Idiotproofing Outline

1. Idiotproofing Outline
2. Idiotproofing
3. Idiotproofing Quotes
4. Bear Spray NOT Like Bug Spray
5. An Idiotproof Website
6. Idiotproofing Example #1
7. Idiotproofing Example #2
8. Idiotproofing Example #3
9. Idiotproofing Example #4
10. Why We Idiotproof
11. The exit Statement #1
12. The exit Statement #2
13. The exit Statement #3
14. The exit Statement #4
15. exit Example’s Flowchart
16. A New File to #include
17. exit Statement Inside an if Block #1
18. exit Statement Inside an if Block #2
19. exit Statement Inside an if Block #3
20. exit Statement Inside an if Block #4
21. Idiotproofing Example’s Flowchart
Idiotproofing means ensuring that a user’s input is valid.
“Idiotproofing is difficult because idiots are so clever.”

“You can't make anything idiot proof because idiots are so ingenious.”

— Ron Burns

“Idiotproofing causes evolutionary selection of more ingenious idiots.”

“Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.”  – Rich Cook

“It doesn’t really matter what effort you put into idiot-proofing a product or procedure. They will always build a better idiot.”

“Idiot-proofing assumes a finite number of idiots.”

“Campaigns to bearproof all garbage containers in wild areas have been difficult because, as one biologist put it, ‘There is a considerable overlap between the intelligence levels of the smartest bears and the dumbest tourists’.”

http://www.goodreads.com/quotes/tag/idiots
http://scienceblogs.com/goodmath/2008/04/the_real_murphys_law.php
Bear Spray NOT Like Bug Spray

Listen,
bear spray DOES NOT work like bug spray.

We would like to not have to say that again.

8:23 AM · May 23, 2022 · Twitter Web App

29.6K Retweets 4,146 Quote Tweets 264.7K Likes
An Idiotproof Website

http://www.idiotproofwebsite.com/
Idiotproofing Example #1

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

int main ()
{ /* main */
    const int minimum_depth = 0;
    const int program_success_code = 0;
    const int program_failure_code = -1;
    float depth_in_fathoms;
```
printf("What is the depth in fathoms?\n");
scanf("%f", &depth_in_fathoms);
if (depth_in_fathoms < minimum_depth) {
    printf("ERROR: you can't have a negative\n");
    printf(" depth in fathoms %f!\n", depth_in_fathoms);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (depth_in_fathoms < ...) */
Idiotproofing Example #3

/*
 * ASSERT: By the time the program gets to here,
 * the depth in fathoms must be valid.
 */
printf("The depth in fathoms is valid.\n");
return program_success_code;
} /* main */
% gcc -o conversions_idiot conversions_idiot.c
% conversions_idiot
What is the depth in fathoms?
-100
ERROR: you can't have a negative depth in fathoms -100.0000!
% conversions_idiot
What is the depth in fathoms?
100
The depth in fathoms is valid.
Why We Idiotproof

- Idiotproofing ensures that input data are valid, which means that, if our program is otherwise correct, then the output will be valid as well.
- Idiotproofing allows us to assert certain properties of the data. For example, in the conversions program, properly idiotproofed input data allow us to assert that, in the calculation section, the depth in fathoms is valid. So, our calculations can assume this fact, which sometimes can be more convenient.
The `exit` statement terminates execution of a given run of a program.
The `exit` Statement #2

```c
% cat exitexample.c
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    const int program_failure_code = -1;
    printf("This statement will be always be executed.\n");
    exit(program_failure_code);
    printf("This statement will be never be executed.\n");
} /* main */
% gcc -o exitexample exitexample.c
% exitexample
This statement will be always be executed.

The program terminates in a controlled, graceful way – that is, it doesn’t actually crash – without executing the remaining executable statements.
```
The `exit` Statement #3

```c
% cat exitexample.c
#include <stdio.h>
#include <stdlib.h>

int main ()
{ /* main */
    const int program_failure_code = -1;
    printf("This statement will be always be executed.\n");
    exit(program_failure_code);
    printf("This statement will be never be executed.\n");
} /* main */
% gcc -o exitexample exitexample.c
% exitexample
This statement will be always be executed.

Notice that the `exit` statement takes an `int` argument.
This argument represents the value that will be returned by
the program to the operating system (for example, Linux).
By convention, returning 0 from a program to the OS means
that the program completed successfully, so if the program is
exiting prematurely, then you should return a non-zero value.
% cat exitexample.c
#include <stdio.h>
#include <stdlib.h>

int main ()
{ /* main */
    const int program_failure_code = -1;

    printf("This statement will be always be executed.\n");
    exit(program_failure_code);
    printf("This statement will be never be executed.\n");
} /* main */

% gcc -o exitexample exitexample.c
% exitexample
This statement will be always be executed.

**Jargon**: In the context of running a program, all of the following terms are generally used to mean the same thing: exit, stop, halt, terminate, abort.
printf("This statement will be always be executed.\n");
exit(program_failure_code);
printf("This statement will be never be executed.\n");

Notice that the symbol for an exit is also an oval.
To use an `exit` statement, you **MUST** include an additional header file, **IMMEDIATELY AFTER** `stdio.h`:

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

This statement will be always be executed.
if (depth_in_fathoms < minimum_depth) {
    printf("ERROR: you can't have a negative\n");
    printf(" depth in fathoms %f!\n",
        depth_in_fathoms);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (depth_in_fathoms < ...) */

When you put an `exit` statement inside an `if` block, the `exit` statement will be executed only in the event that the appropriate clause of the `if` block is entered, and then only after all prior statements in that clause of the `if` block have already been executed.
In the above example, the `exit` statement is executed only in the event that the depth in fathoms is negative, and only after executing the `printf` statement that precedes it.
exit Statement Inside an if Block #3

```c
if (depth_in_fathoms <
    minimum_depth) {
    printf("ERROR: you can't have a negative");
    printf(" depth in fathoms %f!\n",
            depth_in_fathoms);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (depth_in_fathoms < ...) */
```

Notice that the `exit` statement **DOESN’T** have to have a comment after it.
if (depth_in_fathoms < minimum_depth) {
    printf("ERROR: you can’t have a negative");
    printf(" depth in fathoms %f!\n", depth_in_fathoms);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (depth_in_fathoms < ...) */

Notice that the `exit` statement is inside the `if` block and therefore is indented **MORE** than the `if` statement.
if (depth_in_fathoms < minimum_depth) {
    printf("ERROR: you can’t have a negative");
    printf(" depth in fathoms %f\n", depth_in_fathoms);
    exit(program_failure_code);
} /* if (depth_in_fathoms < ...) */