# Mathematical Software Programming (02635)

## Lecture 4 — September 27, 2018

Instructor: Martin S. Andersen

Fall 2018

# 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
- Limitations of integer and floating-point arithmetic
- Automatic 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.

# 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_func1(double* param, const size_t length);
void my_func2(double param[], const size_t length);
double * new_vector(const 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)

```c
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 $N$

```c
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

```c
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

```c
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 (automatic allocation)

- Layout decided at compile-time (variables cannot be resized)
- Local variables
- No allocation/deallocation overhead
- Fast access but limited by stack size

## Heap (dynamic allocation)

- Programmer must explicitly allocate/deallocate memory
- Dynamic memory allocation/deallocation is controlled by operating system
- Variables can be resized and accessed globally
- Memory may become fragmented over time
- No limit on memory size (other than hardware limitations)
- Slower access than stack

Remark: static and global variables stored in “data segment”

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

## Algorithm 1: naive $m \times n$ matrix allocation method

```c
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

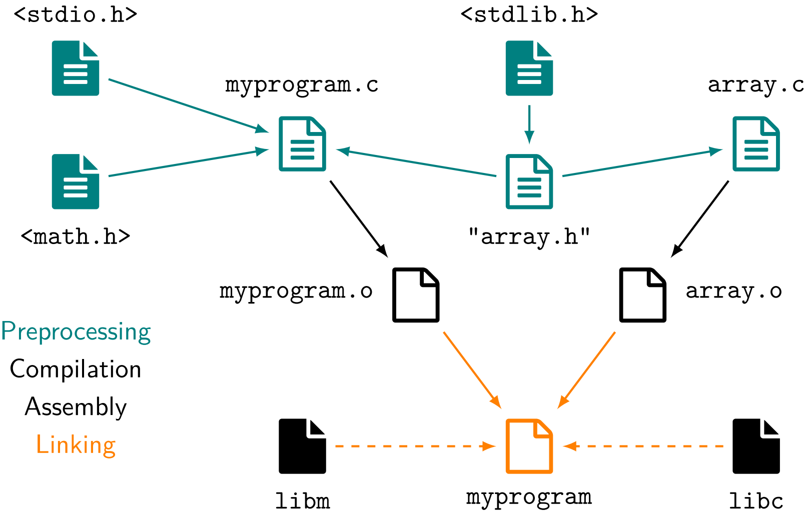
```
B = (double **)malloc(m*sizeof(double *));
if ( B == NULL ) return NULL;

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

/* Set the remaining 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?

# Multiple-file projects




Preprocessing

Compilation

Assembly

Linking

# Building multiple-file projects

## Manual compilation/assembly and linking

```
$ gcc -c -Wall -std=c99 array.c
$ gcc -c -Wall -std=c99 myprogram.c
$ gcc myprogram.o array.o -lm -o myprogram
```

## Building with make

- Create a makefile with source and library dependencies

```
$ make myprogram
gcc -Wall -std=c99   -c -o myprogram.o myprogram.c
gcc -Wall -std=c99   -c -o array.o array.c
gcc   myprogram.o array.o  -lm -o myprogram
```

# Makefiles revisited

```
variable = value
target : dependencies
    command
```

- Make has many implicit rules (`make -p -f/dev/null`), e.g.,

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
$(COMPILE.c) -o $@ $<
LINK.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- implicit dependencies: `target` depends on `target.o`, `target.o` depends on `target.c`
- explicit dependencies
  - object files may depend on header file(s)
  - executable `target` may depend on multiple object files

# Makefile for project with two source files

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=-lm

myprogram: myprogram.o array.o
myprogram.o: array.h
array.o: array.h

.PHONY: clean            # "clean" does not create file with target name
clean:                   # Removes myprogram and all object files
    -$(RM) myprogram *.o
```

Automatically generate dependencies with GCC/Clang

```
$ gcc -MM *.c
```

# Generic makefile for multiple-file projects

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=-lm
objects=$(patsubst %.c,%.o,$(wildcard *.c))

myprogram: $(objects)

.PHONY: clean run
clean:
	-$(RM) myprogram $(objects)

# target for Atom extension "gcc-make-run"
run: myprogram
	./myprogram $(ARGS)
```

# 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)