Errors handling

The problem

```c
#include <stdio.h>
void sophisticatedAlgorithm (char* name)
{
    FILE * fd = fopen (name); // using the file
                              // for an algorithm
    // ...
}
int main() {
    char name[100];
    scanf ("%s", name);
    sophisticatedAlgorithm (name);
    // how do we know if there was a problem with
    // the file, or any other problem?
}
```

OOP (Java / C++) solution: exceptions

```
try
 {
    FileInputStream fstream = new
                   FileInputStream(name);
    DataInputStream in = new
                   DataInputStream(fstream);
    while (in.available() !=0)
    {
        System.out.println (in.readLine());
    }
    in.close();
 }
 catch (IOException e)
 {
   System.err.println("File input error");
 }
```

How can it be done in C?

Errors types

- **Bugs**
  - Deterministic errors
  - Not dependant on the program inputs
  - You **assert** they will never happen

- **Exceptions**
  - Originate from program input and environment
    - Input streams
    - Memory allocations
    - …
  - **May happened** from time to time

Catching bugs -- assert

```
#include <assert.h>
// Sqrt(x) - compute square root of x
// Assumption: x non-negative
double Sqrt(double x )
{
   assert( x >= 0 ); // aborts if x < 0
   //...
}
```

If the program violates the condition, then we'll get:

```
   assertion "x >= 0" failed: file "Sqrt.c",
   line 7 <exception>
```

This allows to catch the event during debugging

Using assert

- Terminates the program continuation
- Good for debugging and logic examination
- Discarded in NDEBUG mode

User of the library function can not decide what to do in case of an error

assert

Note:

- ```
  assert(importantCalculation() == 0);
  ```

  bad, since foo() will not be called if the compiler removes the assert() ⇔ if NDEBUG is defined

- ```
  int errCode = importantCalculation();
  assert(errCode == 0);
  ```

  good ☺

assert

Use for catching bugs

Don't use for checking malloc, user input,...

C exception handling strategies

Detecting the errors

1. Catch the exception before it occurs
2. Use function return value to indicate errors
3. Use global variables to indicate which errors occurred and their description
4. Develop an ‘exception-catching- like’ mechanism

(will not be discussed in this course)

Handling the errors

- May include printing error massages
- May include program termination

Handling the errors
- Printing error messages

- Use the *standard errors stream (stderr)*

- Relevant functions (examples in the following slides):
  - `fprintf(stderr, “format string”, …)`
  - `perror`
  - `strerror (with errno)`

- `stdout` and `stderr` can be redirected separately:
  - `~% (myProg > outputFile) >& errorFile`

Handling the errors
- Return status – from main()

Usually, the following **convention** is used:

- ‘0’ – success
- other values – failure (most common: ‘1’/ ’-1’)
- NULL means failure when function returns a pointer

For example, `stdlib.h` defines the macros:

- `#define EXIT_SUCCESS 0`
- `#define EXIT_FAILURE 1`

ALWAYS CHECK WHAT THE RETURN VALUE MEANS

Handling the errors
- exit() – (try to avoid it, panic..)

`exit(int status)`

terminates the program in case of an exception

```
#include <stdio.h>      /* fprintf, fopen */
#include <stdlib.h>     /* exit, EXIT_FAILURE */
int readFile () {
  FILE * pFile;
  pFile = fopen ( “myfile.txt”, “r” );
  if (pFile == NULL) {
    fprintf (stderr, “Error opening file” );
    exit (EXIT_FAILURE);
  } else {
    /* file operations here */
  }
  return EXIT_SUCCESS;
}
int main() {
  int status = readFile();
  return status;
}
```

Always remember to:

1. Close files
2. Free memory

(if needed)

Detecting - Find the error before it occurred

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int dividend = 20;
   int divisor = 0;
   int quotient;
   scanf(“%d”, &divisor);
   if ( divisor == 0) {
      fprintf(stderr, "Division by zero! Exiting...\n");
      return 1;
   }
   quotient = divide(dividend, divisor);
   fprintf(stderr, "Value of quotient : %d\n", quotient);
   return 0;
}
```

Detecting - Special return values to indicate error

```
#include <stdio.h>
int sophisticatedAlgorithm (char* name)
{
  FILE * fd = fopen (name);
  if ( fd == NULL )
  {
     return -1; // indicate an abnormal
                // termination of the function
  }
  // do your sophisticated stuff here

  return 0; // indicate a normal
            // termination of the function
}
```

Detecting - Special return values to indicate error

```
int main()
{  // ...
   if (sophisticatedAlgorithm(name) == -1)
   {
      // the exceptional case
   }
   else
   {
      // the normal case
   }
}
```

Detecting - Special return values to indicate error

User of a library function can decide what to do in case of an error

But!

- We may have no free value to indicate an error
- May need separate values for each error type
- Requires checking after each function call
  → no separation of regular code from the error checking

Can’t use return val for both result of the function and error type?

In case no return value is free, we can either

- ✓ indicate errors by setting a **global variable**
- ✓ use a **combination of return value** (usually for error indication), and an **address of a given argument** for return value

Modify a global variable

```
int g_divisionError;

int divide(int dividend, int divisor) {
    g_divisionError = 0;
    if (divisor == 0) {
        g_divisionError = 1;
        return 1;
    }
    return dividend / divisor;
}
```

Modify a global variable

```
int main() {
   int c = divide(20,0);
   if (g_divisonError == 1) {
      fprintf(stderr,
              "Division by zero! Exiting...\n");
      return EXIT_FAILURE;
   }
   // else do something with c…
}
```

Modify a local variable using a pointer

```
int divide(int dividend, int divisor,
           int *quotient) {
   if (divisor == 0) {
      return 1;
   }
   *quotient = dividend / divisor;
   return 0;
}
```

Modify a local variable using a pointer

```
int main() {
  int c;
  int div_error = divide(20, 0, &c);
  if (div_error == 1) {
      fprintf(stderr,
              "Division by zero! Exiting...\n");
      return EXIT_FAILURE;
  }
  // else do something with c…
}
```

Detecting - The standard library approach

Combination of return value and global variable to indicate errors

The idea: separate between function return code and error description

- Functions just return 0 in case of success or -1 in case of error
- A global variable holds the specific error code (and message) which describes the occurred error

Example:

```
#include <stdio.h> // for perror
#include <stdlib.h>
#include <errno.h> // for the global variable
                   // errno
#include <string.h> // for strerror
const char *FILE_NAME =
"/tmp/this_file_does_not_exist.haha";
```

Example:

```
int main( int argc, char **argv )
{
    int fd = 0;
    fd = open( FILE_NAME, O_RDONLY, 0644 );
    if( fd < 0 )
    {
        // Error, as expected.
        perror( "Error opening file" );
        printf( "Error opening file: %s\n",
                strerror( errno ) );
    }
    return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>     // for perror
#include <errno.h>     // for the global variable errno
#include <string.h>    // for strerror

extern int errno ;

int main ()
{
   FILE * pf;
   int errnum;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL)
   {
      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   }
   else
   {   // working with the file, and in the end:
      fclose (pf);
   }
   return 0;
}
```

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Detecting - C exceptions

Google “C exceptions” will lead to many useful C libraries that implement some kind of exceptions, very similar to Java/C++

Bug 1

```
(1) typedef struct _Student
(2) {
(2)         int id;
(3)         char * name;
(4) } Student;

(5) Student * stud = (Student *) malloc( sizeof(Student) );
(6) stud->id = 123456;
(7) stud->name = (char *) malloc(100*sizeof(char));
                        …
(8) free(stud);
```

Memory leak of ‘name’!

Bug 2

```
void myFunc()
{
        int * x = randomNumPtr();
        int result = *x;            // unexpected!
        *x = 17;                    // accessing unallocated space!
}

 int * randomNumPtr()
{
        int j= srand( time(0) );
        return &j;
 }
```

Never return an address of a stack-variable !

Bug 3

```
void myFunc(char * input)
{
        char *    name;
        if (input != NULL )
        {
                name = (char*)malloc(MAX_SIZE);
                strcpy(name,input);
        }
  …
        free(name);
}
```

if input is NULL =>
free on an address that was not allocated using malloc.

No bug 3

```
void myFunc(char * input)
{
        char *   name=NULL;
        if (input != NULL )
        {
                name = (char*)malloc(MAX_SIZE);
                strcpy(name,input);
        }
  …
        free(name);
}
```

always initialize pointers to NULL!

Bug 4 –

```
void init(int *numbers)
{
        numbers = (int *)malloc(sizeof(int) * 5);
        for (int i = 0; i < 5; i ++)
        {
                numbers[i] = i;
        }
}

 int main()
{
        int *arr=  NULL;
        init(arr);
        free(arr);
 }
```

For changing a value inside a function – send it's poiter!
init(&arr)