Mathematical Software Programming (02635)

Module 10 — Fall 2016

Instructor: Martin S. Andersen

Checklist — what you should know by now

- How to write a simple program in C (`int main(int argc, char *argv[]) {}`)
- Basic data types (`int`, `long`, `float`, `double`, ...)
- Basic input/output (`printf`, `scanf`)
- Implicit/explicit typecasting
- How to compile and run a program from terminal / command prompt
- Control structures and loops (`if`, `else if`, `switch`, `for`, `do`, `while`)
- Pitfalls with integer and floating point arithmetic
- Arrays and multidimensional arrays
- Pointers: *"dereferencing"* and *"address of"* operators
- Use of functions to structure programs
- Dynamic memory allocation (`malloc`, `calloc`, `realloc`, `free`)
- Basic error checking (check return values, etc.)
- Data structures and types (`struct`, `typedef`, lists, stacks, queues)
- Strings and file input/output
- Parallelization with OpenMP

Assignment 1: user input with infinite loop

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

int main(int argc, char *argv[]) {
  int m, n;
  while(1) {                  /* not a good idea! */
    printf("Input m: ");
    scanf("%d", &m);
    printf("Input n: ");
    scanf("%d", &n);

    if ( m > 0 && n > 0) { break; }
    else {
      fprintf(stderr, "m and n must be larger than zero!\n");
    }
  }
  printf("m = %d\n", m);
  printf("n = %d\n", n);
  return(EXIT_SUCCESS);
}
```

Assignment 1: user input with finite loop

```c
#include <stdio.h>
#include <stdlib.h>
#define ERR_MAX 10  // at most 10 user errors

int main(int argc, char *argv[]) {
  int m, n, errcnt = ERR_MAX;

  while( errcnt ) {           /* a much better idea! */
    printf("Input m: ");
    scanf("%d", &m);
    printf("Input n: ");
    scanf("%d", &n);

    if ( m > 0 && n > 0) { break; }
    else {
       fprintf(stderr, "m and n must be larger than zero!\n");
       errcnt--;
    }
  }
```

Assignment 1: user input with finite loop (cont.)

```
  /* ... continuation of code on previous slide ... */

  if (errcnt == 0) {
    fprintf(stderr, "Too many errors ... aborting!\n");
    exit(EXIT_FAILURE);
  }

  printf("m = %d\n", m);
  printf("n = %d\n", n);

  return(EXIT_SUCCESS);
}
```

This week

Topics

- External libraries and testing

Learning objectives

- **Debug** and **test** mathematical software.
- Call external (third party) programs and libraries.
- Analyze the runtime behaviour and the time and space complexity of simple programs.

Guidelines

- Design your program with testing in mind
- Do not try to construct a *full-featured* program from the beginning
- Start with specifications, data structures, and tests
- Implement and test one module/function at the time
- Use conditional compilation to include/exclude debugging code
- Use error checking and assertions
- Avoid (excessive) use of global variables
- Enable compiler **warnings** (`-Wall` and `-Wextra`)
- Aim for readability (as a rule of thumb, avoid `goto` statements)
- Use proper code indentation

Indentation

Proper indentation makes it easier to read and understand a program

Example 1

```
int x = 5;
while( x > 0 );
   x--;
```

How many times does the loop run? (Why?)

Indentation

Proper indentation makes it easier to read and understand a program

Example 1

```
int x = 5;
while( x > 0 );
   x--;
```

How many times does the loop run? (Why?)

Example 2

```
int x = 5;
while( x > 0 )
    ;
x--;
```

How many times does the loop run? (Why?)

Compiler toolchain

Preprocessing (`cpp`)

Processes *preprocessor directives* (`#include`, `#define`, `#ifdef`, ...)

`hello.c` → `hello.i` (*modified* source)

Compilation (`cc -S`)

`hello.i` → `hello.s` (*assembly language* program)

Assembly (`as`)

`hello.s` → `hello.o` (*machine code*)

Linking (`ld`)

`hello.o`, *libraries*, ... → *executable*

The C preprocessor

Macros

```
#define BUFFER_SIZE 1024
#define PI 3.141592653589793
#define dmalloc(x) malloc(x*sizeof(double))
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

Beware of macro pitfalls!

Pre-defined macros

`__FILE__`, `__LINE__`, C99: `__DATE__`, `__TIME__`, `__func__`

System-specific macros

`_WIN32`, `_WIN64`, `__linux__`, `__APPLE__`, `__MACH__`, `__unix__`

Example: error handling

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    double *data = malloc(100*sizeof(*data));
    if ( data == NULL ) {
        fprintf(stderr, "Malloc failed in %s function,"
            " line %d\n",__func__,__LINE__);
        return EXIT_FAILURE;
    }

    /* .. some code that accesses the array .. */

    free(data);
    data = NULL;
    return EXIT_SUCCESS;
}
```

Assertions

Run-time assertions

- Boolean expressions that should be true *unless* there is a bug
- Useful for debugging, but should not replace error checking

```
#include <assert.h>
...

assert( expression );
```

Switching off assertions

- Define `NDEBUG` macro before including `assert.h`
- Define `NDEBUG` macro at compile time (`-Dname=value`)

```
$ cc -Wall -DNDEBUG source.c -o my_program
```

Example: assertions

```c
#include <assert.h>
#include <stdlib.h>
void my_function(double *data, int size) {
    assert(data != NULL);
    assert(size > 0);
    /* Insert function body here */
    return;
}
int main(void) {
    my_function(NULL, 5);
    return 0;
}
```

```
$ ./example
Assertion failed: (data != NULL), function my_function,
   file example.c, line 4.
Abort trap: 6
```

Debugging

Compile program with -g flag to create "debug version" of executable

Terminal debuggers

- Set breakpoints and step through program
- Trace program and inspect variables
- GNU db (`gdb`), Intel db (`idb`), Sun db (`dbx`), LLVM db (`lldb`)

Integrated debuggers

- Set breakpoints and inspect variables directly in IDE

Conditionally compiled debugging code

- Augment program with debugging code (assertions, etc.)
- Print (selected) variables for debugging purposes

Common errors

Uninitialized pointer

```
double *pd;
...

*pd = 5.0;
```

Debugging: enable compiler warnings (`-Wall`)

Dereferencing `NULL`

```
int *pi = NULL;
...

*pi = 2;
```

Debugging: include assertion before dereferencing pointer

Common errors

Missing allocation

```
int n = 10;
double *A;
for (int i=0;i<n;i++) {
    A[i] = 1.0;
}
```

Debugging: initialize pointers (`double *A = NULL`) and use assertions

Memory leak (missing deallocation)

```
void my_function(size_t n) {
    int *p = malloc(n*sizeof(*p));
    /* Some code but no call to free() before end of scope */
    return;
}
```

Debugging: check calls to `malloc` and `free`, or use memory profiler

Common errors

Buffer overflow / index out of bounds

```
/* Example 1 */
double data[10];
for (int i=0;i<=10;i++) {
    printf("data[%d] = %g\n",i,data[i]);
}

/* Example 2 */
char s[5];
s[5] = '\0';
```

Debugging: add assertions or use debugger

Common errors

Missing null-termination

```
char s[5];
s[0] = 'H'; s[1] = 'e'; s[2] = 'l'; s[3] = 'l'; s[4] = 'o';
puts(s);
```

Debugging: check `char` operations that operate on strings or use debugger

Unindended usage of preprocessor macro

```
#define cube(x) x*x*x
double d = cube(2+3);   // expands to 2+3*2+3*2+3, not 5*5*5
```

Debugging: check macros / preprocessor output

Common errors

Stack overflow: automatic allocation of large arrays

```
double data[2097152]; // requires 16 MB of storage
```

Debugging: check size of automatically allocated data structures

Stack overflow: recursive function calls

```
long fibonacci(long n) {
   if ( n == 0 ) return 0;
   else if ( n == 1 ) return 1;
   else return fibonacci(n-1) + fibonacci(n-2);
}
```

What happens if `fibonacci()` is called with negative *n*?

Debugging: use a debugger to trace function calls