Idiotproofing Outline

1. Idiotproofing Outline
2. Idiotproofing
3. Idiotproofing Quotes
4. An Idiotproof Website
5. Idiotproofing Example #1
6. Idiotproofing Example #2
7. Idiotproofing Example #3
8. Idiotproofing Example #4
9. Why We Idiotproof
10. The `exit` Statement #1
11. The `exit` Statement #2
12. The `exit` Statement #3
13. The `exit` Statement #4
14. `exit` Example’s Flowchart
15. A New File to `#include`
16. `exit` Statement Inside an `if` Block #1
17. `exit` Statement Inside an `if` Block #2
18. `exit` Statement Inside an `if` Block #3
19. `exit` Statement Inside an `if` Block #4
20. Idiotproofing Example’s Flowchart
Idiotproofing

*Idiotproofing* means ensuring that a user’s input is valid.
Idiotproofing Quotes

“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
Idiotproofing Example #1

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

int main ()
{
    /* main */
    const float minimum_area = 0;
    const int program_success_code = 0;
    const int program_failure_code = -1;
    float area_in_square_miles;
}
```
Idiotproofing Example #2

```c
printf("What is the area ");
printf("in square miles?\n");
scanf("%f", &area_in_square_miles);
if (area_in_square_miles < minimum_area) {
    printf("ERROR: you can’t have a negative");
    printf(" area!\n");
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (area_in_square_miles < minimum_area) */
```
Idiotproofing Example #3

```c
/*
 * ASSERT: By the time the program gets to here, 
 * the area must be valid.
 */
printf("The area is valid.\n");
return program_success_code;
} /* main */
```
Idiotproofing Example #4

% gcc -o conversions_idiot conversions_idiot.c
% conversions_idiot
What is the area in square miles?
-100
ERROR: you can't have a negative area!
% conversions_idiot
What is the area in square miles?
100
The area 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 area in square miles is valid. So, our calculations can assume this fact, which sometimes can be more convenient.
The **exit** Statement #1

```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.

The **exit** statement terminates execution of a given run of a program.
The exit Statement #2

```
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 */
```

NOTICE!

`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.
% 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.
The `exit` Statement #4

% 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>
```
exit Statement Inside an if Block #1

```c
if (area_in_square_miles < minimum_area) {
    printf("ERROR: you can't have a negative");
    printf(" area!\n");
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (area_in_square_miles < minimum_area) */
```

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.
exit Statement Inside an if Block #2

if (area_in_square_miles < minimum_area) {
    printf("ERROR: you can’t have a negative");
    printf(" area!\n");
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (area_in_square_miles < minimum_area) */

In the above example, the exit statement is executed only in the event that the area is negative, and only after executing the printf statement that precedes it.
exit Statement Inside an if Block #3

if (area_in_square_miles < minimum_area) {
    printf("ERROR: you can’t have a negative");
    printf(" area!\n");
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (area_in_square_miles < minimum_area) */

Notice that the exit statement DOESN’T have to have a comment after it.
Exit Statement Inside an if Block #4

if (area_in_square_miles < minimum_area) {
    printf("ERROR: you can't have a negative");
    printf(" area!\n");
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (area_in_square_miles < minimum_area) */

Notice that the exit statement is inside the if block and therefore is indented MORE than the if statement.
if (area in square miles < minimum area) {
    printf("ERROR: you can’t have a negative");
    printf(" area!\n");
    exit(program_failure_code);
} /* if (area_in_square_miles < minimum_area) */