NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- eBook in the library.

  S. Malik C++ Programming Program Design including data structures 8th Edition 2018

- Conventions to label slides C and C++ compatible

  C and C++ | C (and C++) | C++

- Review C Program structure
- Review Storage classes
- Dynamic memory

C (and C++)

file3.h

```
extern int global_variable;  /* Declaration of the variable */
```

file1.c

```
#include "file3.h"  /* Declaration made available here */
#include "prog1.h"  /* Function declarations */

/* Variable defined here */
int global_variable = 37;    /* Definition checked against
declaration */

int increment(void) { return global_variable++; }
```

File2.c

```
#include "file3.h"
#include "prog1.h"
#include <stdio.h> // #include <cstdio> C++

void use_it(void)
{
   printf("Global variable: %d\n", global_variable++);
}
```

prog1.h

```
extern void use_it(void);
extern int increment(void);
```

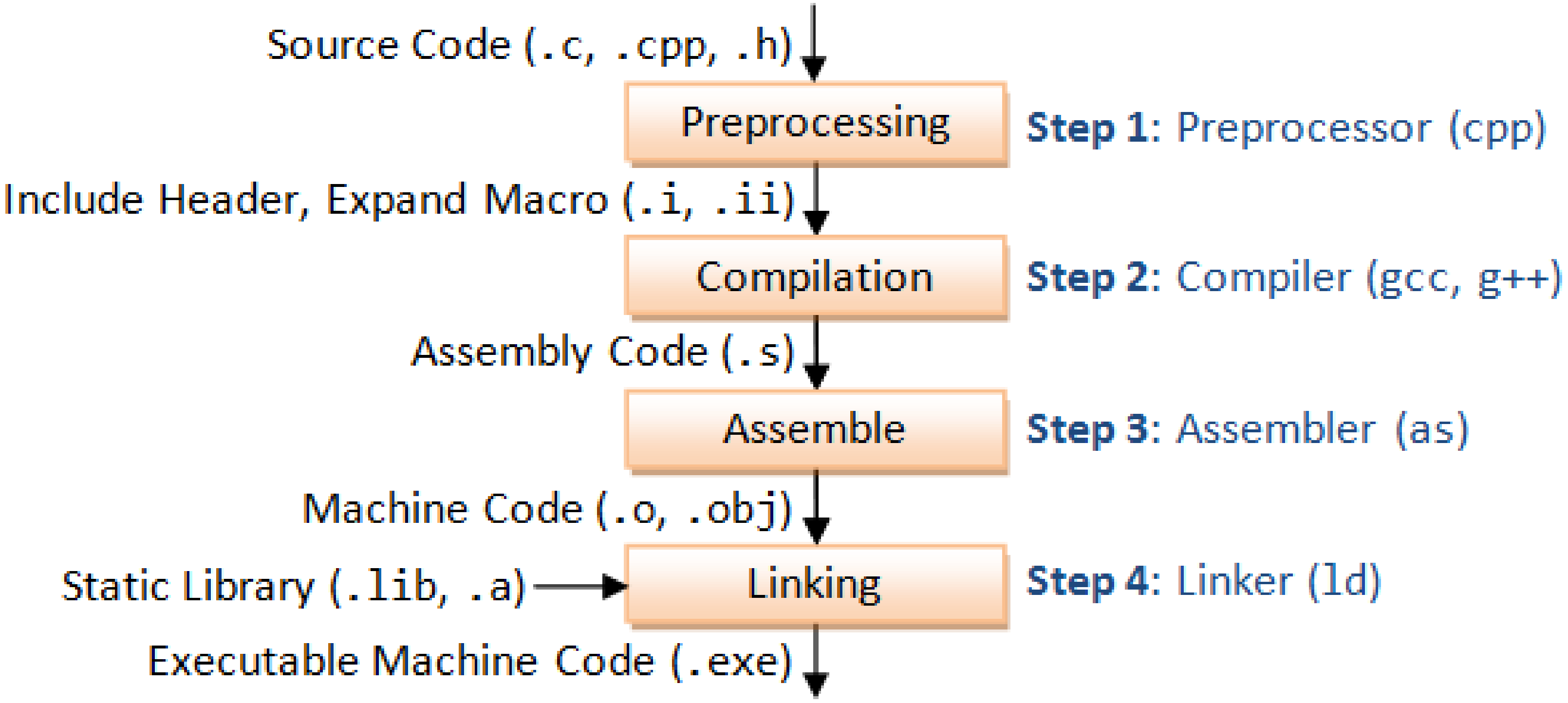
prog1.c

```
#include "file3.h"
#include "prog1.h"
#include <stdio.h> // #include <cstdio> C++

int main(void)
{
   use_it();
   global_variable += 19;
   use_it();
   printf("Increment: %d\n", increment());
   return 0;
}
```

prog1 uses prog1.c, file1.c, file2.c, file3.h and prog1.h

Compilation process

C and C++



GNU C/C++ Compilers (gcc g++)

- gcc and g++ do
  - pre-processing
  - compilation
  - assembly
  - linking
- Normally all done together, but you can get gcc and g++ to stop after each stage

C (and C++)

Using gcc (and g++)

```c
// file named hello.c
#include <stdio.h>
int main() {
        printf("Hello, world!\n");
        return 0;
}
```

- Help use man pages man gcc
- Compile and link a c program
  gcc hello.c // Compile and link source //file hello.c into a
- assign executable file mode via command “chmod a+x hello ”
- **gcc -o hello hello.c** //specify the output //file name use –o option
- Compile and link separately –c option
  gcc -c file1.c
  gcc -c file2.c
  gcc -o myprog.exe file1.o file2.o

```cpp
// file named hello.cpp
#include <iostream>
using namespace std;
 int main() {
   cout << "Hello, world!" << endl;
   return 0;
```

Compiler commands for C++ source files

man g++

g++ hello.cpp

g++ -o hello hello.cpp

g++ -c file1.cpp

g++ -c file2.cpp

g++ -o myprog.exe file1.o file2.o

Previously…

C and C++

Summary of storage classes:

Storage class	Declaration	Default init value	Stored in	Scope	Lifetime
`auto`	Inside block	Garbage	Memory	Local	Automatic
`static`	Inside block	0	Memory	Local	Static
	Outside any block	0	Memory	Global	Static
`extern`	Outside any block	0	Memory	External	Static
`register`	Inside block	Garbage	Maybe in register	Local	Automatic

Runtime Memory Storage Layout

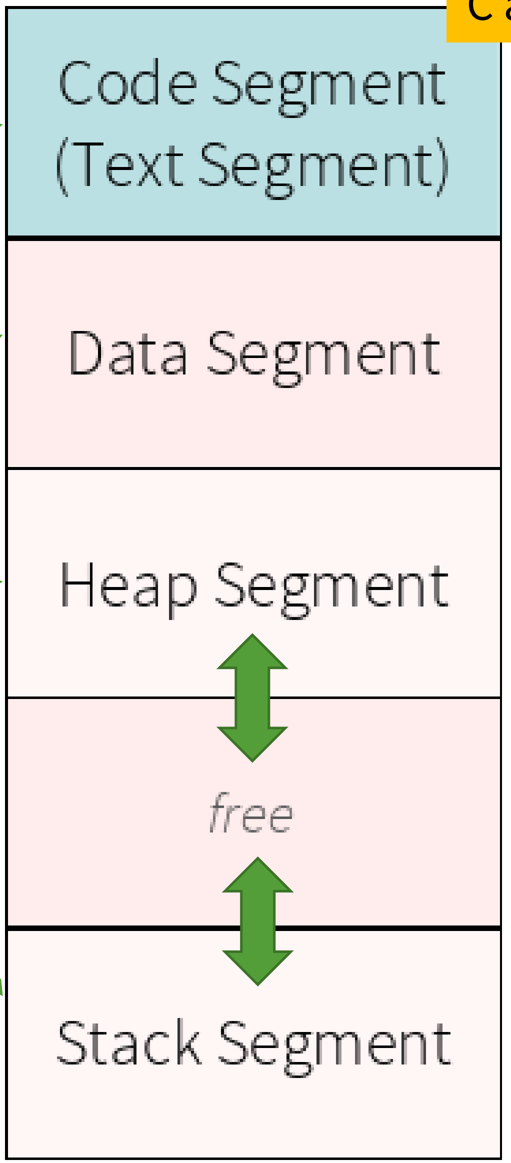
Contains the program's machine code	Code Segment (Text Segment)
Contains static data (e.g., `static` class, `extern` globals)	Data Segment
Contains dynamically allocated data – later…	Heap Segment
Unallocated memory that the stack and heap can use	*free*
Contains temporary data (e.g., `auto` class)	Stack Segment

Memory space for program code includes space for machine language code and data

- Text / Code Segment
  - Contains program's machine code
- Data spread over:
  - Data Segment – Fixed space for global variables and constants
  - Heap Segment – For dynamically allocated memory; expands / shrinks as program runs
  - Stack Segment – For temporary data, e.g., local variables in a function; expands and shrinks as program runs

- Local variables in functions allocated when function starts:
  - Memory allocated on the Stack Segment
  - When function ends, memory space is freed up
  - When size of the data item (int, array, etc.) is known at compile time it is referred to as static memory allocation

C and C++



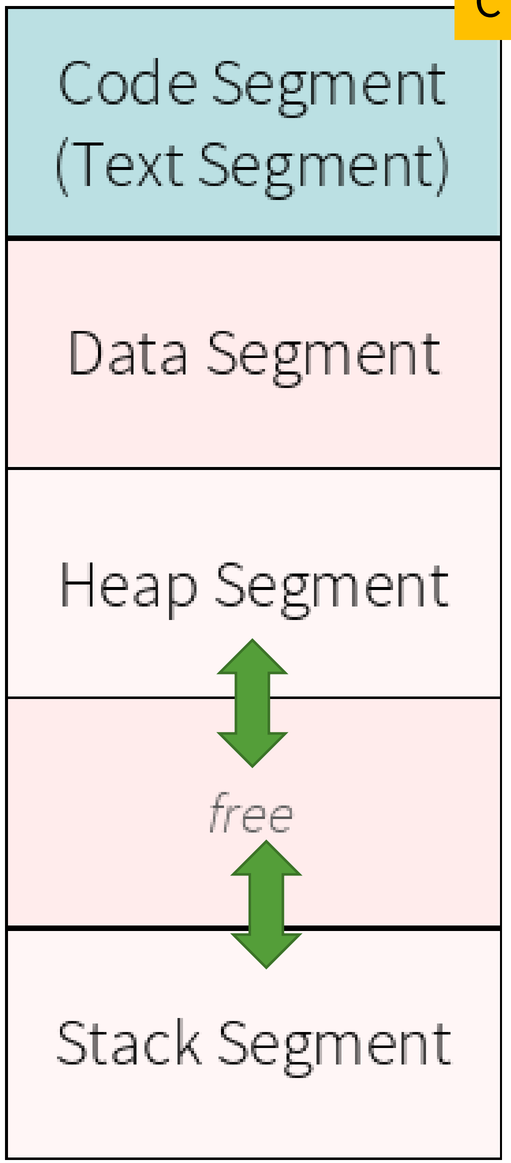
Static memory allocation

- Pre-define the sizes of arrays, and other variables.
- What if we don't know how much space will needed ahead of time? Programmer is writing a program to do the following:
  - ask user how many numbers to read in
  - read set of numbers into an array (of appropriate size)
  - calculate the average (look at all numbers)
  - calculate the variance (based on the average)
- Problem: how big do we make the array??
- Using static allocation, we have to make the array as big as the user might specify, and still might not be big enough … re-compile code with large memory allocation.

C and C++

Dynamic memory allocation

- Allow the program to allocate some variables (notably arrays), during the program execution, based on variables inthe program (dynamically)
- Previous example:
  - ask the user how many numbers to read,
  - then allocate array of appropriate size
- Approach:
  - Program has routines allowing user to
  - Specify some amount of memory – (eg.array size),
  - The program then requests and uses this memory
  - Then returns it when it is finished using the memory
  - Memory is allocated in the Heap Segment or Data Heap



Dynamic memory management functions in C and C++

- calloc - allocate arrays of memory (n elements of “size” bytes)
- malloc - allocate a single block of memory of size bytes
- realloc - extend the amount of space allocated previously
- free - free up a piece of memory that is no longer needed by the program

Memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

C and C++

calloc – allocate memory for array

- Function prototype:

```
void *calloc(size_t num, size_t  esize)
```

  - **size_t –** special type used to indicate sizes, unsigned int
  - **num** – number of elements to be allocated in the array
  - **esize** – size of the elements to be allocated
    - to get the correct value, use sizeof(<type>)
    - memory of size num*esize is allocated on the Data Heap
- Allocated memory is initialized to 0
- calloc returns the address of the 1st byte of this memory
- Cast the returned address to the appropriate type
- If not enough memory is available, calloc returns **NULL**

Calloc example

C (and C++)

```
float *nums;
int a_size;
int idx;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) calloc(a_size, sizeof(float));

/* nums is now an array of floats of size a_size */
for (idx = 0; idx < a_size; idx++) {
  printf("Please enter number %d: ",idx+1);
  scanf("%f",&(nums[idx])); /* read in the floats */
}
/* Calculate average, etc. */
```

What is a potential problem of this code?

Always check the return value of calloc, malloc or realloc!

```
if(nums == NULL) {
/* exit or do some other
stuff */
}
```

free – return memory to heap

Function prototype:

**void free(void ***ptr)

- Memory at location pointed to by ptr is released (so that it could be used again)
- Program keeps track of each piece of memory allocated by where that memory starts;
- If we free a piece of memory allocated with calloc, the entire array is freed (released)
- Results are problematic if we pass as address to free an address of something that was not allocated dynamically (or has already been freed)

Free Example

C (and C++)

```
float *nums;
int a_size;

printf("Read how many numbers:");
scanf("%d",&a_size);
nums = (float *) calloc(a_size, sizeof(float));

/* Use array nums */

/* When done with nums: */
free(nums);

/* Would be an error to do it again - free(nums) */
```

Memory leaks

C and C++

```
void myfunc()
{
  float *nums;
  int a_size = 5;

  nums = (float *) calloc(a_size, sizeof(float));
  /* But no call to free(nums) */
} /* myfunc() ends */
```

- When function myfunc() is called,
  - space for array of size a_size allocated;
  - when function ends, pointer variable nums goes away,
  - but the space nums pointed at (the array of size a_size) which remains allocated on the Data Heap
- Worse, we have lost the address of that memory space!!!
- Problem called memory leakage

malloc – allocate memory

C and C++

Function prototype:

void * malloc(size_t esize)

- Similar to calloc, except we use it to allocate a single block of the given size esize
- Like calloc, memory is allocated from Data Heap
- NULL returned if not enough memory available
- Memory allocated is not initialised
- Memory must be released using free when no longer needed
- Can perform the same function as calloc if we simply multiply the two arguments of calloc together
- Following are equivalent:

```
malloc(a_size * sizeof(float))
calloc(a_size, sizeof(float))
```

realloc – increase memory allocation

Function prototype:

void * realloc(void * ptr, size_t esize)

- ptr is a pointer to a piece of memory previously dynamically allocated
- esize is new size to allocate (no effect if esize is smaller than the size of the memory block ptr points to already)
- Function performs following action:
  - allocates memory of size esize,
  - copies the contents of the memory at ptr to the first part of the new piece of memory, and lastly,
  - old block of memory is freed up.

realloc Example

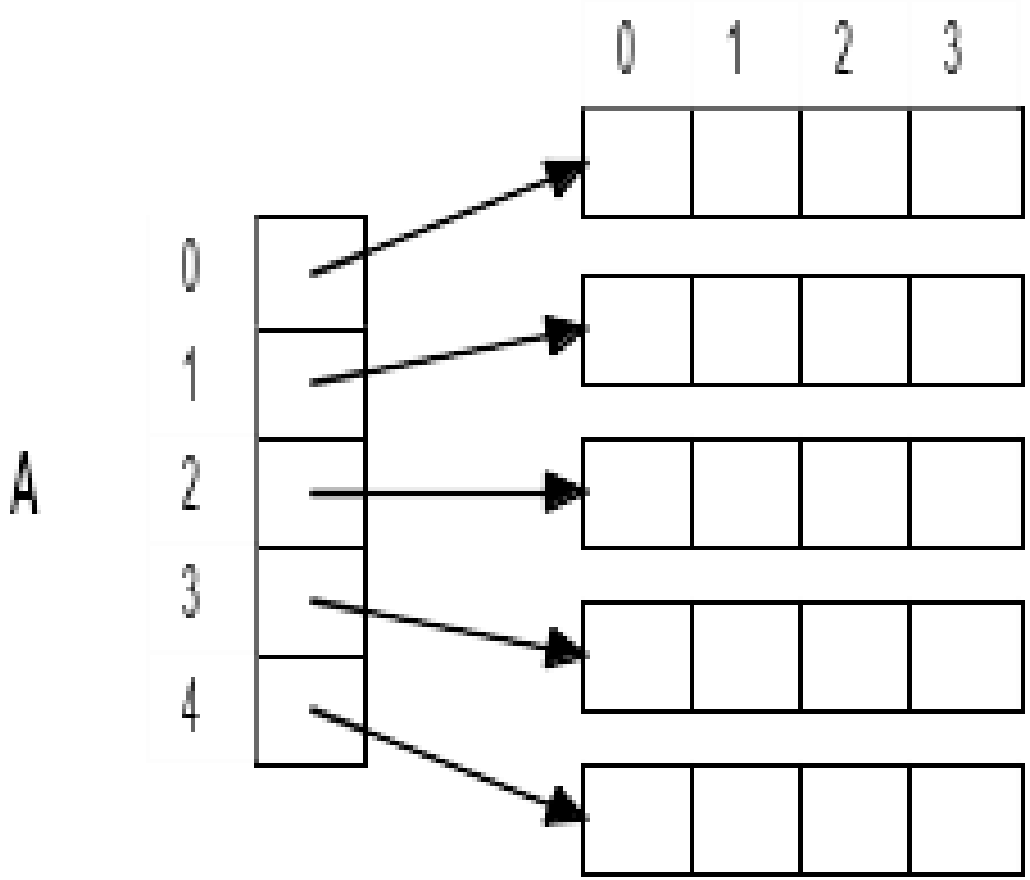
C and C++

```
float *nums;
int a_size;

nums = (float *) calloc(5, sizeof(float));
/* nums is an array of 5 floating point values */

for (a_size = 0; a_size < 5; I++)
  nums[a_size] = 2.0 * a_size;
/* nums[0]=0.0, nums[1]=2.0, nums[2]=4.0, etc. */

nums = (float *) realloc(nums,10 * sizeof(float));
/* An array of 10 floating point values is allocated, the first 5 floats from the old
   nums are copied as the first 5 floats of the new nums, then the old nums is
   released */
```

Allocating memory for 2D array

C (and C++)

- Can not simply allocate 2D (or higher) array dynamically
- Solution:
  - Allocate an array of pointers (1st dimension),
  - Make each pointer point to a 1D array of the appropriate size



Allocating memory for 2D array

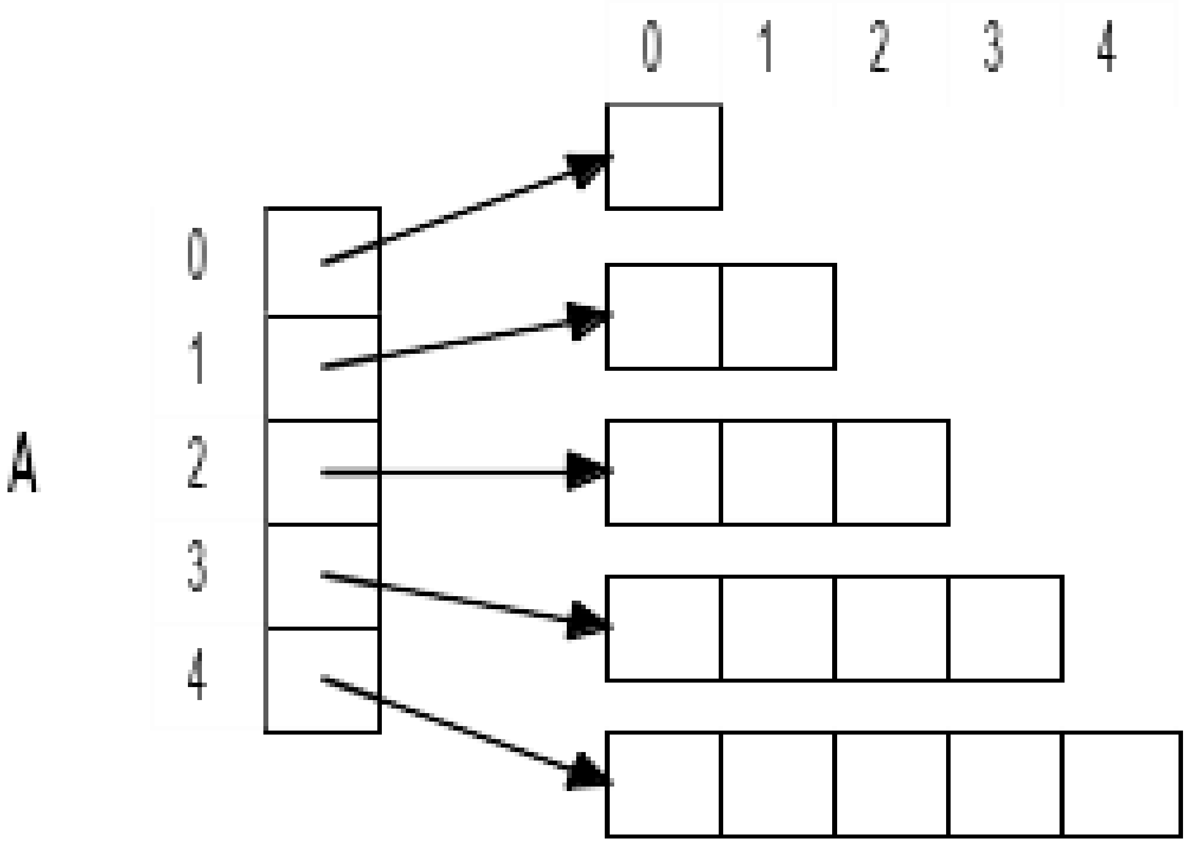
C (and C++)

```
float **A;  /* A is an array (pointer) of float pointers */
int X;

A = (float **) calloc(5,sizeof(float *));
/* A is a 1D array (size 5) of float pointers */

for (X = 0; X < 5; X++)
  A[X] = (float *) calloc(4,sizeof(float));
/* Each element of array points to an array of 4 float variables */

/* A[X][Y] is the Yth entry in the array that the Xth member of A points to */
```

Irregular-sized 2D array

C and C++

```
float **A;
int X;

A = (float **)calloc(5,
   sizeof(float *));

for (X = 0; X < 5; X++)
  A[X] = (float *)
        calloc(X+1,
        sizeof(float));
```



Common issues with dynamic memory allocation

C and C++

```
int *foo(void)  {
int x;               /* x does not exist outside the function */
                    /* Returning its address will result in  unknown program
 ...                 behaviour */
return &x;
}
```

- Returning a pointer to an automatic variable

- Heap block overrun -Similar to array going out of bounds

```
void foo(void) {
int *x = (int *) malloc(10 * sizeof(int));
x[10] = 10;   /* Allocated memory is only up to x[9] */
...
ffree(x);
}
```

```
int *pi;
void foo(void) {
pi = (int*) malloc(8*sizeof(int)); /* Leaked the old memory pointed to by pi */
…
free(pi); /* foo() is done with pi, so free it */
}
int main(void) {
pi = (int*) malloc(4*sizeof(int));
foo();
}
```

- Memory leak – loss of pointer to allocated memory

Common issues with dynamic memory

C and C++

- Potential memory leak
  - Loss of pointer to memory block
  - May still recover through pointer arithmetic

```
int *ip = NULL;

void foo(void) {
   ip = (int *) malloc(2 * sizeof(int));
   …
   ip++;    /* ip is not pointing to the start of  of the block anymore */
}
```

- Freeing non-heap or unallocated memory

```
void foo(void) {
  int fnh = 0;
  free(&fnh); /* Freeing stack memory */
}
void bar(void) {
  int *fum = (int *) malloc(4 * sizeof(int));
  free(fum+1); /* fum+1 points to middle of block */
  free(fum);
  free(fum); /* Freeing already freed memory */
}
```

Detecting memory leaks

- Valgrind is an open-source tool for detecting memory management and threading bugs
- It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour
- For more information: http://valgrind.org/