Brief Review of Programming with the C Language

Writing C code

- .c extension for source file
- .h extension for header file (contains exported interface such as public user-defined data-structures, functions, etc)

Generating programs

- Under unix, linux, windows (cygwin) with the gcc compiler (GNU compiler toolkit): gcc
- For single file:

```
$ gcc -o program_name source.c
```

- For multiple files:

```
$ gcc -o program_name source1.c source2.c source3.c
```

- Omitting "-o": the program name will be "a.out"

Compiling

- Generating a program = 2 steps: compiling and linking
- Compiling

```
$ gcc -c source.c
```

- Linking (this invokes the linker, ld on unix/linux):

```
$ gcc -o program_name source.o
```

- Similarly for a program that involves multiple files

Useful options

- -O: no optimization; -O2, -O3: different level of optimizations
- -Wall: prints all type of warnings
- -s: generate assembly code
- -std=c99, -std=c11, ...: specify the language standard
- -I/path/to/headers/ specifies additional paths to look for headers
- -L/path/to/libs/ specifies additional paths to libraries; used in conjunction with -l<libname> to specify the names of the libraries to link with

Debugging

- The GNU debugger: gdb
- Compile the source code with the -g flag (to keep debugging information):

```
$ gcc -o program_name source.c -Wall -g
```

- gdb is a command line program; GUI interface exists (ddd)
- start gdb by typing gdb followed by the program name:

```
$ gcc -o program_name source.c -Wall -g
$ gdb ./program_name
```

Useful GDB commands

This is a short list of useful gdb commands:

- run: start the execution of the program to be debugged
- kill: stop the execution of the program
- break filename.c:line: set a breakpoint in the source file filename.c at the given line
- c: continue execution until the next breakpoint
- next: execute the next instruction
- step: execute the next instruction; if it is a function call, enter the function body
- print expression: print the value of the specified expression
- quit: quit the gdb program

Datatypes

- Datatype: the datatype of an object determines the set of values that the object can take and the type of operations that can be applied on it
- Example of types:
  - Numeric: int, float
  - Character: char
  - User defined (with struct, union or enum)

Datatypes

For numerics, qualifiers specify if the quantity is signed or not and the number of bits used for storage:

- short unsigned int, short int
- unsigned int, int
- long unsigned int, long int
- unsigned char, char

Variables

- A variable is a name given to a value stored in the system memory or to an expression that can be evaluated
- Example:

```
int i = 0;
int j = 1;
int k = i + j;
```

Variable names

Naming rules:

- Variable names can contain letters, digits and underscores
- Variable names should start with letters
- Keywords of the language can not be used as variable names
- Variable names are case-sensitive. x and X refer to two different variables.

Variable declaration

Variable declaration follows the pattern: type name [= *value*]

```
char x; // unitialized
char x = 'a'; // initialized
unsigned short int i = 1;
```

Arithmetic Operators

- Addition: $+$
- Subtraction: $-$
- Multiplication: $*$
- Division: $/$
- Modulo: $\%$

Relational Operators

- Greater than: $>$
- Greater than or equal to: $\geq$
- Less than: $<$
- Less than or equal to: $\neq$
- Equal: $==$
- Different: $!=$

Logical Operators

- And: &&
- Or: ||
- Not: !

Increment Operators

- Post increment: `var++`. Meaning: read the value of the variable then increment it by one unit
- Pre increment: `++var`. Meaning: increment the value of the variable by one unit and then read its value.
- Post decrement: `var--`
- Pre decrement: `--var` with same meanings as post or pre increment (but for decrement)

Bitwise Operators

- &: bit by bit AND
- |: bit by bit OR
- ^: bit by bit XOR
- $\gg$: shift by given number of bits to the right (i.e. divide by power of two). Example: `x >> 5` shifts x by 5 bits to the right.
- $\ll$: shift by given number of bits to the left (i.e. multiply by power of two)

Assignment Operator

- $=$ Assigns the value of the right hand side expression to the variable on the left hand side.
- Example:

```
int x = 0;
// assign value of x plus 2
// to the variable named x
x = x + 2;
```

Statements

- Statements are valid expression terminated by a semicolon:

```
int x = a + b;
```

- Multiple statements consist in a list of statements:

```
int x = a + b;
x = x * c + d;
```

- Multiple statements can be grouped in a block; a block is delimited by curly braces: { and }

Blocks

- A block is a group of statements delimited by curly braces
- A block can substitute for a single statement
- It is compiled as a single unit; variables can be declared within this unit (their scope is delimited by the braces)
- Blocks can be nested

Conditional Statements

Conditional statements allow to branch to a statement depending on some condition. Condition statements in C are:

- if else
- switch case

If else statement

```
if (condition) {
 // statements
} else {
 // statements
}
```

Switch case statement

```
switch (input) {
 case 'a':
   // do something
   break;

 case 'b':
   // do something
   break;

 default:
   // execute if input is not 'a' or 'b'
}
```

Loops

Loop statements allow to repeat a statement as long as a condition is verified:

- for
- while
- do while

Special keywords allow to modify the normal flow of a loop:

- break: interrupt a loop before its normal termination
- continue: allow to continue to the next iteration of the loop while skipping the statements after the keyword continue

For

```
int i;
int sum = 0;
for (i = 0; i < n; i++) {
  sum = sum + i;
}
```

While

```
int i = 0;
int sum = 0;
while (i < n) {
  sum = sum + i;
  i = i + 1;
}
```

Function

- Functions allow to decompose a program in smaller units;
- A function is defined by:
  - its prototype, i.e. its name, the number and types of its argument and its return type
  - its body, i.e. a block grouping the statements to be executed by the function
- In C, a function can call itself. This is called recursion

Example of function

```
int gcd(int a, int b) {
 if (b > a) swap(&a, &b);
 if (b) return gcd(b, a % b);
 return a;
}
```

Modular programming

Module: interface and implementation

- interface: public data-structures, global variables and functions; found in header files
- implementation: source code for the public functions, internal functions, and definition of internal data-structures; found in c files

Extern keyword

- The keyword extern is used to inform the compiler that a variable has been alredy defined somewhere else
- This enables to access and modify global variables from other source files

Extern keyword

```
// IO.h
extern int file_counter;

// IO.c
int file_counter = 0;

int open(char* file_name) {
  // ...
  file_counter++;
}
```

Extern keyword

```
// main.c
#include "IO.h"

int main(void) {
  // ...
  if (file_counter > 50) {
    // ...
  }
}
```

Scope

- The region where a variable exists
- Generally, the block within which the variable is defined
- If a variable is defined outside of a function it is global
- Function definitions also have scope

Static scope

- Variable or function prefixed by the static keyword
- Outside of a function, it means that the variable (or function) is visible only within the file
- Inside a function, a static variable:
  - is local to the function
  - is initialized only once during the program initialization (i.e. it keeps the variable value from one call to another)

Register variable

- Variable prefixed by the keyword: register
- Only for local variables and for primitive types
- Excess or unallowed register variables will be compiled as regular variables (i.e. memory is reserved on the program stack)

Pointers

- Memory address of a variable
- The address can be used to access or modify the pointed variable from anywhere
- Usage for arrays, string (array of chars), complex data-structures (recursive data-structures like trees)

Address of variables

- Address is obtained by the operator: &
- Example:

```
int a = 1;
int* pa = &a;
double p = 1.0;
double* pp = &p;
```

- The followings don't have address:
  - register variables
  - constant

Access by dereferencing

- Variable pointed to is accessed by the dereferencing operator: *
- This allows us to read or modify the pointed variable
- Examples:

```
double p = 1.0;
double* pp = &p;
printf("%g\n", *pp); // read
*pp = p * 2.0; // modify
```

Pointers and function variables

- Modification of variables in the caller

```
void swap(int* a, int* b) {
 int tmp = *a;
 *a = *b;
 *b = tmp;
}
```

- Other type of usage: multiple output from a function

Arrays and pointers

- Primitive arrays as pointer to a block of memory:

```
int array[8];
int* pa = array; // or pa = &array[0]
```

Pointer arithmetic

```
pa + i; // or array[i]
pa + 1;
pa++;
int* pa2 = pa;
pa2--;
```

Pointer to Pointer

- Stores the address of a variable that itself stores the address of a variable in memory
- Example:

```
int a = 3;
int* pa = &a;
int** ppa = &pa;
```

Usage of Pointer to Pointer

- Array of strings (pointer to pointer to char)

```
char** string_array;
```

- Multidimensional array (pointer to pointer to some type)

```
int** matrix; // array of n x n int
```

Void Pointers

- void variables can not be declared in C
- void pointers can be used to point to any data type
- void pointers can not be dereferenced. Void pointers should always be cast before deferencing

```
int x = 1;
void* p = &x;
int* px = (int*) p;
printf("%d", *px);
```

Function Pointers

- Functions in C are not first order objects (so for example it is not possible to pass a function as an argument to another function)
- Instead it is possible to declare variables as pointers to function. Example:

```
int (*fp) (int);
```

- Function pointers can be manipulated like any variables, for example passed to functions

Callbacks

Callbacks are functions used to customize the behavior of some other functions. This is done by passing a function pointer to a given function. The sort function of the standard C library is an example of function customized by a callback:

```
void qsort(void* a, int n, int sz,
   int (*fp)(void*,void*));
```

The fourth argument is a pointer to a function that will be used to compare two given elements of the input array. (Note the use of void pointer to have a generic qsort function).

Structure

- Collection of related variables (of possibly different types) grouped together
- This defines a new data-type
- Examples:

```
struct Employee {
  char name[100];
  char first_name[100];
  int age;
};
```

Structure

- Variables declared within a structure are called its members
- Variables of type struct are declared as:

```
struct Employee employee; // employee is the var name
```

- Members are accessed with the dot notation:

```
employee.age = 10;
```

- Assignment copies every member bitwise. Care should be taken with pointers.

Structure

```
struct Employee employee = {"Doe", "John", 10};
struct Employee employee2 = {
      .age=10, .name="Doe", .first_name="John"
};
```

Recursive structure

- A structure can be defined recursively.
- Example of a binary tree:

```
struct Node {
  int key;
  struct Node* left;
  struct Node* right;
};

typedef struct Node * Tree;
```

Union

A variable that can hold objects of different types in the same memory location:

```
union data {
 int i;
 float f;
};
union data d;
d.i = 10;
```

Union

The compiler does not verify that the format used for reading is the same format as the format used for writing

```
union data d;
d.i = 10;
float f = d.f; // junk
```

Memory allocation

- Memory allocation is done with malloc() from the standard library

```
void* malloc(size_t);
// Example of usage:
int* vector = (int*)malloc(10*sizeof(int));
```

- Note the use of the operator sizeof to get the size in memory of a given datatype
- Memory allocated with malloc is located on the heap of the process

Memory allocation

- Memory allocated with malloc should be released with free

```
int* vector = (int*)malloc(10*sizeof(int));
// ...
free(vector);
```

- Can only free regions that were dynamically allocated (i.e. by malloc, calloc or realloc)
- Can call free only once on a dynamically allocated region (e.g. don't write `free(vector); free(vector);`