1. **Strings:**
   - Strings are not a built-in data type.
   - A string is an array of characters terminated with a null character ('\0'). Therefore, the size of the array must be 1 larger than the length of the string.
   - Initializing a string (Either way works):
     - **char course_name[8] = {'c','s','c','b','0','9','h','\0'};**
     - **char course_name[8] = "cscb09h";**
   - C has no built-in support, but many string functions are provided in a library. We can use **#include<string.h>** to access the built in functions.
   - Common string operations:
     - Length of a string:
       - The length is the number of non-null characters.
       - E.g. Suppose we have
         char course_name[50] = "cscb09h";
         The length of the string is 7, because there are 7 non-null characters.
       - **Note:** The length of a string is not the same as the size of the array. In the example above, the size of the array is 50, while the length of the string is 7.
       - Library function that returns length of a string:
         strlen (const char *str)
       - Example of strlen:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (){
   int len;
   char x[8] = "cscb09";
   len = strlen(x);
   printf("Length of %s is %d\n", x, len);
   // The output is "Length of cscb09 is 6"
   return 0;
}
```

     - Copying a string:
       - Two library functions:
         1. **char *strcpy (char *dest, char * src)**
         2. **char *strncpy(char *dest, const char *src, int n)**
            Like strcpy, but copies at most n characters from src
       - Example of strcpy and strncpy.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main (){
   char src[50], dest[50];
   strcpy(src,  "This is source");
   strncpy(dest, "This is destination", 10);
   printf("This is src : %s\n", src);
   // Prints "This is src : This is source"
   printf("This is dest : %s\n", dest);
   // Prints "This is dest : This is de"
   return 0;
}
```

- Concatenating two strings:
  - **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.
  - The problem with strncat is that it is unsafe, if src is too long. E.g. Consider the code below:
    char s1[6] = “abc”;
    strncat(s1, “def”, 6);
    The second line would cause an overflow because the size of the array has to be at least 1 bigger than the size of the string. Therefore, n should be <= sizeof(s1) - strlen(s1) - 1.
  - Example of strncat.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (){
   char src[50] = “This is source”;
   char dest[50] = “This is destination”;
   strncat(dest, src, 14);
   printf("Final destination string : %s", dest);
   // Prints "Final destination string :This is
   destinationThis is source"
   return 0;
}
```

- Example of strcat.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int min(int num1, int num2);

int main (){
   char src[50] = "This is source";
   char dest[50] = "This is destination";
   strcat(dest, src);
   printf("Final destination string : %s", dest);
   // Prints "Final destination string : This is
   destinationThis is source"
   return 0;
}
```

- Comparing two strings:
  - **int *strcmp (const char *s1, const char *s2)**
  - if Return value < 0 then it indicates str1 is less than str2.
  - if Return value > 0 then it indicates str1 is more than str2.
  - if Return value = 0 then it indicates str1 is equal to str2.
  - **int *strncmp (const char *s1, const char *s2, int n)**
    Same, but compares only the first (at most) n characters.
  - Example of strcmp.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main (){
   char str[10] = "abcdefg";
   char str2[10] = "bcdefgh";
   int ret;

   ret = strcmp(str, str2);

   printf("%d", ret); // Prints -1

   return(0);
}
```

  - Example of strncmp.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int main (){
   char str[10] = "abcdefg";
   char str2[10] = "bcdefgh";
   int ret;

   ret = strncmp(str, str2, 1);

   printf("%d", ret); // Prints -1

   return(0);
}
```

- Searching a string for occurrence of a character:
  - **char *strchr(const char *s, int c)**
    This finds the first occurrence.
  - **char *strrchr(const char *s, int c);**
    This finds the last occurrence.
  - Both return a pointer to the character if found, NULL otherwise.
  - Example of strchr.

```
#include <stdio.h>
#include <string.h>

int main () {
  int len;
  char str[] = "abcdefghc";
  char ch = 'c';
  char *ret;

  ret = strchr(str, ch);

 printf("String starting from the first %c is : %s", ch, ret);
 // Prints "String starting from the first c is : cdefghc"

  return(0);
}
```

- Example of strrchr.

```
#include <stdio.h>
#include <string.h>

int main () {
  int len;
  char str[] = "abcdefghc";
  char ch = 'c';
  char *ret;

  ret = strrchr(str, ch);

  printf("String starting from the last %c is : %s", ch, ret);
  // Prints "String starting from the last c is : c"

  return(0);
}
```

2. **Dynamic Memory Management:**
   - An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it. Sometimes, the size of array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
   - Memory allocated here will never be released /freed automatically by the system. This is 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.
   - **Malloc:**
     - Used to allocate space in memory during the execution of the program.
       I.e. Malloc allocates size bytes in the Dynamic Data segment.
     - Does not initialize the memory allocated during execution. It also carries garbage value.
     - Returns a pointer to the newly acquired memory, or NULL if there is not enough available memory.
     - Syntax: **`void *malloc(int num);`**
     - This function allocates an array of num bytes and leave them uninitialized.
     - The size parameter malloc expects is in bytes, which is also the size of one char.
     - For anything besides char, use sizeof to obtain the size of one element.
     - E.g.
       `int *a = malloc (4 * sizeof (int));`

- **Calloc:**
  - Calloc is also like malloc, but calloc initializes the allocated memory to zero while malloc doesn't.
  - Furthermore, malloc allocates a single block of memory. Whereas, calloc allocates multiple blocks of memory.
  - Syntax: **void *calloc(int num, int size);**
- **Realloc:**
  - Modifies the allocated memory size by malloc and calloc to new size.
  - If enough space doesn't exist in memory of current block to extend, a new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.
  - Syntax: **void *realloc(void *address, int newsize);**
  - This function re-allocates memory extending it upto newsize.
- **Free:**
  - 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.
  - Frees the allocated memory by malloc, calloc and realloc and returns the memory to the system.
  - Syntax: **void free(*address);**
  - This function releases a block of memory block specified by address.
- **Dangling Pointers:**
  - A pointer pointing to a memory location that has been deleted or freed is called dangling pointer. The pointer does not point to a valid object. This is sometimes referred to as a **premature free**.
  - The use of dangling pointers can result in a number of different types of problems, including:
    - Unpredictable behavior if the memory is accessed.
    - Segmentation faults when the memory is no longer accessible.
    - Potential security risks.
  - These types of problems can result when memory is accessed after it has been freed.
- **Memory Leak:**
  - Memory leak occurs when programmers create a memory in heap and forget to delete it.
  - To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

Memory can leak …

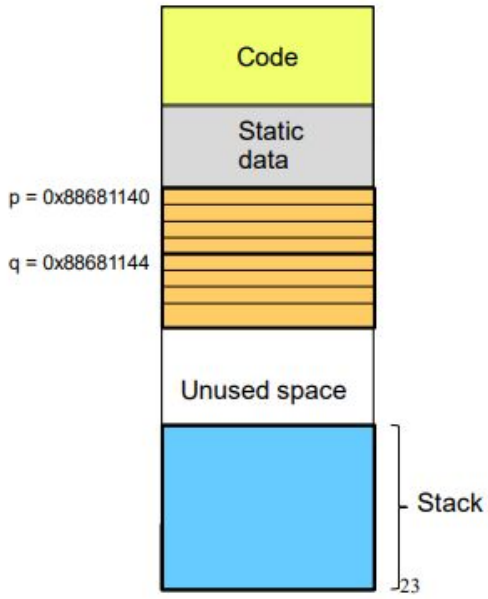
- Consider:

```
p = malloc(…)
q = malloc(…)
```

- Next consider:

```
p=q;
```

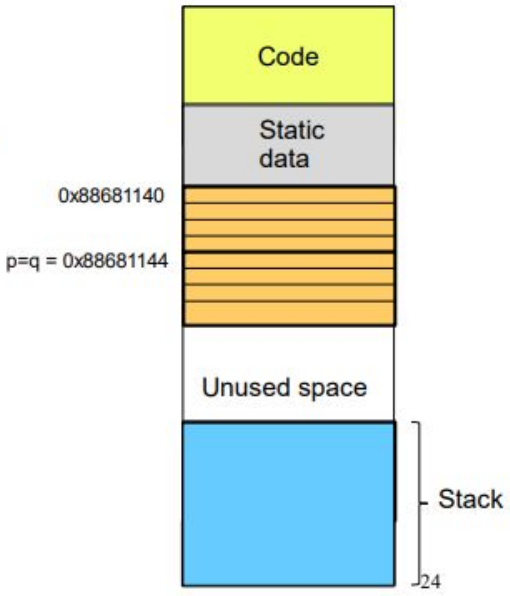
What happens?



23

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.



24

3. Static VS Dynamic Memory Allocation:

Static memory allocation	Dynamic memory allocation
Variables get allocated permanently and allocation is done before program execution (compile time).	Variables get allocated only if your program unit gets active and allocation is done during program execution (run time).
Consists of fixed sizes that cannot change throughout the program. I.e. Memory size can't be modified during execution.	Infinitely flexible. I.e. Memory size can be modified during execution.
Less efficient	More efficient
There is no memory reusability.	There is memory reusability and memory can be freed when not required.
Uses a stack for implementing static allocation.	Uses a heap for implementing dynamic allocation.

4. Structs:

- A struct is a collection of related data items. It is a collection of variables (can be of different types) under a single name.
- Structs are used for:
  - Serialization of data.
  - Passing multiple arguments in and out of functions through a single argument.
  - Data structures such as linked lists, binary trees, and more.
- Syntax:

```
struct [name] {
Member definition
Member definition
...
Member definition
} [variable name]; // The variable name is optional.
```

- Example:

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

- Struct members **cannot be** initialized with declaration. The reason for this is that when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.
  E.g. The code below will fail and cause an error.

```
struct Point
{
  int x = 0;
  int y = 0;
};
```

- 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:
  - If we want to modify the arguments of a function, we need to pass a pointer. This is because parameters in C are **passed by value**, which means that the functions works on the copy, not the original variable.