# Idiotproofing Outline



# 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

http://c2.com/cgi/wiki?IdiotProofProcess

# Bear Spray NOT Like Bug Spray

Oklahoma Department of Wildlife Conservation
@OKWildlifeDept

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

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{ /* main */
    const int minimum_volume       =  0;
    const int program_success_code =  0;
    const int program_failure_code = -1;
    float volume_in_fluid_ounces;

    printf("What is the volume in fluid ounces?\n");
    scanf("%f", &volume_in_fluid_ounces);
    if (volume_in_fluid_ounces <
        minimum_volume) {
        printf("ERROR: you can't have a negative");
        printf(" volume %f!\n",
            volume_in_fluid_ounces);
        exit(program_failure_code); /* <--- NOTICE! */
    } /* if (volume_in_fluid_ounces < ...) */
   /*
    * ASSERT: By the time the program gets to here,
    * the volume in fluid ounces must be valid.
    */
    printf("The volume in fluid ounces is valid.\n");
    return program_success_code;
} /* main */
```

# Idiotproofing Example #2

```
% gcc -o conversions_idiot conversions_idiot.c
% conversions_idiot
What is the volume in fluid ounces?
-1000
ERROR: you can't have a negative volume -1000.0000!
% conversions_idiot
What is the volume in fluid ounces?
1000
The volume in fluid ounces 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 volume in fluid ounces is valid.

So, our calculations can assume this fact,
which sometimes can be more convenient.

# The `exit` Statement #1

```
% 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!**

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

# The `exit` Statement #2

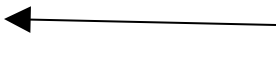
```
% cat exitexample.c
#include <stdio.h>
#include <stdlib.h>

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

**NOTICE!**

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

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



**NOTICE!**

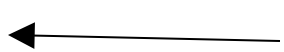
```
int main ()
{ /* main */
    const int program_failure_code = -1;

    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>
```



**NOTICE!**

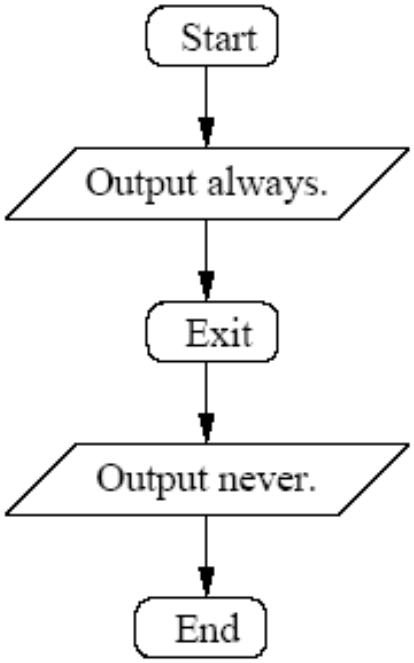
```
int main ()
{ /* main */
    const int program_failure_code = -1;

    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 used to mean the same thing:
exit, stop, halt, terminate, abort.

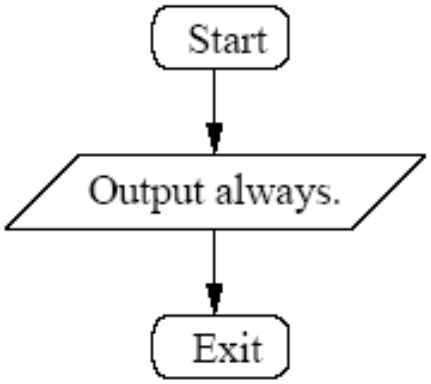
# `exit` Example's Flowchart

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

**Apparent Flowchart**



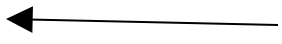

**Actual Flowchart**




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

# A New File to `#include`

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



**NOTICE!**

```
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.
```

To use an `exit` statement, you **MUST** include an additional header file, **IMMEDIATELY AFTER** `stdio.h`:

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

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

```
if (volume_in_fluid_ounces <
    minimum_volume) {
    printf("ERROR: you can't have a negative");
    printf(" volume %f!\n",
        volume_in_fluid_ounces);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (volume_in_fluid_ounces < ...) */
```

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 (volume_in_fluid_ounces <
    minimum_volume) {
    printf("ERROR: you can't have a negative");
    printf(" volume %f!\n",
        volume_in_fluid_ounces);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (volume_in_fluid_ounces < ...) */
```

In the above example, the `exit` statement is executed only in the event that the volume in fluid ounces is negative, and only after executing the `printf` statement that precedes it.

# `exit` Statement Inside an `if` Block #3

```
if (volume_in_fluid_ounces <
    minimum_volume) {
    printf("ERROR: you can't have a negative");
    printf(" volume %f!\n",
        volume_in_fluid_ounces);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (volume_in_fluid_ounces < ...) */
```

Notice that the `exit` statement

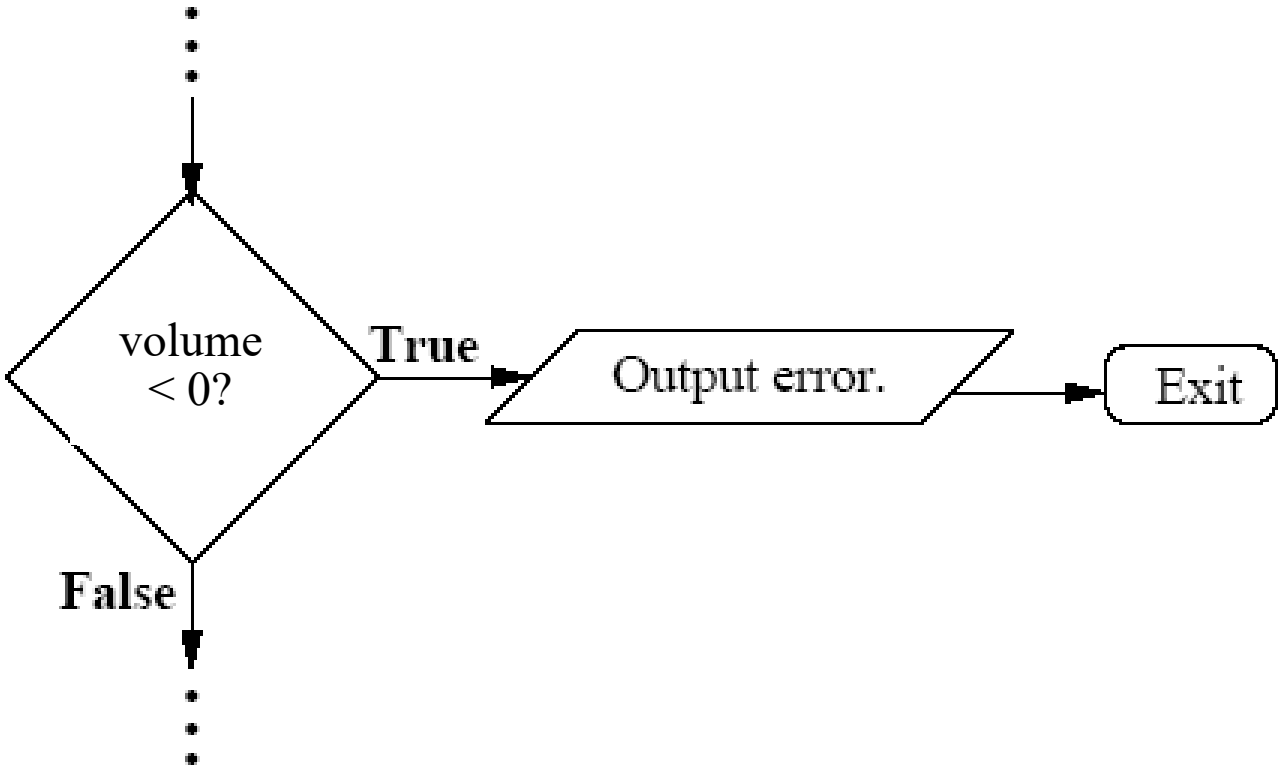
**DOESN'T** have to have a comment after it.

# `exit` Statement Inside an `if` Block #4

```
if (volume_in_fluid_ounces <
    minimum_volume) {
    printf("ERROR: you can't have a negative");
    printf(" volume %f!\n",
        volume_in_fluid_ounces);
    exit(program_failure_code); /* <--- NOTICE! */
} /* if (volume_in_fluid_ounces < ...) */
```

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

# Idiotproofing Example's Flowchart




```
if (volume_in_fluid_ounces <
    minimum_volume) {
    printf("ERROR: you can't have a negative");
    printf(" volume %f!\n",
        volume_in_fluid_ounces);
    exit(program_failure_code);
} /* if (volume_in_fluid_ounces < ...) */
```