COMP 356
Programming Language Structures
Notes for class 3: Programming in C

1 Basics of the C programming language

Many good integrated development environments (IDEs) exist for programming in C and C++. On Windows, Visual Studio Express is popular (and free). On other platforms, options include Eclipse (with a C/C++ plug-in installed—this is available on the Tome machines), Code::Blocks, CodeLite. You're free to use any of these if you wish, but the instructor can't help with any questions about these environments. Instead...

In this class I ask that you learn to use a combination of a text editor and command line tools for writing and debugging code in C and C++. Any decent text editor is fine (do not use Notepad or TextEdit). On the Tome machines, I personally recommend Aquamacs, and pico is another option. On Windows and Linux, I personally recommend Emacs. But there are many other good text editors so feel free to use one that you are comfortable with.

To compile a C program from the command line, we will use the Gnu C compiler, called `gcc`. To compile `foo.c` into `foo.exe`, use

```
gcc -o foo.exe foo.c
```

To run `foo.exe`, use

```
./foo.exe
```

Exercise: type in, compile, and run a hello world program in C.

Exercise: write a program that uses a nested loop to produce a multiplication table for an integer entered as a commandline argument. For example, if the command line is `./multTable.exe 3` the output would be:

```
1 times 1 is 1
1 times 2 is 2
1 times 3 is 3
2 times 1 is 2
2 times 2 is 4
2 times 3 is 6
3 times 1 is 3
3 times 2 is 6
3 times 3 is 9
```

To complete the above exercises, you will need to know the following basic features of C. They are mentioned very briefly in these notes, and will be explained a little further in class. For further details, use your own online research. There are many excellent online tutorials and repositories for this information.

Basic features:

- `#include <stdio.h>`: include the *header file* "stdio.h", which makes available various standard input-output functions, such as `printf`.
- `printf("hello\n")`: the "\n" prints a newline character.

- `printf("The value of integer x is %d\n", x)`: the "%d" is a placeholder for an integer variable, which you give as a parameter.
- `printf("x is %d and y is %d\n", x, y)`: you can use multiple placeholders with multiple parameters. Use `%g` for floating-point values and `%s` for strings.
- `int main(int argn, char* argv[])`: this is the signature of the `main` function in C. `argn` is the number of commandline arguments; `argv[n]` is the nth commandline argument represented as a string (starting at $n = 0$, which is the name of the program itself).
- `int x = atoi("6")`: the function `atoi` converts a string representation of an integer into an integer datatype, so x gets the value 6 in this example.

2 Pointers in C

Unlike Java, C and C++ include facilities for directly accessing and manipulating the computer's memory. This is achieved using a data type called a *pointer*. Pointers are declared using the `*` character. Examples:

```
int* x;  // a variable named x that is a pointer to an integer
int *y;  // a variable named y that is a pointer to an integer
char* z; // a variable named z that is a pointer to a character
```

Another way to create a pointer is to take the *address* of a variable. This is done using the `&` character. e.g. (continuing previous example)

```
int val = 8;
printf("The address of val is %p\n", &val);
x = &val;
char c = 'A';
z = &c;
```

To access the value that a pointer variable points to, you need to *dereference* the variable, using the `*` character. e.g. (continuing previous example)

```
int w = *x;
*z = 'B';
```

3 Memory management: the stack and the heap

In this course, *subprogram* is the general term used for a method, function, procedure, or subroutine in a programming language.

By default, local variables in a subprogram are stored in a data structure known as the *call stack* or *runtime stack*, or just the *stack*. Memory for these variables is allocated when the subprogram starts, and is automatically deallocated when the subprogram exits. The operating system manages the call stack automatically.

Another important place for storing variables is known as the *heap*. The operating system provides some functionality for interacting with the heap, but it does not manage the heap memory automatically. Instead, heap memory is managed either by the programmer (in languages like C and C++) or by some extra facilities in the programming language (e.g. Java, Python).

Question: why would you ever allocate something on the heap? Why not just use the stack for everything?

In C, you allocate heap memory using the `malloc` function, usually with help from the `sizeof` function (and both require `#include <stdlib.h>`). e.g.

```
int* x = malloc(sizeof(int));
*x = 234;
```

Memory allocated using `malloc` will not be automatically reclaimed while the program is running (the operating system will reclaim all program memory when the corresponding process terminates). The programmer must ensure that it is deallocated using the `free` function. e.g. (continuing previous example)

```
free(x);
```

Failure to free some memory leads to a *memory leak*. e.g.

```
void doSomething() {
  int* x = malloc(sizeof(int));
  *x = 234;
  int y = *x;
}
```

After the function `doSomething()` exits, there is no way to reclaim the memory allocated to `x`. If the function were called 1 billion times during the execution of a program, the program would end up consuming at least 1 billion `int`s-worth of memory—which on a typical 64-bit machine would be 8 gigabytes.

4 Arrays and pointer arithmetic in C

Declaring an array on the stack is similar to Java. e.g.

```
int x[5];
int y[] = {23, 57, 83, 29};
x[2] = 5;
y[0] = x[2];
```

Array variables in C are actually pointer variables. e.g. the data type of `x` above is `int*`.

Declaring an array on the heap requires the programmer to calculate the size of the array. e.g. (continuing previous example)

```
int* z = malloc(sizeof(int)*7); // a 7-element array of ints
```

C permits calculation of addresses using *pointer arithmetic*. e.g. (continuing previous example)

```
int* a = z + 3; // *a is z[3], and a[2] is z[5]
*(a+1) = 251; // sets z[4] to value 251
int* b = &(z[3]); // b == a
```

The special value `NULL` is used for a pointer that doesn't point to anything:

```
char* x = NULL;
```

In C, strings are just arrays of characters:

```
char* x = "abcde";
x[2] = 'Z'; // now the string x is "abZde"
```

In C, a `char` is a byte (i.e. 8 bits). Strings are terminated with a byte containing the value 0. For example, the above string `x` has `x[5]==0`.

5 Output parameters

By using pointers, information can be returned from a function by altering the parameters of the function—or more explicitly, but altering the values *pointed to* by the parameters of the function. Example signatures for doing this are:

```
void getXandY(/*out*/ double *x, /*out*/ double *y)
void incrementMandN(/*in-out*/ int *m, /*in-out*/ int *n)
```

In `getXandY()` the two parameters are *output parameters*. Their initial values do not affect the function, but new values are returned to the caller in these parameters. In `incrementMandN()`, the two parameters are *input-output parameters*. Their initial values are used by the function, but new values are also returned to the caller.

6 Everything else

Consult the presentation C for Java Programmers by Niranjan Nagarajan for details on all the other elementary C constructs required for the course. (Except for unions and enums, which are covered in lecture notes for the next class.)