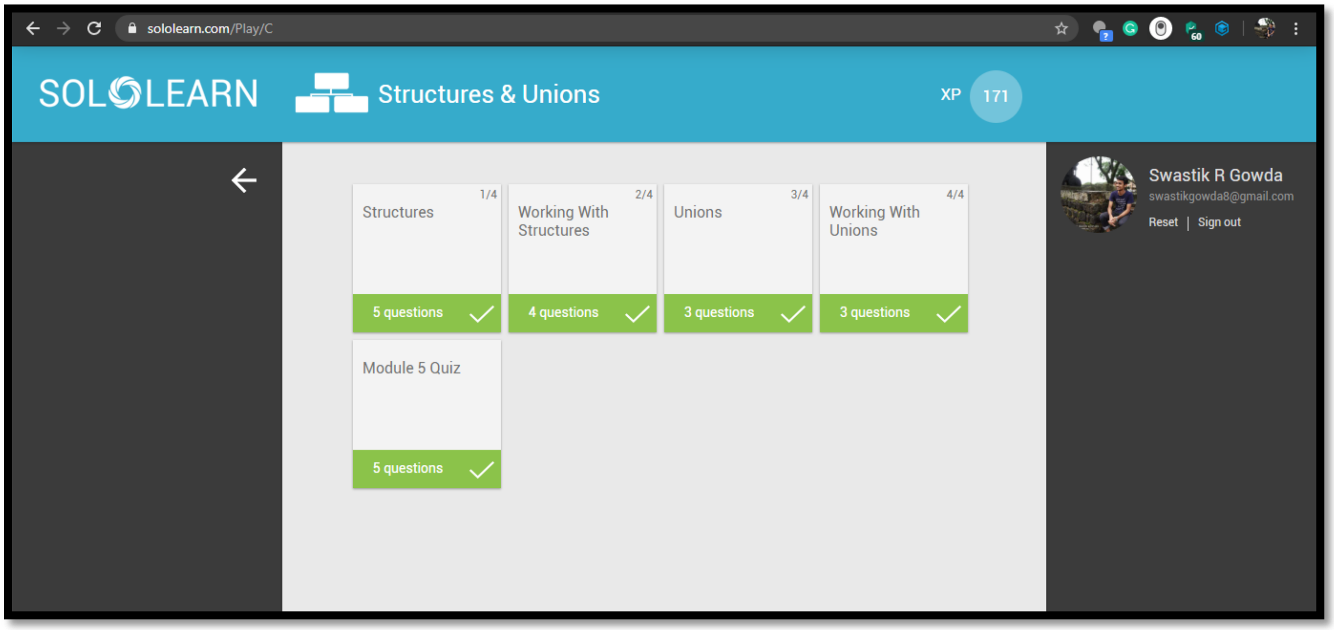
DAILY ASSESSMENT

Date:	19-June-2020	Name:	Swastik R Gowda
Course:	Solo-Learn	USN:	4AL17EC091
Topic:	❖ Structures & Unions ❖ Memory Management	Semester & Section:	6th Sem 'B' Sec
Github Repository:	swastik-gowda		

SESSION DETAILS
Image of session

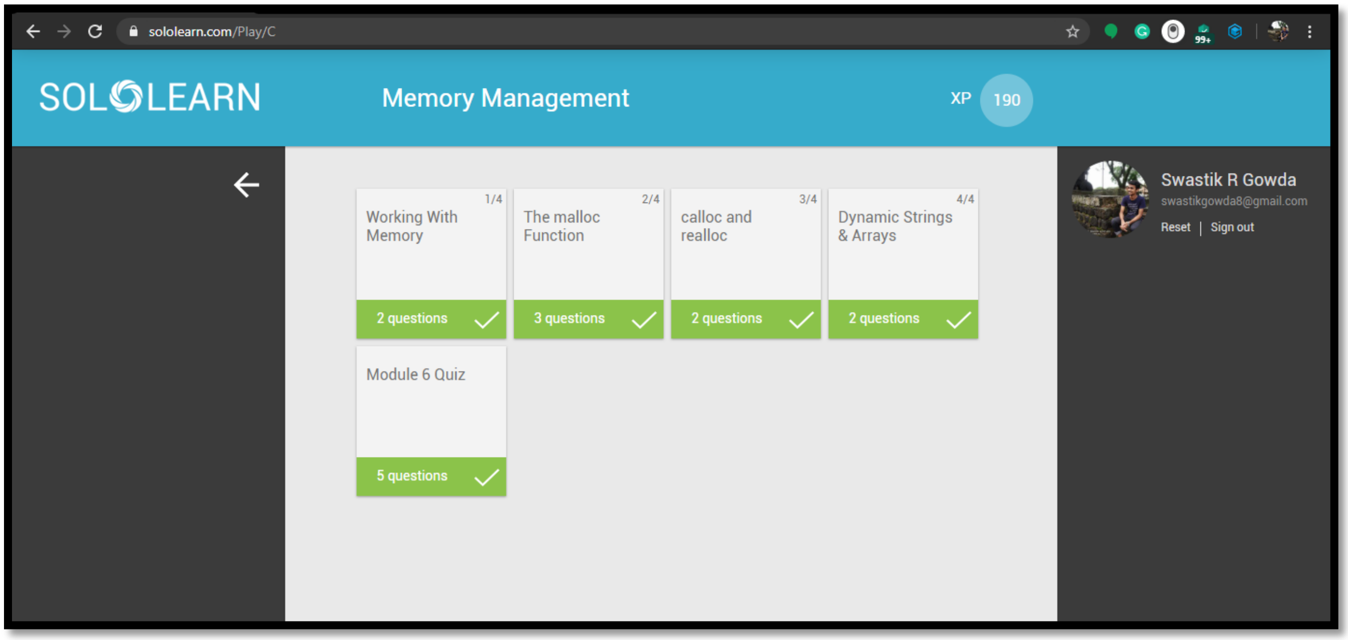


sololearn.com/Play/C

SOLOLEARN Structures & Unions XP 171

Swastik R Gowda
swastikgowda8@gmail.com
Reset | Sign out

Structures 1/4 — 5 questions ✓	Working With Structures 2/4 — 4 questions ✓	Unions 3/4 — 3 questions ✓	Working With Unions 4/4 — 3 questions ✓
Module 5 Quiz — 5 questions ✓			



sololearn.com/Play/C

SOLOLEARN Memory Management XP 190

Swastik R Gowda
swastikgowda8@gmail.com
Reset | Sign out

Working With Memory 1/4 — 2 questions ✓	The malloc Function 2/4 — 3 questions ✓	calloc and realloc 3/4 — 2 questions ✓	Dynamic Strings & Arrays 4/4 — 2 questions ✓
Module 6 Quiz — 5 questions ✓			

Report – Report can be typed or hand written for up to two pages.

Structures

A **structure** is a **user-defined data type** that groups related variables of different data types.

A structure **declaration** includes the keyword **struct**, a **structure tag** for referencing the structure, and curly braces { } with a list of variable declarations called **members**.
For example:

```
struct course
{
        int id;
        char title[40];
        float hours;
};
```

This struct statement defines a new data type named **course** that has three members.
Structure members can be of any data type, including basic types, strings, arrays, pointers, and even other structures

Declarations Using Structures

To declare variables of a structure data type, you use the keyword ***struct*** followed by the struct tag, and then the variable name.
For example, the statements below declares a structure data type and then uses the student struct to declare variables s1 and s2:

```
struct student
{
         int age;
         int grade;
         char name[40];
};
/* declare two variables */
struct student s1;
struct student s2;
```

Declarations Using Structures

A struct variable can also be initialized in the declaration by listing initial values in order inside curly braces:

```
struct student s1 = {19, 9, "John"};
struct student s2 = {22, 10, "Batman"};
```

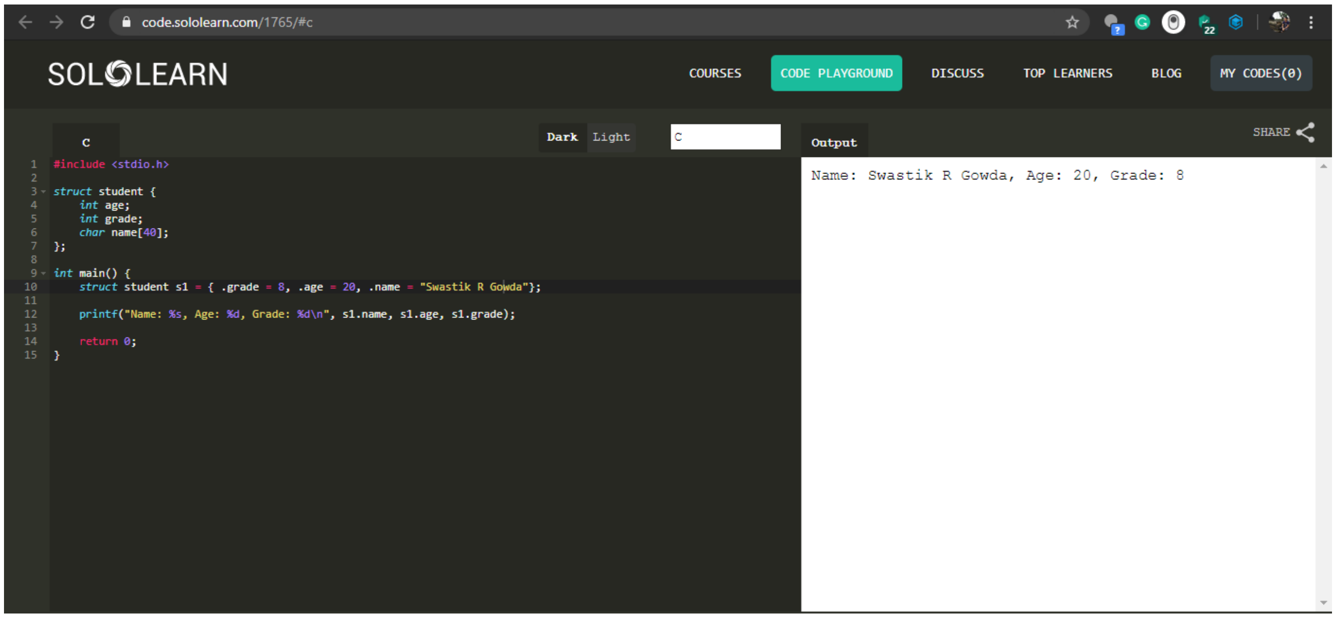
If you want to initialize a structure using curly braces after declaration, you will also need to type cast, as in the statements:

struct student s1;
s1 = (struct student) {19, 9, "John"};

You can use named member initialization when initializing a structure to initialize corresponding members:

struct student s1
= { .grade = 9, .age = 19, .name = "John"};

In the example above, ***.grade*** refers to the grade member of the structure. Similarly, ***.age*** and ***.name*** refer to the age and name members.



```c
#include <stdio.h>

struct student {
    int age;
    int grade;
    char name[40];
};

int main() {
    struct student s1 = { .grade = 8, .age = 20, .name = "Swastik R Gowda"};

    printf("Name: %s, Age: %d, Grade: %d\n", s1.name, s1.age, s1.grade);

    return 0;
}
```

```
Name: Swastik R Gowda, Age: 20, Grade: 8
```

Accessing Structure Members

You access the members of a struct variable by using the. (Dot operator) between the variable name and the member name.

For example, to assign a value to the age member of the s1 struct variable, use a statement like:

s1.age = 19;

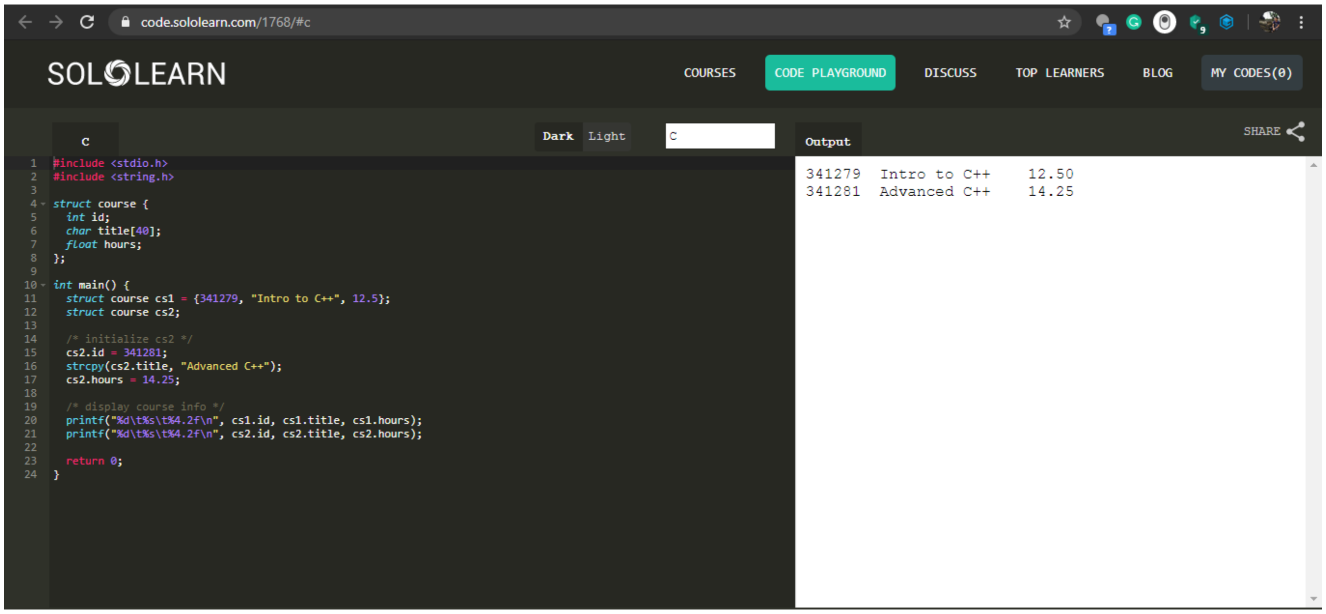
You can also assign one structure to another of the same type:

struct student s1 = {19, 9, "Jason"};
struct student s2;
s2 = s1;

The following code demonstrates using a structure:

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

struct course {
  int id;
  char title[40];
  float hours;
};

int main() {
  struct course cs1 = {341279, "Intro to C++", 12.5};
  struct course cs2;

  /* initialize cs2 */
  cs2.id = 341281;
  strcpy(cs2.title, "Advanced C++");
  cs2.hours = 14.25;

  /* display course info */
  printf("%d\t%s\t%4.2f\n", cs1.id, cs1.title, cs1.hours);
  printf("%d\t%s\t%4.2f\n", cs2.id, cs2.title, cs2.hours);

  return 0;
}
```

String assignment requires ***strcpy()*** from the ***string.h*** library.



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

struct course {
  int id;
  char title[40];
  float hours;
};

int main() {
  struct course cs1 = {341279, "Intro to C++", 12.5};
  struct course cs2;

  /* initialize cs2 */
  cs2.id = 341281;
  strcpy(cs2.title, "Advanced C++");
  cs2.hours = 14.25;

  /* display course info */
  printf("%d\t%s\t%4.2f\n", cs1.id, cs1.title, cs1.hours);
  printf("%d\t%s\t%4.2f\n", cs2.id, cs2.title, cs2.hours);

  return 0;
}
```

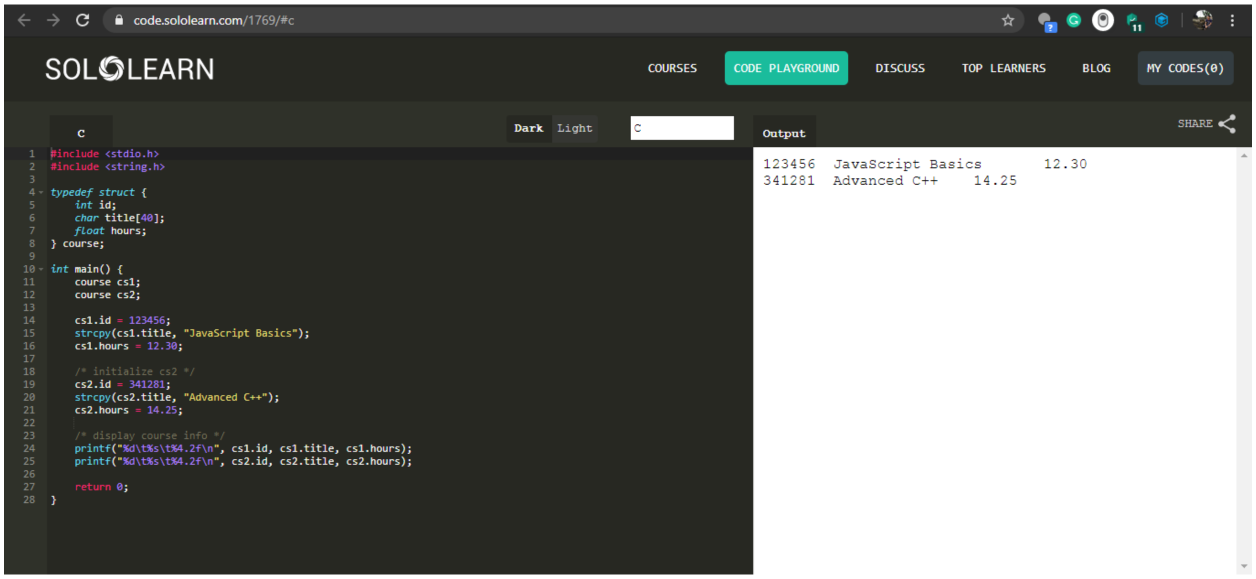
```
341279  Intro to C++    12.50
341281  Advanced C++    14.25
```

Using typedef

- ❖ The typedef keyword creates a type definition that simplifies code and makes a program easier to read.
- ❖ Typedef is commonly used with structures because it eliminates the need to use the keyword struct when declaring variables.

For example:

```
typedef struct
{
        int id;
        char title[40];
        float hours;
} course;
course cs1;
course cs2;
```



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

typedef struct {
    int id;
    char title[40];
    float hours;
} course;

int main() {
    course cs1;
    course cs2;

    cs1.id = 123456;
    strcpy(cs1.title, "JavaScript Basics");
    cs1.hours = 12.30;

    /* initialize cs2 */
    cs2.id = 341281;
    strcpy(cs2.title, "Advanced C++");
    cs2.hours = 14.25;

    /* display course info */
    printf("%d\t%s\t%4.2f\n", cs1.id, cs1.title, cs1.hours);
    printf("%d\t%s\t%4.2f\n", cs2.id, cs2.title, cs2.hours);

    return 0;
}
```

```
123456  JavaScript Basics       12.30
341281  Advanced C++    14.25
```

Unions

- ❖ A union allows to store different data types in the same memory location.
- ❖ It is like a structure because it has members. However, a union variable uses the same memory location for its entire member's and only one member at a time can occupy the memory location.
- ❖ A union declaration uses the keyword union, a union tag, and curly braces { } with a list of members.
- ❖ Union members can be of any data type, including basic types, strings, arrays, pointers, and structures.

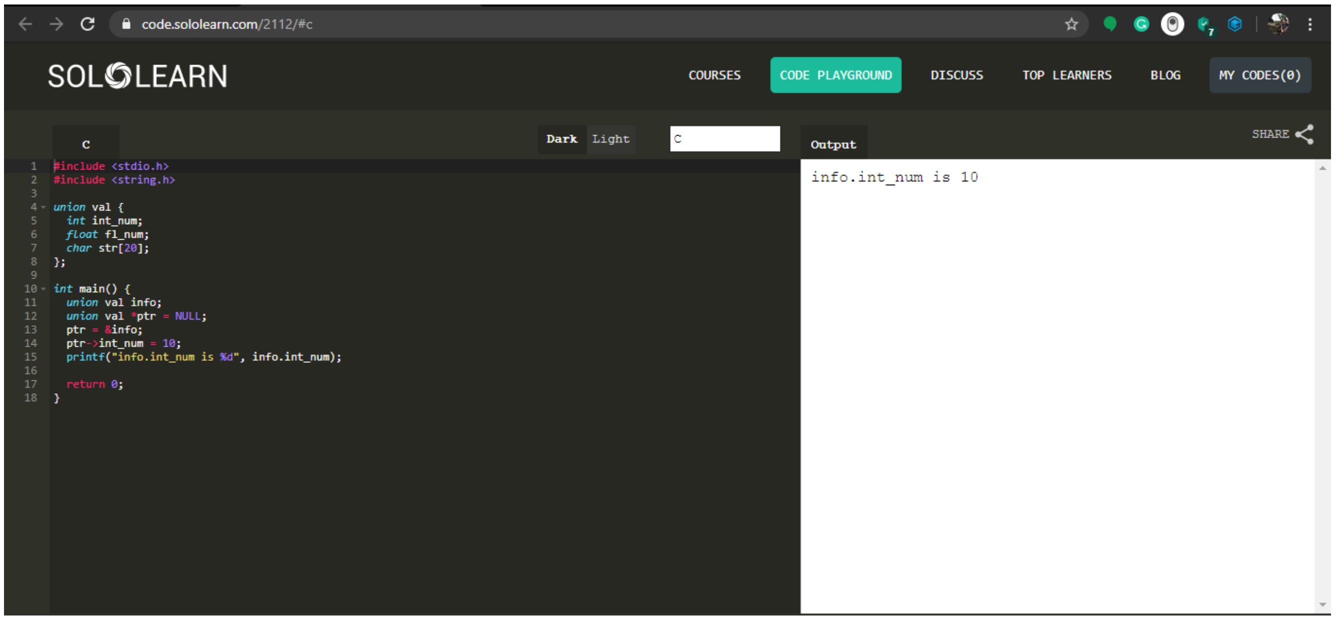
For example:

```
union val
{
        int int_num;
        float fl_num;
        char str[20];
};
```

After declaring a union, you can declare union variables. You can even assign one union to another of the same type:

```
union val u1;
union val u2;
u2 = u1;
```

Unions are used for memory management. The largest member data type is used to determine the size of the memory to share and then all members use this one location. This process also helps limit memory fragmentation.



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

union val {
  int int_num;
  float fl_num;
  char str[20];
};

int main() {
  union val info;
  union val *ptr = NULL;
  ptr = &info;
  ptr->int_num = 10;
  printf("info.int_num is %d", info.int_num);

  return 0;
}
```

```
info.int_num is 10
```

Memory Management

Understanding memory is an important aspect of C programming. When you declare a variable using a basic data type, C automatically allocates space for the variable in an area of memory called the stack.

An int variable, for example, is typically allocated 4 bytes when declared. We know this by using the sizeof operator:

```
int x;
printf("%d", sizeof(x));                                        /* output: 4 */
```

As another example, an array with a specified size is allocated contiguous blocks of memory with each block the size for one element:

```
int arr[10];
printf("%d", sizeof(arr));                                      /* output: 40 */
```

So long as your program explicitly declares a basic data type or an array size, memory is automatically managed. However, you have probably already been wishing to implement a program where the array size is undecided until runtime.

Dynamic memory allocation is the process of allocating and freeing memory as needed. Now you can prompt at runtime for the number of array elements and then create an array with that many elements. Dynamic memory is managed with pointers that point to newly allocated blocks of memory in an area called the heap.

Memory Management Functions

The ***stdlib.h*** library includes memory management functions.
The statement ***#include <stdlib.h>*** at the top of your program gives you access to the following:

- ***malloc(bytes)*** Returns a pointer to a contiguous block of memory that is of size bytes.

- ***calloc(num_items, item_size)*** Returns a pointer to a contiguous block of memory that has ***num_items*** items, each of size ***item_size*** bytes. Typically used for arrays, structures, and other derived data types. The allocated memory is initialized to 0.

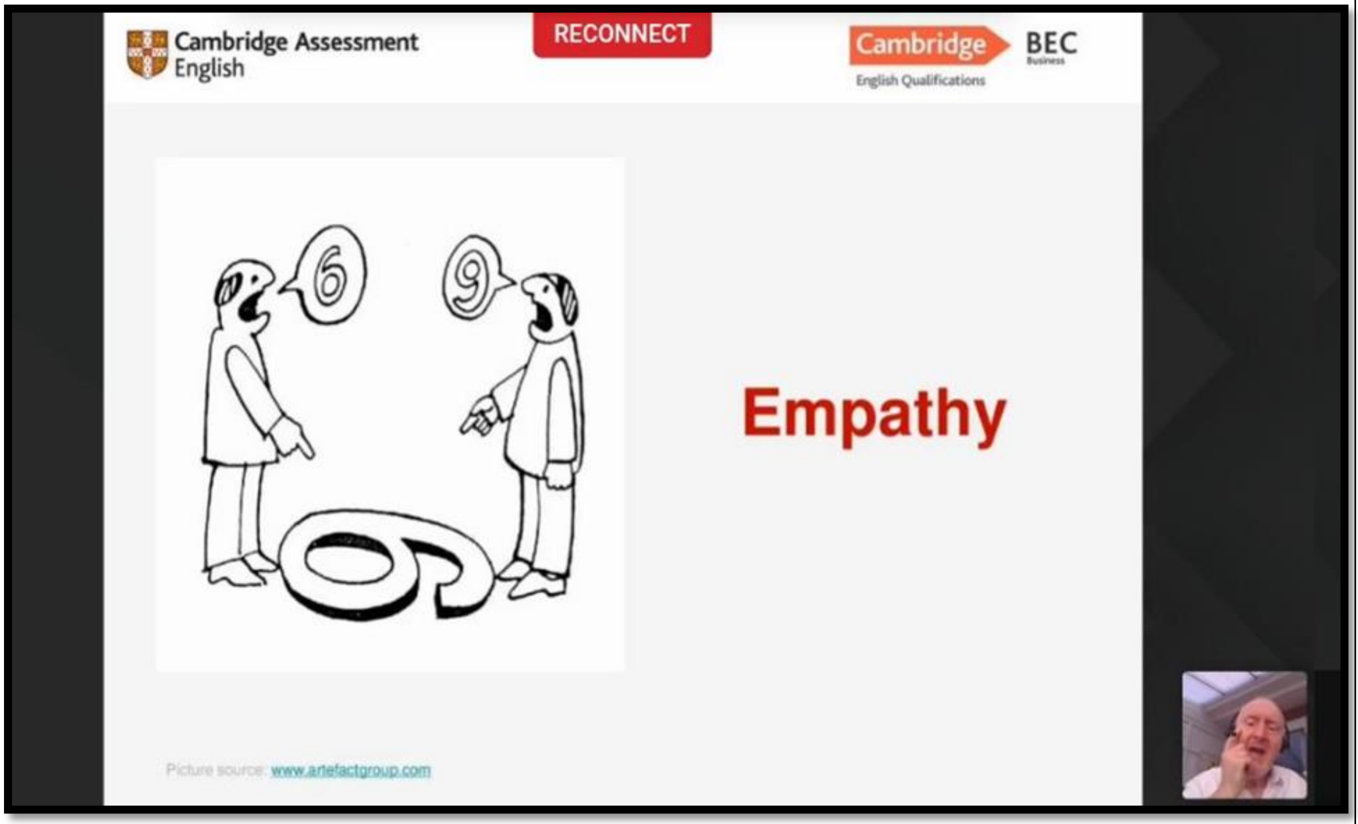
- ***realloc(ptr, bytes)*** Resizes the memory pointed to by ptr to size bytes. The newly allocated memory is not initialized.

- ***free(ptr)*** Releases the block of memory pointed to by ptr.

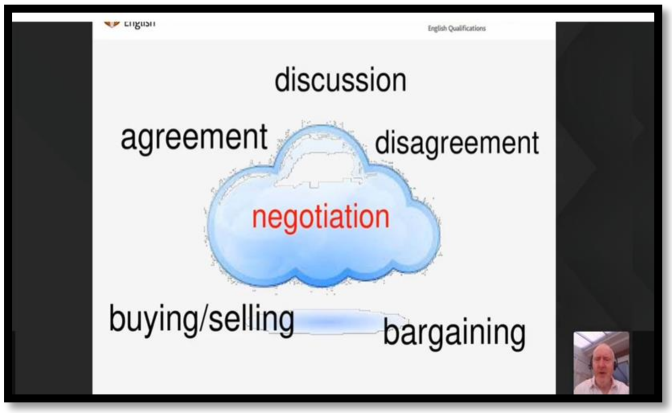
Date:	19-June-2020	Name:	Swastik R Gowda
Course:	Webinar	USN:	4AL17EC091
Topic:	Skills for Work - Negotiation skills	Semester & Section:	6th Sem 'B' Sec

AFTERNOON SESSION DETAILS

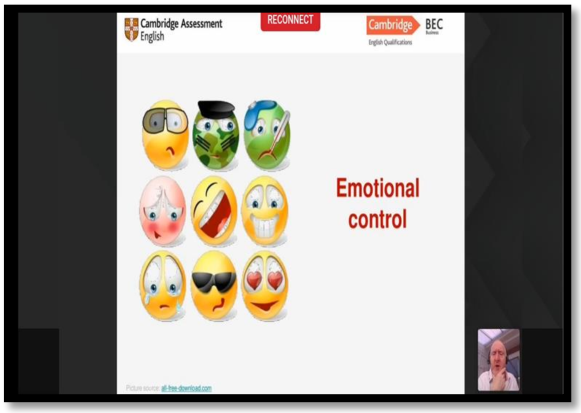
Image of session


Cambridge Assessment English
RECONNECT
Cambridge BEC Business English Qualifications
6
9
Picture source: www.artefactgroup.com

Empathy



discussion
agreement
disagreement
negotiation
buying/selling
bargaining


Cambridge Assessment English
RECONNECT
Cambridge BEC
English Qualifications
Picture source: all-free-download.com

Emotional control