Robust Programming

CS 217

Program Errors

- 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

Compile-Time Errors

- 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-1065 cc: ERROR File = foo.c, Line = 2
  A semicolon is expected at this point.

    int c = 6;
    ^

cc-1020 cc: ERROR File = foo.c, Line = 6
  The identifier "d" is undefined.

    d = c + 3;
    ^
```

Run-Time User Errors

- User provides invalid input
  - User types in name of file that does not exist
  - User provides program argument with value outside legal bounds
  - etc.
- Detected with “if” checks in program
  - Program should print message and recover gracefully
  - 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

- Internal error from which recovery is impossible (bug)
  - Null pointer passed to **`Array_removeLast()`**
  - Invalid value for array index (k = -7)
  - Invariant is violated
  - etc.
- Detected with conditional checks in program (assert)
  - Program should print message and abort

```
#include <assert.h>

void Array_removeLast(Array_T oArray)
{
  assert(oArray);
  oArray->nelements--;
}
```

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

```
#include <limits.h>
...
int a = MAX_INT;
int b = MAX_INT;
int c = 6;
int d = 0;
...
a = a + d;
d = a + b;
b = a - c;
...
```

Robust Programming

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

```
#include <stdio.h>
#include “stringarray.h”

int main()
{
  StringArray_T stringarray = StringArray_new();

  StringArray_read(stringarray, stdin);
  StringArray_sort(stringarray, strcmp);
  StringArray_write(stringarray, stdout);

  StringArray_free(stringarray);

  return 0;
}
```

Robust Programming

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

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

void ParseArguments(int argc, char **argv)
{
  argc--; argv++;

  while (argc > 0) {
    if (!strcmp(*argv, "-filename")) {
       ...
    }
    else if (!strcmp(*argv, "-help")) {
      PrintUsage();
      exit(0);
    }
    else {
      fprintf(stderr, "Unrecognized argument: %s\n", *argv);
      PrintUsage();
      exit(1);
    }
    argv++; argc--;
  }
}
```

Robust Programming

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

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

void *Array_getKth(Array_T oArray, int k)
{
  if (!oArray) {
    fprintf(stderr, “oArray=NULL in Array_getKth\n”);
    abort();
  }

  if ((k < 0) || (k >= oArray->nelements)) {
    fprintf(stderr, “k=%d in Array_getKth\n”, k);
    abort();
  }

  return oArray->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!!!

`assert.h`

```
#ifdef NDEBUG
#define assert(_e) 0
#else
#define assert(_e) \
  if (_e) { \
    fprintf(stderr, “Assertion failed on line %d of file %s\n”, __LINE__, __FILE__); \
    abort(); \
  }
  0
#endif
```

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_getKth(Array_T oArray, int k)
{
  assert(oArray);
  assert((k >= 0) && (k < oArray->nelements));

  return oArray->elements[k];
}
```

Assert

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

```
#include <assert.h>

void Array_removeKth(Array_T oArray, int k)
{
  int i;

  assert(oArray);
  assert((k >= 0) && (k < oArray->nelements));

  for (i = k+1; i < oArray->nelements; i++)
    oArray->elements[i-1] = oArray->elements[i];

  oArray->nelements--;

  assert(oArray->nelements >= 0);
}
```

C Preprocessor

- Invoked automatically by the C compiler
  - try `gcc –E foo.c`
- C preprocessor manipulates text prior to C compiling
  - file inclusion
  - conditional compilation
  - macros

File Inclusion

- Header files contain declarations for modules
  - Names of header files should end in `.h`
- User-define header files `“ ... ”`

```
#include “mydefs.h”
```

- System header files: `< ... >`

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

Conditional Compilation

- Removing macro definitions

```
#undef plusone
```

- Conditional compilation

  #ifdef *name*
  #ifndef *name*
  #if *expr*
  #elif *expr*
  #else
  #endif

- Why use?

```
#ifndef FOO_H
#define FOO_H

#ifdef WINDOWS_OS
#include <windows.h>
#endif

.
.
.
#endif
```

```
gcc -DWINDOWS_OS foo.c
```

Macros

- Provide parameterized text substitution
- Macro definition

```
#define MAXLINE 120
#define lower(c) ((c)-`A'+'a')
```

- Macro replacement

```
char buf[MAXLINE+1];
```

becomes

```
char buf[120+1];
```

```
c = lower(buf[i]);
```

becomes

```
c = ((buf[i])-`A'+'a');
```

Macros (cont)

- Always parenthesize macro parameters in definition

```
#define plusone(x) x+1

i = 3*plusone(2);
```

becomes

```
i = 3*2+1
```

```
#define plusone(x) ((x)+1)

i = 3*plusone(2);
```

becomes

```
i = 3*((2)+1)
```

Macros (cont)

- Always avoid side-effects in parameters passed to macros

```
#define max(a, b) ((a)>(b)?(a):(b))

y = max(i++, j++)
```
becomes
```
y = ((i++)>(j++)?(i++):(j++));
```

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
  - Complete successfully, or
  - Output a meaningful error message

Different execution times

1. Preprocessing time
2. Compile time
3. Run time