Faculty of Computer Science, Dalhousie University *9-Nov-2018*

CSCI 2132 — Software Development

Lecture 28: Structures and Dynamic Memory Allocation

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- prj-dec2bin and stack example
- Compilation of large programs
- make utility (started)
- make utility
- Using gdb with multi-file programs
- Structures (started)

The following piece of code declares a structure that can be used to store a student record:

```
struct student {
  int number;
  char name[26];
  char username[11];
} x, y;
```

Here we assume that a student number is an integer. In this declaration, student is the tag of this structure type. A tag is optional, but with a tag, we can use "`struct tag variable_name;`" to declare more variables of this structure type after this type definition, as shown below. This declaration also defined two variables, `x` and `y`, of the type `struct student`. This variable list is optional since we can just declare the type `struct student` and use it later to define variables.

Now let us make use of the tag to declare another structure variable, a pointer and an array:

```
struct student z, *p, first_year[200];
```

To access a member of the structure, we can use the dot operator (`.`). For example, if we wish to assign values to the members of the variable `z` defined above, we can write:

```
z.number = 123456;
strcpy(z.name, "John King");
```

Slide notes:

Example Code

```
x.number = 123;
strcpy(x.name, "Dennis Ritchie");
strcpy(x.username, "dritchie");
y = x;

#define PRINT \
  printf("number:%d\nname:%s\nusername:%s\n\n", \
  x.number, x.name, x.username); \
  printf("number:%d\nname:%s\nusername:%s\n\n", \
  y.number, y.name, y.username);

PRINT
x.number = 456;
strcpy(x.name, "Ken Thompson");
strcpy(x.username, "kthompson");
PRINT
```

There is another operator called *arrow* operator (`->`), and it is the shorthand for using dereference and dot operators on a pointer to a structure. Read the following piece of code, in which p and z are variables defined above:

```
p = &z;
(*p).number = 222333;
```

The second line of the code is equivalent to:

```
p->number = 222333;
```

Structure can also be used as function parameters. It is however not always a good idea to pass a large structure by value to a function: This requires the copying of the values of all the members of the structure argument to corresponding members of the structure parameter, which is time-consuming. To address this efficiency issue, we often pass pointers to structures to a function. We will see how to do this soon when implementing linked lists.

23 Dynamic Memory Allocation

When we implement dynamic data structures, as the number of data elements stored in these data structures changes during program execution, we need acquire and free blocks of memory during program execution. Dynamic memory allocation makes this possible.

In C, dynamic memory allocation is done using functions defined in `stdlib.h`. The function for acquiring a block of memory is `malloc`:

```
void* malloc(size_t size);
```

This function returns a pointer to an unused memory block of size bytes, or the NULL pointer if the system is out of memory. In this prototype, `void *` is a "generic" pointer, which is just a memory address. To use `malloc` properly, we need always check whether it succeeded in allocating a block of memory, and write code similar to:

```
int *p  = (int*) malloc(10000*sizeof(int));
if (p == NULL ) {
```

```
  ... /* Error */
}
```

To free a block of dynamically allocated memory that the program no longer uses, we use another function called `free`:

```
void free(void *ptr);
```

This frees the memory block pointed to by ptr. This function can be used only for memory allocated by malloc. You cannot use it to free memory that is not dynamically allocated. After executing this function, ptr will become a dangling pointer: free does not change the value of ptr itself. It still points to the memory block that is freed. In our programs, we should never access memory that has been freed, as they might be used to store some other data that we have no knowledge of.

If we use dynamic memory allocation improperly, our programs may create garbage, which is a memory block that is no longer accessible to a program. This can happen if we have a pointer that is the only pointer pointing to a dynamically allocated block of memory, and we then make it point to another variable without freeing this memory block first. C does not do automatic garbage collection, which is a feature of Java. This is because automatic garbage collection undermines the time and space efficiency of programs, and since efficiency is the top priority of C, programmers are required to do garbage collection themselves by freeing blocks of memory when our program no longer accesses them. A program that leaves behind garbage has a memory leak, and we must always avoid memory leaks.