Mathematical Software Programming (02635)

Module 4 — Fall 2016

Instructor: Martin S. Andersen

Checklist — what you should know by now

- How to write a simple program in C (`int main(void) {}`)
- 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

This week

Topics

- Program structure
- Memory allocation

Learning objectives

- Describe and use data structures such as **arrays**, linked lists, stacks, and queues.
- Choose appropriate data types and data structures for a given problem.
- Design, implement, and document a program that solves a mathematical problem.

Program structure

Functions

```
<type> function_name(<type> <arg1>, <type> <arg2>, ...) {
    // body
}
```

- function prototype, header, and body
- single return value, multiple inputs
- variables are *automatic* — scope is code block enclosed between { }
- never return a pointer to a local variable!

Examples

```
int main(void);
int printf(const char* format, ...);
void my_func(double* arr, const unsigned int length);
double * new_vector(size_t length);
```

C uses *call-by-value* method to pass arguments

```c
#include <stdio.h>
void swap(int a, int b);   // Function prototype

int main(void) {
    int a = 1, b = 3;
    swap(a,b);
    printf("a = %d and b = %d\n",a,b);
    return 0;
}

void swap(int a, int b) {
    int c = a;  // Store value of a in c
    a = b;      // Overwrite a with b
    b = c;      // Overwrite b with c
    return;
}
```

What is the value of `a` and `b` after calling `swap(a,b)`?

Pointers as arguments

```c
#include <stdio.h>
void swap2(int* a, int* b);   // Function prototype

int main(void) {
    int a = 1, b = 3;
    swap2(&a,&b);
    printf("a = %d and b = %d\n",a,b);
    return 0;
}

void swap2(int* a, int* b) {
    int c = *a;  // Store value of *a in c
    *a = *b;     // Overwrite *a with *b
    *b = c;      // Overwrite *b with c
    return;
}
```

What is the value of `a` and `b` after calling `swap2(&a,&b)`?

Dynamic memory allocation

Prototypes (`stdlib.h`)

```
void *malloc(size_t size);
void *calloc(size_t nelements, size_t elementSize);
void *realloc(void *pointer, size_t size);
void free(void *pointer);
```

Allocating an array of length N

```
double *pdata = malloc(N*sizeof(*pdata));

// Check if memory allocation failed
if ( pdata == NULL ) {
    // Code to deal with memory allocation failure ...
}
```

Extending dynamically allocated memory

```
double *pdata = malloc(N*sizeof(*pdata));
if ( pdata == NULL ) {
    // Code to handle memory allocation failure ...
}

...

// Request more memory (N + 100)
N += 100;
double *ptmp = realloc(pdata, N*sizeof(*pdata));
if ( ptmp == NULL ) {
    // Code to handle reallocation failure ...
    //   pdata is still a valid pointer
}
else
    pdata = ptmp;
```

Releasing memory

```
free(pdata);     // Free memory pointed to by pdata.
pdata = NULL;    // <--- Not necessary, but good practice!
```

Common errors

- freeing memory twice
- freeing unallocated memory
- using pointer after freeing memory
- forgetting to free memory (memory leak)

Memory: stack vs heap

Stack

- layout decided at compile-time (automatically allocated at run-time)
- no allocation/deallocation overhead
- variables cannot be resized
- local variables only
- fast access but limit on stack size

Heap

- dynamic memory allocation is controlled by operating system
- you must manage memory
- memory may become fragmented over time
- variables can be accessed globally
- variables can be resized
- no limit on memory size (other than hardware limitations)
- slower access than stack

Allocating a two-dimensional array (WSS, p. 94)

Algorithm 1: Naive $m \times n$ matrix allocation method

```
B = (double **)malloc(m*sizeof(double *));
if ( B == NULL ) return NULL;

for ( i = 0; i < m; i++ )
{
    B[i] = (double *)malloc(n*sizeof(double));
    if ( B[i] == NULL ) { free(B); return NULL; }
}
```

How should you free the memory allocated by Algorithm 1?

Is it possible for Algorithm 1 to leak memory?

Allocating a two-dimensional array (WSS, p. 94)

Algorithm 2: Fast $m \times n$ matrix allocation method

```
B = (double **)malloc(m*sizeof(double *));
if ( B == NULL ) return NULL;

B[0] = (double *)malloc(m*n*sizeof(double));
if ( B[0] == NULL ) { free(B); return NULL; }

/* now set the other pointers */
for ( i = 1; i < m; i++ )
    B[i] = B[0] + i*n;
```

How should you free the memory allocated by Algorithm 2?

Is it possible for Algorithm 2 to leak memory?

Quiz time!

1. Go to socrative.com on your laptop or mobile device
2. Enter "room number" **02635**
3. Answer ten quick question (the quiz is anonymous)