNewDirectory(char*
                        dirname);
```

directoryHandlingLinux.c

```
#ifdef __unix__
  #include <sys/stat.h>

  void createNewDirectory(char* dirname)
  {
    mkdir(dirname,S_IRWXU);
  }
#endif
```

directoryHandlingWindows.c

```
#ifdef _WIN32
  #include <windows.h>

  void createNewDirectory(char* dirname)
  {
    CreateDirectory(dirname, NULL);
  }
#endif
```

main.c

```
#include "directorySelection.h"
#include "directoryHandling.h"
#include <string.h>
#include <stdio.h>

int main()
{
  char dirname[50];
  char filename[50];
  char* my_data = "Write this data to the file";
  getDirectoryName(dirname);
  createNewDirectory(dirname);
  sprintf(filename, "%s%s", dirname, "newfile");
  FILE* f = fopen(filename, "w+");
  fwrite(my_data, 1, strlen(my_data), f);
  fclose(f);
  return 0;
}
```

In the code above, there are still #ifdef statements present. Each of the implementations files has one huge #ifdef in order to decide whether the whole code in the file should be compiled. Alternatively, the decision which files should be compiled could be put into a Makefile. The decision whether to use the home- or the working-directory could even be made a runtime decision. Both are ways to get rid of the #ifdefs. Both are ways to simply use another mechanism to chose between features. However, deciding which mechanism to use is not so important. Much more important, as described throughout this paper, is to isolate and abstract the #ifdefs.

While the code-files would look cleaner when using such other mechanisms to handle the variants, the complexity would simple be there - it would simply be somewhere else. Putting the complexity into Makefiles can be a good idea, because the purpose of Makefiles is to decide which files to build. However, if for example for building the operating system specific code, instead of platform-independent makefiles, a proprietary IDE for Windows and another IDE for Linux is used to decide which files to build, then simply using the solution shown above with #ifdef statements in the code is much cleaner, because the configuration which files should be built for which operating system is only done once.

The final code of the running example above showed very clearly, how code with operating system specific variants or other variants can be improved step by step. Compared to the first code example, this final piece of code is well readable and can easily be extended with additional features or can easily be ported to additional operating systems without requiring to touch any of the existing code.

CONCLUSION

This paper presented pattern on how to handle variants, like hardware or operating system variants, in C code and on how to organize and get rid of #ifdef statements.

The AVOID VARIANTS pattern suggests to use standardized functions instead of self-implemented variants. This pattern should be applied anytime it is applicable, because it resolved issues with code variants in one blow. However, there is not always a standardized function available and in such cases, the programmer has to implement his or her own function to abstract the variant. As a start, ISOLATE PRIMITIVES suggests to put variants into separate functions and ATOMIC PRIMITIVES suggests to only handle one kind of variant in such functions. ABSTRACTION LAYER takes the additional step to hide the implementations of the primitives behind an API and SPLIT VARIANT IMPLEMENTATIONS suggests to put each variant into a separate C-file.

With these patterns as part of the programming vocabulary, a C programmer has a toolbox and has step-by-step guidance on how to tackle C code variants in order to structure code and in order to escape from the #ifdef hell.

For experienced programmers, some of the patterns might look like very obvious solutions. That is good. One of the tasks of patterns is to educate people to do the right thing and once people know how to do the right thing, the patterns are not necessary anymore, because people then intuitively do it as suggested by the patterns. However, until everybody gets there, the pattens presented in this paper provide valuable insights on how to structure code variants.

This paper is part of a series of papers on C programming [Preschern 2015][Preschern 2016][Preschern 2017][Preschern 2018a][Preschern 2018b][Preschern 2019]. This series of papers is the start of a collection of hands-on best practices for the C programming language.

ACKNOWLEDGMENTS

I want to thank my shepherd...

REFERENCES

Flemming Bunzel. 2013. Hardware-abstraction of an open source real-time Ethernet stack - Design, realisation and evaluation. (2013).

Brian Hook. 2005. *Write Portable Code: An Introduction to Developing Software for Multiple Platforms*. No Starch Press.

Didier Malenfant. 2000. Writing Portable Code. In *GDC*.

Christopher Preschern. 2015. Idioms for Error Handling in C. In *Proceedings of the 20th European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2016. API Patterns in C. In *Proceedings of the 21st European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2017. Patterns for C Iterator Interfaces. In *Proceedings of the 22nd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2018a. C Patterns on Objects and their Lifetime. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2018b. Patterns for Returning Data from C Functions. In *Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP)*.

Christopher Preschern. 2019. Patterns for Returning Error Information in C. In *Proceedings of the 24th European Conference on Pattern Languages of Programming (EuroPLoP)*.

Henry Spencer and Geoff Collyer. 1992. #ifdef Considered Harmful, or Portability Experience With C News. In *Proceedings of the 1992 USENIX Conference*.