Error handling

What exceptional things might our programs run in to?

Exceptions do occur

Whenever we deal with programs, we deal with computers and users.

Whenever we deal with computers, we know things don’t always go as we’d like.

Whenever we deal with users, we know that users don’t always behave as we’d hoped for.

When we write code, we also deal with programmers. We know that programmers don’t always *do the right thing*™

How do we deal with these cases?

Different types of exceptions

Some things might happen which we couldn’t really predict. A network failure, a crashed hardware etc.

Some things are more of a contingency (eventualities) nature. You don’t have enough money to make an order, you are not allowed to do something right now.

From these two types of extreme, unwanted situations arises a need to programmatically deal with these situations.

Contingencies - things that might occur

If we know that on rare occasions, an action will not work for logical reasons which are part of the rules for our application, it would be good if we could deal with these cases in a structured way.

Let’s say that we have a rule for a system for a webshop which says that a customer cannot place two orders within five seconds (because this indicates fraud or unnatural behavior). Then the method for processing orders could revoke an order in an orderly fashion.

Code calling the process_order() method should of course be required to deal with this rare eventuality.

Error handling - the C way

Mainly

Do something

Check if it worked

Also

Do something

Get signal, act on it

Error handling - the C way

Mainly

```
File *fp = fopen (name, “r”);
If (fp==NULL)
  {
    /* handle failure */
  }
/* continue writing as if things Worked */
```

printf

Upon successful return, these functions return the number of characters printed (excluding the null byte used to end output to strings).

If an output error is encountered, a negative value is returned.

Printf (printf-example.c)

```
if ( ret < 0 )
  {
    fprintf(stderr, "Uh oh, output error is encountered\n");
  }
```

Printf (printf-example.c)

```
  else if ( (unsigned int)ret != strlen(msg) )

    {

      fprintf (stderr, "Message was only partly printed\n");

      fprintf (stderr, " * Characters printed:     %d\n", ret);

      fprintf (stderr, " * Message length:         %lu\n", strlen(msg));

      fprintf (stderr, " * Remaining characters:   %lu\n", strlen(msg)-(unsigned
int)ret);

    }
```

scanf (scanf-example.c)

```
int

read_from_user(char *prompt, char *buffer)

{

  int ret;

  printf("%s", prompt);

  ret = scanf("%255s",buffer);

  return ret;

}
```

scanf (scanf-example.c)

```
int ret = read_from_user("Enter your name: ", user_name);


if (ret > 0 )
  {
    printf("Your name: %s\n",user_name);
  }
```

scanf (scanf-example.c)

```
else if ( ret == 0 )
  {
    fprintf(stderr, "Failed reading string\n");
    return 2;
  }
else if ( ret == EOF )
  {
    fprintf(stderr, "Could not read string... EOF (%d) returned\n", EOF);
    return 1;
  }
```

Return values

Common return values indicating a failure:

NULL

-1

1, 2, 3

….. So waddya think 0 usually means?

gets - gets-example.c

NAME

gets - get a string from standard input (DEPRECATED)

DESCRIPTION

Never use this function.

gets() reads a line from stdin into the buffer pointed to by s until

either a terminating newline or EOF, which it replaces with a null byte

('\0'). No check for buffer overrun is performed (see BUGS below).

atoi

NAME

atoi, atol, atoll - convert a string to an integer

RETURN VALUE

The converted value.

atoi - atoi-example.c

```
void
print_atoi(char *s)
{
  int scanned_value;
  printf("%10s  %4d  [", s==NULL?"NULL":s, atoi(s));
  if (sscanf(s, "%d", &scanned_value)!=1)
    {  printf("fail"); }
  else
    { printf ("%d", scanned_value); }
  printf("]\n");
}
```

atoi - atoi-example.c

```
gcc atoi-example.c -o atoi-example && ./atoi-example

     string  atoi sscanf

----------------------------

      -123  -123  [-123]

         0     0  [0]

       123   123  [123]

   AS Roma     0  [fail]

Segmentation fault (core dumped)
```

A bit of warning

Don’t call exit in functions you’re writing as a “library”. Just don’t! Really, **don’t!**

pause

Signals - not examined

“Signals are a limited form of **inter-process communication** used in Unix, Unix-like, and other POSIX-compliant operating systems. A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred. Signals originated in 1970s Bell Labs Unix and have been more recently specified in the POSIX standard.

When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction. **If the process has previously registered a signal handler, that routine is executed**. Otherwise, the default signal handler is executed.”

https://en.wikipedia.org/wiki/Unix_signal

Signals - ctrl-c anyone?

Compile and execute **signal-example.c**

```
while (1)
  {
    printf ("tick nr: %d  ", global_ctr);
    print_current_date();
    usleep(SEC_TO_USEC(1));
    global_ctr++;
  }
```

sigint

Can we send that signal by some other means?

`kill` command, e g

```
kill -SIGINT 4075
```

sigint

Can we send other signals?

`kill` command, e g

```
kill -SIGUSR1 4075
```

Ctrl -c - interrupts the program

Actually by sending a signal called SIGINT

Can we catch it?

Catching / handling signals

```
void handle_usr(int sigid)

{

  if (sigid==SIGINT)

    {

      printf ("interrupt signal received... %d\n", sigid);

    }

}
```

In main:

```
        signal(SIGINT, handle_usr);
```