Idiotproofing Outline

1. Idiotproofing Outline
2. Idiotproofing
3. Idiotproofing Example Program
4. Idiotproofing Example Run
5. Why We Idiotproof
6. The `exit` Statement
7. `exit` Example's Flowchart
8. Putting an `exit` Statement Inside an `if` Block
9. Idiotproofing Example's Flowchart

Idiotproofing

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

Quotes

“Idiotproofing is difficult because idiots are so clever.”

“Idiotproofing causes evolutionary selection of more ingenious idiots.”

“Computer Scientists are constantly creating better fool proof software. The Universe is constantly creating better fools. So far, the Universe is winning.”

“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.”
Idiotproofing Example Program

```c
#include <stdio.h>

int main ()
{
    /* main */
    const int no_chickens = 0;
    const int error_exit_code = -1;
    int number_of_chickens, number_of_brown_chickens;

    printf("How many chickens did you think of?\n");
    scanf("%d", &number_of_chickens);
    if (number_of_chickens < no_chickens) {
        printf("ERROR: you can't think of\n");
        exit(error_exit_code); /* <--- NOTICE! */
    }
    /* if (number_of_chickens < no_chickens) */
    printf("How many of the %d chickens were brown?\n", number_of_chickens);
    scanf("%d", &number_of_brown_chickens);
    if (number_of_brown_chickens < no_chickens) {
        printf("ERROR: you can't think of\n");
        exit(error_exit_code); /* <--- NOTICE! */
    }
    /* if (number_of_brown_chickens < no_chickens) */
    if (number_of_brown_chickens > number_of_chickens) {
        printf("ERROR: you can't think of more brown\n");
        exit(error_exit_code); /* <--- NOTICE! */
    }
    /* if (number_of_brown_chickens > ...) */
    /* ASSERT: By the time the program gets to here, */
    /* the number of chickens and the number */
    /* of brown chickens must be valid. */
    printf("The number of chickens and the number of\n");
    printf(" chickens that are brown are both valid.\n");
} /* main */
```

Idiotproofing Example Run

```
% gcc -o chickenidiot chickenidiot.c
% chickenidiot
How many chickens did you think of?
-1
ERROR: you can't think of negative chickens!
% chickenidiot
How many chickens did you think of?
2
How many of the 2 chickens were brown?
-3
ERROR: you can't think of negative brown chickens!
% chickenidiot
How many chickens did you think of?
2
How many of the 2 chickens were brown?
5
ERROR: you can't think of more brown chickens than total chickens!
% chickenidiot
How many chickens did you think of?
2
How many of the 2 chickens were brown?
1
The number of chickens and the number of chickens that are brown are both 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 chickens program, properly idiotproofed input data allow us to assert that, in the calculation section, the number of brown chickens does not exceed the total number of chickens. So, our calculations can assume this fact, which sometimes can be more convenient.

The exit Statement

The `exit` statement terminates execution of a given run of a program. For example:

```c
#include <stdio.h>

int main ()
{ /* main */
  const int error_exit_code = -1;
  printf("This statement will be always be executed.\n");
  exit(error_exit_code);
  printf("This statement will be never be executed.\n");
} /* main */
```

The program terminates in a controlled, graceful way — that is, it doesn’t actually crash — without completely the remaining executable statements.

Notice that the exit statement takes an integer argument. This argument represents the value that will be returned by the program to the operating system (e.g., 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.

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.
**exit** Example’s Flowchart

```c
printf("This statement will always be executed.\n")
exit(error_exit_code);
printf("This statement will never be executed.\n")
```

**Apparent Flowchart**

```
Start

Output always.

Exit

Output never.

End
```

**Actual Flowchart**

```
Start

Output always.

Exit
```

Notice that the symbol for a **exit** is also an oval.

---

**Putting an **exit** Statement Inside an **if** Block**

When you put a **exit** statement inside an **if** block, the **exit** statement will only be executed in the event that the **if** block is entered, and then only after all prior statements in the **if** block have already been executed.

```c
if (number_of_chickens < no_chickens) {
    printf("ERROR: you can’t think of’");
    printf(" negative chickens!\n");
    exit(error_exit_code);
} /* if (number_of_chickens < no_chickens) */
```

In the above example, the **exit** statement is executed only in the case that the number of chickens is negative, and only after executing the **printf** statement that precedes it.

By the way, notice that the **exit** statement does not have to be surrounded by blank lines and does not have to have a comment after it.
Idiotproofing Example’s Flowchart

if (number_of_chickens < no_chickens) {
    printf("ERROR: you can’t think of");
    printf("negative chickens!\n");
    exit(error_exit_code);
} /* if (number_of_chickens < no_chickens) */

Notice that the `exit` statement is inside the `if` block and is indented MORE than the `if` statement.