# Lecture 4: C Programming (Structures, Memory, Strings)

**Dr. Naureen Nizam**
nnizam@cs.toronto.edu

CSCB09H3: Software Tools and Systems Programming (SY 110)
Department of Computer and Mathematical Science
January 28/30, 2019

UTSC Tomorrow is created here.
UNIVERSITY OF TORONTO SCARBOROUGH
1265 Military Trail, Toronto, Ontario M1C 1A4

## Administrivia

- **Labs**
  - Lab 1, 2 solution posted
  - Lab 3 posted
- Midterm
  - Monday, **February 25** (in-class) both Lec 01 and Lec 30
  - Make-up (only those approved) – Wednesday **February 27th** (in-class)



## Last Week

- C intro
- Basic C data types
- Arrays
- Pointers

KING Chapters 1-8, 11, 12



## The plan for today

- Strings — KING Chapters 13
- Dynamic memory management — KING Chapters 17
- Structures — KING Chapters 16



## Strings

- Strings are not a built-in data type
- A string is an array of characters terminated with a null character (‘\0’).
- Initializing a string:

```
char course_name[8] = {'c','s','c','b','0','9','h','\0'};
```

Or more conveniently:

```
char course_name[8] = "cscb09h";
```

Now you can do all the usual array operations, e.g.

```
course_name[3] = 'z';
```



## Common string operations

- What is the length of a string?
- Copy a string
- Concatenate two strings
- Compare two strings
- Search a string for occurrence of a character
- Search a string for a substring
- C has no built-in support, but many string functions are provided in a library.
  - Try `man string`
  - `#include<string.h>`

# What is the length of a string?

•E.g. : `char course_name[50] = "cscb09h";`

- The length is the number of non-null characters
  –7 in the example above
  –Remember: The array needs to be (at least) one bigger than that
- Library function that returns length of a string:
  - `strlen (const char *str)`
- Not the same as size (= storage requirement)
  –50 bytes in the example above (1 char is 1 byte)
  –`sizeof(course_name)` returns 50



# Copying a string

•Two library functions:

```
char *strcpy (char *dest, char * src)

char *strncpy(char *dest, const char *src, int n)
```

*Like strcpy, but copies at most n characters from src.*



# Copying a string

```
char s1[3];
char s2[5] = "abcd";
strcpy (s1, s2);
  //Overflow
strncpy (s1, s2, strlen(s2));
  // Overflow


strncpy (s1, s2, sizeof(s1) -1);  // Correct
s1[sizeof(s1)-1] = '\0';
```



# Concatenating a string

•`char *strcat (char *dest, const char *src)`
- Appends src to dest (including the null byte of src), overwriting the null byte of dest.

•`char *strncat(char *dest, const char *src, int n)`
- Like strcat, but takes at most n characters from src (up to null byte)
- If src has >= n characters, it takes n characters and adds null byte

•Both return a pointer which is usually ignored because it equals dest.
•Problem with strcat?
- Unsafe, if src is too long

```
char s1[6] = "abc";
strncat(s1, "def", 6);  // Overflow
// n should be <=  sizeof(s1) - strlen(s1) - 1
```



# Comparing string

•`int *strcmp (const char *s1, const char *s2)`
- Returns negative number, 0, or positive number if s1 is <, =, or > s2, respectively.

•`int *strncmp (const char *s1, const char *s2, int n)`
- Same, but compares only the first (at most) n characters.



# Searching for characters

```
char *strchr(const char *s, int c)
   // finds the first occurrence

char *strrchr(const char *s, int c);
   // finds the last occurrence
```

•Both return a pointer to the character if found, NULL otherwise.

## What do you think about C so far …?

– Disappointed about the lack of support for strings, booleans, arrays, etc…. ?
– Turns out you also have to do much of memory management manually ….

Your ride with Python:
- Cruise control
- Seat heating
- Cup holders
- Automatic transmission

Your ride with C:
-Performance and speed
-But, no amenities …
-And it only comes with a stickshift …



## Dynamic memory management – why?

•When declaring an array we have to specify its size.
- E.g. `int my_array[100];`

•What if we don't know the size of an array in advance?
•What if we realize later that we need a bigger array?



## Dynamic memory management – why?

•Imagine we want to write a function `concat` that takes two strings, and returns their concatenation as a new string.

•We would use it as follows:

```
char *s;
s = concat ("abc", "def");
```



## Dynamic memory management – why?

•Here is an attempt at writing `concat`:

```
char *concat(const char *s1, const char *s2) {
      char result[70];
      strcpy (result, s1);
      strcat (result, s2);
      return result;
}
```

Any problems with this implementation?
- `strncpy & strncat` would be safer..
- But what else?



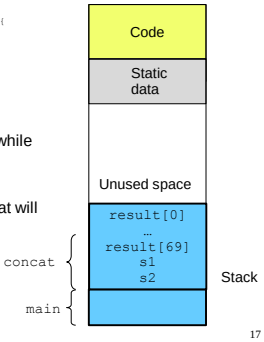
## Recall memory management from last week

```
char *concat(const char *s1, const char *s2) {
      char result[70];
      strcpy (result, s1);
      strcat (result, s2);
      return result;
}
```

•The memory for `results` exists only while `concat` is running.

•Need to be able to allocate memory that will not be lost when concat finishes.

•How?



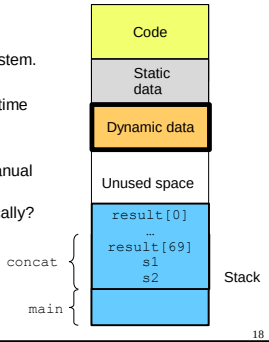



## Remember the dynamic data segment in the address space?

•Memory allocated here will never be released /freed automatically by the system.
- Completely under programmers control.

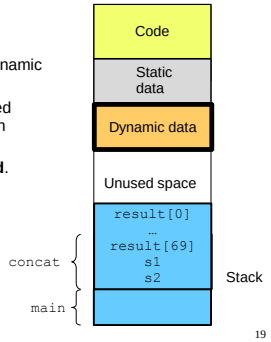
•Memory here can be allocated at any time during the run of a program.

•Dynamic memory allocation allows manual control of memory allocation!
•How do you allocate memory dynamically?



## Malloc()

- void *malloc (`size_t size`);
- Malloc allocates `size` bytes in the Dynamic Data segment
- Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
- The allocated memory is **uninitialized**.





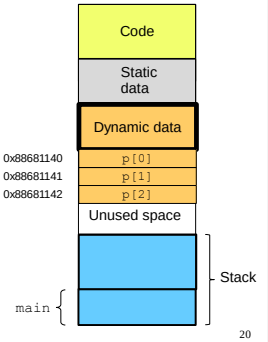

## Malloc()

- void *malloc (size_t size);
- Example:

```
char *p;
p = malloc(3);
```

- Allocates 3 bytes and p will contain the address of the first byte (p == 0x88681140).
- Can use it like any character array.
- E.g.

```
strcpy(p, "ab");
```






## A correct `concat`

```
char *concat(const char *s1, const char *s2) {
     char *result;
     result = malloc(strlen(s1) + strlen(s2) + 1);
     if (result == NULL) {
         printf ("Error: malloc failed\n");
         exit(EXIT_FAILURE);
     }
     strcpy (result, s1);
     strcat (result, s2);
     return result;
}
```



## Freeing memory

- Memory allocated via malloc is not released automatically.
  - Other non-malloced memory is freed automatically when it goes out of scope
- If we keep calling malloc without releasing any of the memory, memory will run out..
- Need to free un-needed memory using `free`:

```
char *p;
p= malloc (n+1);
// doing lots of fun stuff with p here …
…
// done with p
free (p);
```



## Memory can leak …

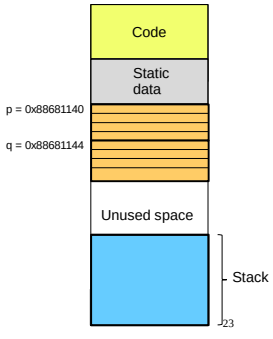
- Consider:

```
p = malloc(…)
q = malloc(…)
```

- Next consider:

```
p=q;
```

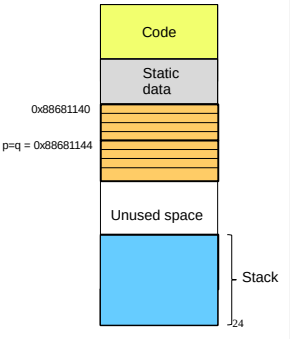
What happens?






## Memory can leak …

- Problem we have no way of accessing p's old block.
- We also have no way of freeing p's old block.
- This is called a memory leak.
- One of the most common programming errors.



## Dangling pointers

```
char *p = malloc(5);
…
free (p)
```

•p is now a pointer to memory it does not own.

```
strcpy(p,"abcd");   // Bad things can happen
```

## Allocating space for other datatypes

•So far we only allocated space for `chars`

•How would you allocate space for an int array with 4 elements?

```
int *a = malloc(???);
```

•The size parameter `malloc` expects is in bytes (which btw. is also the size of one `char`).

•For anything besides `char`, use `sizeof` to obtain the size of one element:

```
int *a = malloc (4 * sizeof (int));
```

## Other types of allocation

•In addition to malloc, there are two other functions to dynamically allocate memory:

•`void *calloc(size_t nmemb, size_t size);`

– similar to malloc, but zeros allocated memory

•`void *realloc(void *ptr, size_t size);`

– changes the size of the memory block pointed to by `ptr` to `size` bytes.
– `ptr` must point to memory previously allocated by `malloc, calloc`.

## Congrats!

•You have survived pointers and dynamic memory management.

•These are the two hardest topics in learning C programming.

•Make sure you get lots of practice!

– Driving a stickshift takes practice
– Reading the technical specs does not replace practice..

•Even if the theoretical concepts seem clear, implementing bug-free code is a different story …

•Don't skip the labs!


## Structs

•Build your own data type!

•A struct is a collection of related data items

•E.g. an application managing a database of a department's students could define the following struct with 3 members:

```
struct student {
   char firstName[20];
   char secondName[20];
   int year;
}
```

## Working with structs

```
struct student {
    char firstName[20];
    char secondName[20];
    int year;
 };
```

Now we can use `struct student` as a datatype, just like an `int, float, ….`
To make some students and modify their members:

```
struct student my_first_student;
my_first_student.year = 3;
strcpy (my_first_student.firstName, "Dan");
strcpy (my_first_student.lastName, "Jones");
```

## Pointers to structs

```
struct student student1;
…… // code for initializing student1 here
struct student *p;
p = &student1;
```

How can we access `student1`'s members?

```
student1.year = 3;
(*p).year = 3;
p->year = 3;
```

## Structs and malloc

- Struct can be used with malloc like any other datatype:

```
struct student *s;
s = malloc (sizeof (struct student));
s->year = 3;
```

## Structs and functions

- Functions can take structs as parameters and access their elements:

```
void PrintStudent (struct student s) {
        printf ("First name: %s\n", s.firstName);
        printf ("Last name: %s\n", s.lastName);
        printf ("Year: %s\n", s.year);
}
```

## Pointers and functions

- Modifying the arguments of a function:

```
void ChangeStudentYear (struct student s, int y) {
    s.year = y;
};

void SwapNumbers (int x, int y) {
    int temp = x;
    x = y;
    y = temp;
};
```

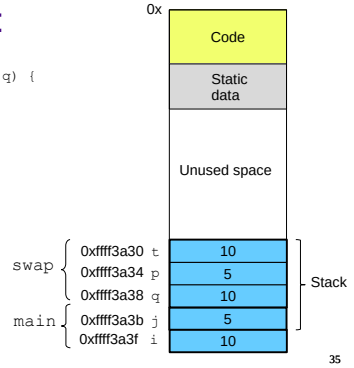
What's wrong with these examples?

## Remember from last time:

```
void swap(int p, int q) {
    int t = p;
    p = q;
    q = t;
}

int main() {

    int i = 10;
    int j = 5;
    swap(i, j);
    return 0;
}
```






## Pointers and functions

```
// WRONG:
void ChangeStudentYear (struct student s, int y) {
    s.year = y;
};
```

- Parameters in C are *passed by value*, i.e. a copy of each argument is passed to the function.
- The functions works on the copy, not the original variable.
- Solution: pass a pointer!

```
// CORRECT:
void ChangeStudentYear (struct student *s, int y) {
    s->year = y;
};
```

## Pointers and functions

```
// WRONG:
void SwapNumbers (int x, int y) {
     int temp = x;
     x = y;
     y = temp;
};

// CORRECT:
void SwapNumbers(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

Call by: SwapNumbers(&a, &b);
```

## One more thing to think about …

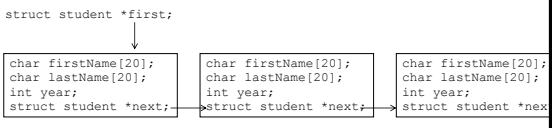
- Going back to the example of the student database:
- We need to manage many students
- The number of students will grow over time
- How do we manage all the different student variables?

- One large array, e.g. `struct student[1000]` ?
- Disadvantages?
  - Might be too big: wastes space not needed
  - Might be too small: requires frequent realloc to adjust (expensive!)

## Idea: Create a linked list

- We only allocate new space when a new student is added to the system, i.e. malloc for one student struct.
- How do we manage all the pointers to the different structs we created. In array? Same disadvantages as before ..

- Instead: in each struct keep a pointer to the following struct!




Think about this and read King, chapter 17.5 (Linked lists)!

## That's it for today!