

Compile-Time Errors



3

- Code does not conform to C specification
 - Forgetting a semicolon
 - Forgetting to declare a variable
 - etc.
- Detected by compiler

```
int a = 0;
int b = 3
int c = 6;
a = b + 3;
d = c + 3;
cc-1020 cc: ERROR File = foo.c, Line = 6
The identifier "d" is undefined.
d = c + 3;
```

Link-Time Errors



```
• Error in linking together the .o files to make an a.out
        Symbol referenced (used) in one module, not defined in another
    extern int not_there;
        .
        .
        main() {
        printf("%d", not_there);
        }
        Undefined first referenced
        symbol in file
        not_there foo.o
        ld: fatal: Symbol referencing errors.
        No output written to a.out
```

Run-Time User Errors



- · User provides invalid input
 - $\,\circ\,$ User types in name of file that does not exist
 - $\circ\,$ User provides program argument with value outside legal bounds
- Detected with "if" checks in program
 - $\circ\,$ Program should print message and recover gracefully
 - $\circ~\ensuremath{\mathsf{Possibly}}$ ask user for new input
- Your program should anticipate and handle EVERY possible user input

int ReadFile(const char *filename)
{
 FILE *fp = fopen(filename, "r");
 if (!fp) {
 fprintf(stderr, "Unable to open file: %s\n", filename);
 return 0;
 }
}

Run-Time Program Errors



5

- Internal error from which recovery is impossible (bug)
 - $\circ~$ Null pointer passed to ${\tt Array_getData()}$
 - $\,\circ\,$ Invalid value for array index (k = -7)
 - Invariant is violated
 - etc.
- Detected with conditional checks in program
 - Program should print message and abort

void Array_getData(Array_T array, int k)
{
 return array->elements[k];

Run-Time Program Errors



• What errors can this function encounter?

void Array_getData(Array_T array, int k)

return array->elements[k];

Run-time Exceptions



- Rare error from which recovery may be possible
 - User hits interrupt key
 - Arithmetic overflow
 - etc.
- Detected by machine or operating system
 - Program can handle them with signal handlers (later)
 - Not usually possible/practical to detect with conditional checks

.

Robust Programming



- Your program should never terminate without either ...
 - Completing successfully, or
 - Outputing a meaningful error message
- How can a program terminate?
 - Return from main
 - Call exit
 - Call abort

Robust Programming

- Your program should never terminate without either ...
 - Completing successfully, or
 - Outputing a meaningful error message
- How can a program terminate?
 - > Return from main
 - Call exit
 - Call abort

#include <stdio.h></stdio.h>
#include "stringarray.h"
int main()
{
<pre>StringArray_T stringarray = StringArray_new()</pre>
<pre>StringArray_read(stringarray, stdin);</pre>
<pre>StringArray_sort(stringarray, strcmp);</pre>
<pre>StringArray_write(stringarray, stdout);</pre>
<pre>StringArray_free(stringarray);</pre>
return 0;
}

Robust Programming



9

11

- Your program should never terminate without either ...
 - Completing successfully, or
 - Outputing a meaningful error message
- How can a program terminate?
 - Return from main
 - > Call exit
 - Call abort

<pre>#include <stdlib.h></stdlib.h></pre>
<pre>void ParseArguments(int argc, char **argv) {</pre>
<pre>argc; argv++;</pre>
while (argc > 0) {
<pre>if (!strcmp(*argv, "-filename")) {</pre>
•••
}
else if (!strcmp(*argv, "-help")) {
<pre>PrintUsage();</pre>
exit(0);
}
else {
<pre>fprintf(stderr, "Unrecognized argument: %s\n", *argv); PrintUsage(); exit(1);</pre>
}
argv++; argc;
}
}





- Your program should never terminate without either ...
 - Completing successfully, or
 - Outputing a meaningful error message
- How can a program terminate?

С	Return	from	main
	0.11	••	

- Call exit
- > Call abort

```
#include <stdlib.h>
void *Array_getData(Array_T array, int k)
{
    if (!array) {
        fprintf(stderr, "array=NULL in Array_getData\n");
        abort();
    }
    if ((k < 0) || (k >= array->nelements)) {
        fprintf(stderr, "k=%d in Array_getData\n", k);
        abort();
    }
    return array->elements[k];
}
```



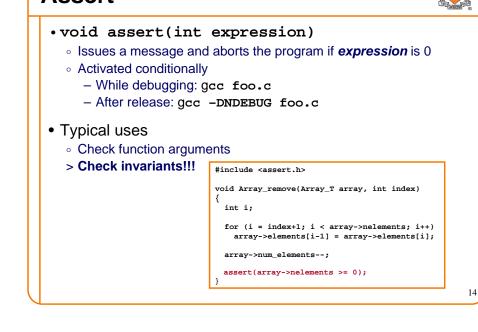
Assert



- •void assert(int expression)
 - Issues a message and aborts the program if expression is 0
 - Activated conditionally
 - While debugging: gcc foo.c
 - After release: gcc -DNDEBUG foo.c
- Typical uses
 - > Check function arguments
 - Check invariants!!!

#include <assert.h>
void *Array_getData(Array_T array, int k)
{
 assert(array);
 assert((k >= 0) && (k < array->nelements));
 return array->elements[k];
}

Assert



Assert



15

13

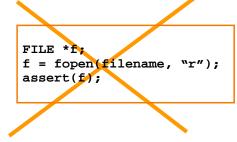
- •void assert(int expression)
 - Issues a message and aborts the program if expression is 0
 - Activated conditionally
 - While debugging: gcc foo.c
 - After release: gcc -DNDEBUG foo.c
- Typical uses
 - · Check function arguments
 - Check invariants!!!

as	s	er	t	•	ł





- Assert is meant for <u>bugs</u>, conditions that "can't" occur (or if they do, it's the programmer's fault)
 - File-not-present happens all the time, beyond the control of the programmer
 - Asserts shouldn't trigger during the execution of a working program
 - Instead of an assert, print a nice error message to the user, then exit or retry



Robust Programming Summary



- Programs encounter errors
 - Good programmers handle them gracefully
- Types of errors
 - Compile-time errors
 - Run-time user errors
 - Run-time program errors
 - Run-time exceptions
- Robust programming
 - $\circ~$ Complete successfully, or
 - Output a meaningful error message

Different execution times

- 1. Preprocessing time
- 2. Compile time
- 3. Link time
- 4. Run time

Debugging

• Bug

b. A defect or fault in a machine, plan, or the like. orig. U.S.
1889 Pall Mall Gaz. 11 Mar. 1/1 Mr. Edison, I was informed, had been up the two previous nights discovering 'a bug' in his phonograph an expression for solving a difficulty, and implying that some imaginary insect has secreted itself inside and is causing all the trouble.

Oxford English Dictionary, 2nd Edition

- Debugging is backward reasoning
 - $\circ\,$ Like solving mysteries, think backwards from the results to reasons
 - Most problems are our own faults

Easy Bugs



17

 Look for familiar patterns int n; scanf("%d", n);

if (x & y == 0)...

- Examine the most recent change
 - If previous version is correct, check the differences
 - Version control is helpful
- Don't make the same mistake twice

switch (argv[i][1]) {
 case `o':
 outname = argv[i]; break;
case `f':
 from = atoi(argv[i]); break;

% program -o file.txt -f 20

19

Good Disciplines



- Debug now, don't wait
 - $\,\circ\,$ Bug will show up later and it will become harder to fix over time
- Get a stack trace
 - Probably the most useful function of a debugger
- Read before typing
 - $\circ\,$ "Read and think" is often better than "type and try."
 - $\circ~$ Take a break for a while
- Do a good, old flowchart
 - $\circ\,$ The technique works at all levels
- Explain your code to someone
 - Rethink through your code

Hard Bugs



- Make the bug reproducible
 - Construct input and settings
 - Or, try to understand why not reproducible
- Divide and conquer
 - Binary search is fast
- Display output to localize your search
 - $\circ~$ You will have to be selective
- Log the events
 - Useful for long running programs
- Use tools
 - $\circ~$ Compare and visualize the results
- "Defensive programming"
 - E.g.: lots of asserts

Summary of Debugging



21

- Solving a puzzle
- Hard thinking is the best first step
- Explain your code to someone else
- Reproducing bugs is the key
- Make mistakes fast and don't make them again

Hard Bugs

- Remember many languages are very forgiving
 - C's type checking is not strong
- · Caused by your own faults
 - Uninitialized variables
 - Global variables
 - $\circ~$ Use freed memory, access memory out of bounds
- Other people's bugs
 - Read another program is challenging
 - $\circ\,$ Learn testing to find bugs without source code
- Infrequent causes
 - Library code
 - Compiler optimizations
 - Hardware

