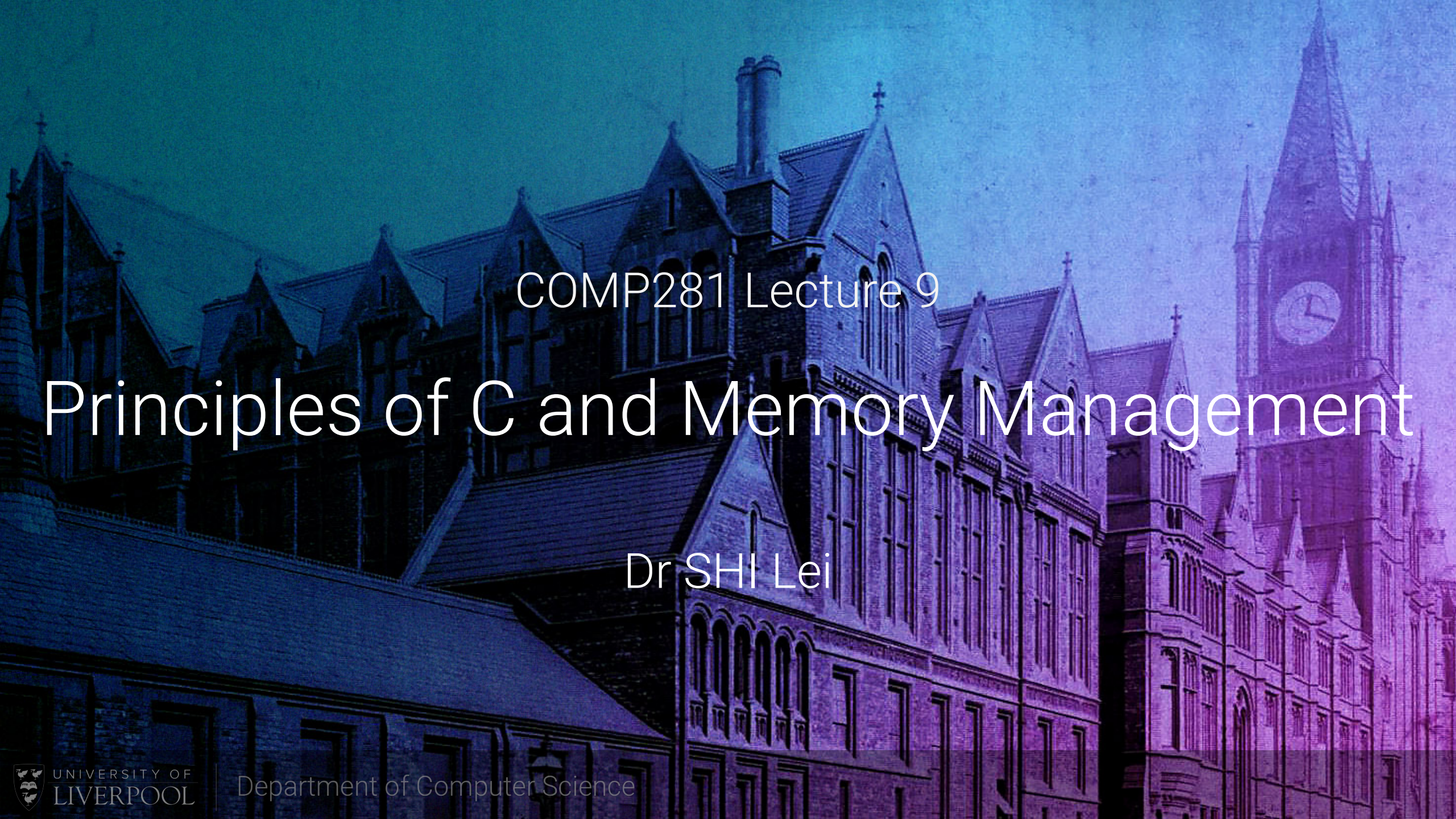

COMP281 Lecture 9

Principles of C and Memory Management

Dr SHI Lei

UNIVERSITY OF LIVERPOOL | Department of Computer Science

Last Lecture

- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

Today

- `struct`
- `typedef`
- `union`
- String
- `Storage classes`

struct

`struct`

- `struct` is a user-defined datatype.
- It combines data of different types.
- It constructs a complex data type which is more meaningful (vs Array).
- It is a way to package primitive data objects into an aggregate data object

Defining a `struct`

Syntax

```
struct [struct_tag]
{
    /* member variable 1 */
    /* member variable 2 */
    /* member variable 3 */
    ...
}[struct_variables];
```

It's optional to provide the `struct` a name, but it's better to provide a name.

Variables of different datatypes like `int`, `float`, `char`, `array` etc

It's optional to specify one or more `struct` variables.

Defining a `struct`

Syntax

```
struct [struct_tag]
{
    /* member variable 1 */
    /* member variable 2 */
    /* member variable 3 */
    ...
}[struct_variables];
```

It's optional to provide the `struct` a name, but it's better to provide a name.

Variables of different datatypes like `int`, `float`, `char`, `array` etc

It's optional to specify one or more `struct` variables.

There must be a semicolon(`;`)!

Defining a `struct`

An example

```
struct Employee
{
    char name[50];
    int age;
    char gender;
    float salary;
};
```

```
struct Employee
{
    char name[50];
    int age;
    char gender;
    float salary;
} employee1, employee2;
```

Accessing `struct` members

- **`struct` members** have no meaning individually without the `struct`.
- To assign a value to a `struct` member, the member name must be linked with the **struct variable** using a dot (`.`) operator.

Example 1

```
#include <stdio.h>
#include <string.h>
struct Lecturer
{
    char name[50];
    char gender;
    int age;
};
int main()
{
    struct Lecturer lec;
    strcpy(lec.name, "Lei Shi");
    lec.gender = 'M';
    printf("Name: %s\n", lec.name);
    printf("Gender: %c\n", lec.gender);
    return 0;
}
```

Accessing `struct` members

- **`struct` members** have no meaning individually without the `struct`.
- To assign a value to a `struct` member, the member name must be linked with the **struct variable** using a dot (`.`) operator.

Example 1

```
#include <stdio.h>
#include <string.h>
struct Lecturer
{
    char name[50];
    char gender;
    int age;
};
int main() {
    struct Lecturer lec;
    strcpy(lec.name, "Lei Shi");
    lec.gender = 'M';
    printf("Name: %s\n", lec.name);
    printf("Gender: %c\n", lec.gender);
    return 0;
}
```

Output

```
Name: Lei Shi
Gender: M
```

Initialising struct

```
struct House
{
    int id;
    float area;
};

struct House h1 = {110, 89.6};
```

or

```
struct House h1;
h1.id = 110;
h1.area = 89.6;
```

Array of struct

Example 2

```
#include <stdio.h>
#include <string.h>
struct Student {
    char name[10];
    int grade;
};
struct Student stu[2];
int main() {
    for(int i = 0; i < 2; i++) {
        printf("\nEnter record of student %d:\n", i+1);
        printf("Enter name:\t");
        scanf("%s", stu[i].name);
        printf("Enter grade:\t");
        scanf("%d", &stu[i].grade);
    }
    printf("\n============================\n");
    printf("Displaying record:\n");
    printf("============================\n");
    for(int i = 0; i < 2; i++) {
        printf("Student %d", i+1);
        printf("\nName is %s", stu[i].name);
        printf("\nGrade is %d\n",stu[i].grade);
        printf("-------------------\n");
    }
    return 0;
}
```

Array of struct

Example 2

```
#include <stdio.h>
#include <string.h>
struct Student {
    char name[10];
    int grade;
};
struct Student stu[2];
int main() {
    for(int i = 0; i < 2; i
        printf("\nEnter reco
        printf("Enter name:\
        scanf("%s", stu[i].n
        printf("Enter grade:
        scanf("%d", &stu[i].
    }
    printf("\n============
    printf("Displaying reco
    printf("================
    for(int i = 0; i < 2; i
        printf("Student %d",
        printf("\nName is %s
        printf("\nGrade is %
        printf("------------
    }
    return 0;
}
```

Output

```
Enter record of student 1:
Enter name: Alan
Enter grade: 1

Enter record of student 2:
Enter name: Blair
Enter grade: 2


============================
Displaying record:
============================
Student 1
Name is Alan
Grade is 1
----------------
Student 2
Name is Blair
Grade is 2
```

Nested struct

```
struct Student {
    char[30] name;
    int age;
    /* here Address is a structure */
    struct Address {
        char[50] locality;
        char[50] city;
        int pincode;
    } addr;
};
```

`struct` as function arguments

- Passing a `struct` as a function argument
- Just like passing any other variable or an array as a function argument

Example 3

```
#include <stdio.h>
struct Student {
    char name[10];
    int grade;
};
void print(struct Student st) {
    printf("\nName: %s\n",st.name);
    printf("\nGrade %d\n",st.grade);
}
```

```
int main() {
    struct Student stu;
    printf("Enter Student record:\n");
    printf("Enter Name:\t");
    scanf("%s", stu.name);
    printf("Enter Grade:\t");
    scanf("%d", &stu.grade);
    print(stu);
    return 0;
}
```

`struct` as function arguments

- Passing a `struct` as a function argument
- Just like passing any other variable or an array as a function argument

Example 3

```
#include <stdio.h>
struct Student {
    char name[10];
    int grade;
};
void print(struct Student st) {
    printf("\nName: %s\n",st.name);
    printf("\nGrade %d\n",st.grade);
}
```

```
int main() {
    struct
    printf
    printf
    scanf(
    printf
    scanf(
    print(
    return
}
```

Output

```
Enter Student record:
Enter Name: Frans
Enter Grade: 1

Name: Frans
Grade 1
```

Pointers to `struct`

Example 4

```
#include <stdio.h>
struct point {
    int x;
    int y;
    double dist;
};
```

```
void init_point(struct point *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    /* syntactic sugar: */
    p->x = p->x = 0;
    p->dist = 0.0;
}
int main() {
    struct point p;
    struct point *pt = &p;
    init_point(pt);
    printf("x=%d\n", (*pt).x);
    return 0;
}
```

Pointers to `struct`

Example 4

```c
#include <stdio.h>
struct point {
    int x;
    int y;
    double dist;
};
```

```c
void init_point(struct point *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    /* syntactic sugar: */
    p->x = p->x = 0;
    p->dist = 0.0;
}
int main() {
    struct point p;
    struct point *pt = &p;
    init_point(pt);
    printf("x=%d\n", (*pt).x);
    return 0;
}
```

Output

```
X=0
```

typedef

`typedef`

- Typing `struct X` all the time is tedious
- `typedef` is to assign alternative names to existing datatypes (type alias).
- It is mostly used with user defined datatypes, when names become complicated to use.

Defining a `typedef`

Syntax

```
typedef <existing_name> <alias_name>
```

For example

```
typedef unsigned long ulong;
```

- Define a term `ulong` for an `unsigned long` datatype.
- Now this `ulong` identifier can be used to define `unsigned long` type variables

```
typedef struct point Point;
```

Defining a `typedef`

Type component of `typedef` can also be a `struct`

```
typedef struct {     /* no name for the struct */
    int x;
    int y;
    double dist;
} Point p1, p2;      /* no "struct" */
```

Note, this is an anonymous `struct`

`typedef` and Pointers

`typedef` can also be used to give an alias name to pointers.

```
/* declaring two pointers? */
int* x, y;

/* declare any number of pointers in a single statement */
typedef int* IntPtr;
IntPtr x, y, z;
```

union

union

- `union` is as special datatype allowing to store different datatypes in the same memory location.
- A `union` can have many members, but only one member can contain a value at any given time.
- `union` provides an efficient way of using the same memory location.

Defining a `union`

Syntax

```
union [union_tag]
{
    /* member variable 1 */
    /* member variable 2 */
    /* member variable 3 */
    ...
}[union_variables];
```

It's optional to provide the `union` a name, but it's better to provide a name.

Variables of different types like `int`, `float`, `char`, `array` etc

It's optional to specify one or more `union` variables.

union vs struct

The syntax to declare/define a `union` is similar to that of a `struct`

```
union Example
{
   int i;
   float f;
   char str[20];
} e;
```

```
struct Example
{
   int i;
   float f;
   char str[20];
} e;
```

union vs struct

But their usages of memory are different

Example 5

```
#include <stdio.h>
#include <string.h>
union Data1 {
   char c;/* 1 bytes */
   int i; /* 4 bytes */
   double d;/* 8 bytes */
};
struct Data2 {
   char c;/* 1 bytes */
   int i; /* 4 bytes */
   double d;/* 8 bytes */
};
int main( ) {
   union Data1 data1;
   struct Data2 data2;
   printf( "Memory size occupied
by data1 : %lu\n", sizeof(data1));
   printf( "Memory size occupied
by data2 : %lu\n", sizeof(data2));
   return 0;
}
```

union vs struct

But their usages of memory are different

Example 5

```
#include <stdio.h>
#include <string.h>
union Data1 {
    char c;/* 1 bytes */
    int i; /* 4 bytes */
    double d;/* 8 bytes */
};
struct Data2 {
    char c;/* 1 bytes */
    int i; /* 4 bytes */
    double d;/* 8 bytes */
};
```

```
int main( ) {
    union Data1 data1;
    struct Data2 data2;
    printf( "Memory size occupied
by data1 : %lu\n", sizeof(data1));
    printf( "Memory size occupied
by data2 : %lu\n", sizeof(data2));
    return 0;
}
```

Output

```
Memory size occupied by data1 : 8
Memory size occupied by data2 : 16
```

Accessing union members

- Similar to `struct`.
- To assign a value to a `union` member, the member name must be linked with the **union variable** using a dot (`.`) operator.

Example 6.1

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   data.i = 66;
   data.f = 99.9;
   strcpy( data.str, "comp281");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

Accessing union members

- Similar to `struct`.
- To assign a value to a `union` member, the member name must be linked with the **union variable** using a dot (`.`) operator.

Example 6.1

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   data.i = 66;
   data.f = 99.9;
   strcpy( data.str, "comp281");
   printf( "data.i : %d\n", data.i);
```

Output

```
data.i : 1886220131
data.f : 29393042243167123688416660684 8.000000
data.str : comp281
```

Accessing union members

- Similar to `struct`.
- To assign a value to a `union` member, the member name must be linked with the **union variable** using a dot (`.`) operator.

Example 6.1

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   data.i = 66;
   data.f = 99.9;
   strcpy( data.str, "comp281");
   printf( "data.i : %d\n", data.i);
```

Output

```
data.i : 1886220131
data.f : 29393042243167123688416660684 8.000000
data.str : comp281
```

Accessing union members

- Similar to `struct`.
- To assign a value to a `union` m
  linked with the **union variable** usi

Example 6.1

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
```

```
int m
   un
   do
   do
   st
   pr
   pr
   pr
   re
}
```

```
Output
data.i : 1886
data.f : 29393
data.str : com
```

The values of `i` and `f` members of `union` got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of `str` member is getting printed as well.

Accessing union members

Example 6.2

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
```

```
int main( ) {
   union Data data;
   data.i = 66;
   printf( "data.i : %d\n", data.i);

   data.f = 99.9;
   printf("data.f : %f\n", data.f);

   strcpy(data.str, "comp281");
   printf("data.str : %s\n", data.str);
   return 0;
}
```

Accessing union members

Example 6.2

```
#include <stdio.h>
#include <string.h>
union Data {
   int i;
   float f;
   char str[20];
};
int main( ) {
   union Data data;
   data.i = 66;
   printf( "data.i : %d\n", data.i);

   data.f = 99.9;
   printf("data.f : %f\n", data.f);

   strcpy(data.str, "comp281");
```

Output

```
data.i : 66
data.f : 99.900002
data.str : comp281
```

string

String

- String is a sequence of `chars` that is treated as a single data item and terminated by null `char '\0'`.
- C does not support string as a datatype
- A string is actually an array of `chars`.

Declaring and initialising string variables

Example 7

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

int main () {
   char msg1[] = {'H','e','l','l','o',',' ',','w','o','r','l','d','!','\0'};
   char msg2[] = "Hello, world!";

   printf("Message 1: %s\n", msg1);
   printf("Message 2: %s\n", msg2);

   printf("Size of Message 1 is %lu.\n", sizeof(msg1) / sizeof(msg1[0]));
   printf("Size of Message 2 is %lu.\n", sizeof(msg2) / sizeof(msg2[0]));

   return 0;
}
```

Declaring and initialising string variables

Example 7

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

int main () {
   char msg1[] = {'H','e','l','l','o',',' ',','w','o','r','l','d','!','\0'};
   char msg2[] = "Hello, world!";

   printf("Message 1: %s\n", msg1);
   printf("Message 2: %s\n", msg2);

   printf("Size of Message 1 is %lu.\n", size
   printf("Size of Message 2 is %lu.\n", size

   return 0;
}
```

Output

```
Message 1: Hello, world!
Message 2: Hello, world!
Size of Message 1 is 14.
Size of Message 2 is 14.
```

- The string “Hello, world!” contains 14 chars including '\0' char which is automatically added by the compiler at the end of the string.
- When initialising a char array by listing all of its characters separately, '\0' char must be provided explicitly.

String functions

Method	Description
`strcat()`	To concatenate(combine) two strings
`strcpy()`	To copy one string into another
`strlen()`	To show length of a string
`strcmp()`	To compare two strings
`strchr()`	To search for the first occurrence of the character

String functions

`strcat()`	To concatenate(combine) two strings

- Syntax

```
char *strcat(char *s1, const char *s2);
```

- Parameters

  s1: A pointer to a string that will be modified. *s2* will be copied to the end of *s1*.

  s2: A pointer to a string that will be appended to the end of *s1*.

- Returns

  A pointer to s1 (where the resulting concatenated string resides).

Example 8

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

int main() {
   /* Define a temporary variable */
   char str[100];

   /* Copy the first string into the variable */
   strcpy(str, "The University ");

   /* Concatenate the following two strings to the end of the first one */
   strcat(str, "of Liverpool ");
   strcat(str, "is established in 1882.");

   /* Display the concatenated strings */
   printf("%s\n", str);

   return 0;
}
```

Example 8

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

int main() {
   /* Define a temporary variable */
   char str[100];

   /* Copy the first string into the variable */
   strcpy(str, "The University ");

   /* Concatenate the following two strings to the end of the first one */
   strcat(str, "of Liverpool ");
   strcat(str, "is established in 1882.");

   /* Display the concatenated strings */
   printf("%s\n", str);

   return 0;
}
```

Output

```
University of Liverpool is established in 1882.
```

String functions

`strcpy()`	To copy one string into another

- Syntax

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

- Parameters

  dest: the pointer to the destination array where the content is to be copied.

  src: the string to be copied.

- Returns

  A pointer to the destination string dest.

Example 9

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

int main() {
   char src[40];
   char dest[100];

   strcpy(src, "This is COMP281.");
   strcpy(dest, src);

   printf("Final copied string : %s.\n", dest);

   return 0;
}
```

Example 9

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

int main() {
   char src[40];
   char dest[100];

   strcpy(src, "This is COMP281.");
   strcpy(dest, src);

   printf("Final copied string : %s.\n", dest);

   return 0;
}
```

Output

```
Final copied string : This is COMP281
```

String functions

`strlen()`	To show length of a string

- Syntax

```
size_t strlen(const char *s);
```

- Parameters

  s: the string whose length is to be found.

- Returns

  The length of the string pointed to by s.

  It does NOT include the null character in the length calculation. `'\0'`

Example 10

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

int main() {
    char str[50];
    int len;

    strcpy(str, "Hello, world!");

    len = strlen(str);

    printf("Length of |%s| is |%d|\n", str, len);

    return 0;
}
```

Example 10

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

int main() {
    char str[50];
    int len;

    strcpy(str, "Hello, world!");

    len = strlen(str);

    printf("Length of |%s| is |%d|\n", str, len);

    return 0;
}
```

Output

```
Length of |Hello, world!| is |13|
```

String functions

`strcmp()`	To compare two strings

- Syntax

```
int strcmp(const char *str1, const char *str2)
```

- Parameters

  str1: the first string to be compared

  str2: the second string to be compared

- Returns

  0 if s1 and s2 are the same;

  Less than 0 if s1<s2;

  Greater than 0 if s1>s2.

Example 11

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[50];
   char str2[50];
   int ret;
   strcpy(str1, "012345");
   strcpy(str2, "O12345");
   ret = strcmp(str1, str2);

   if(ret < 0) {
      printf("str1 is less than str2.\n");
   } else if(ret > 0) {
      printf("str2 is less than str1.\n ");
   } else {
      printf("str1 is equal to str2.\n ");
   }
   return 0;
}
```

Example 11

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[50];
   char str2[50];
   int ret;
   strcpy(str1, "012345");
   strcpy(str2, "O12345");
   ret = strcmp(str1, str2);

   if(ret < 0) {
      printf("str1 is less than str2.\n");
   } else if(ret > 0) {
      printf("str2 is less than str1.\n ");
   } else {
      printf("str1 is equal to str2.\n ");
   }
   return 0;
}
```

Output

```
str1 is less than str2.
```

String functions

`strchr()`	To search for the first occurrence of the character

- Syntax

```
char *strchr(const char *str, int c)
```

- Parameters

  str: the string to be searched in

  c: the character to be searched for

- Returns

  A pointer to the first occurrence of the character c in the string str, or

  NULL if the character is not found.

Example 12

```c
#include <stdio.h>
#include <string.h>
int main() {
   const char str[] = "https://www.liverpool.ac.uk";
   const char ch = '.';

   char *ret;

   ret = strchr(str, ch);

   printf("String after |%c| is - |%s|\n", ch, ret);

   return 0;
}
```

Output

```
String after |.| is - |.liverpool.ac.uk|
```

String functions

```
char *strncat(char *dest, const char *src, size_t n)
int strcoll(const char *str1, const char *str2)
char *strncpy(char *dest, const char *src, size_t n)
char *strerror(int errnum)
char *strpbrk(const char *str1, const char *str2)
char *strrchr(const char *str, int c)
size_t strspn(const char *str1, const char *str2)
char *strtok(char *str, const char *delim)
size_t strxfrm(char *dest, const char *src, size_t n)
```

…

Don’t forget

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

Storage classes

Storage classes

Each variable has a storage class which decides the following things:

- **Scope**: where the value of the variable would be available inside a program.
- **Default initial value**: default initial value if NOT explicitly initialised.
- **Life time**: for how long will that variable exist.

The following storage classes are mostly used:

- **Automatic variables**
- **External variables**
- **Static variables**
- **Register variables**

Automatic variables

Each variable has a storage class which decides the following things:

- **Scope**: local to the function block inside which they are defined.
- **Default initial value**: any random value, i.e. garbage value.
- **Life time**: till the end of the function/method block where the variable is defined.

```
{
    int detail;       /* by default it's an 'auto' */
    /* or */
    auto int details;/* Both are same */
    return 0;
}
```

External (Global) variables

Each variable has a storage class which decides the following things:

- **Scope**: not bound by any function; available everywhere.
- **Default initial value**: 0(zero).
- **Life time**: Till the program doesn't finish its execution.

Global values can be changed by any function in the program!

Example 13

```
#include <stdio.h>
int number; /* global variable */

void fun1() {
    number = 20;
    printf("I am in function fun1. My value is %d.\n", number);
}

void fun2() {
    printf("I am in function fun2. My value is %d.\n", number);
}

int main() {
    number = 10;
    printf("I am in main function. My value is %d\n", number);
    fun1(); /* function calling, discussed in next topic */
    fun2(); /* function calling, discussed in next topic */
    return 0;
}
```

Example 13

```c
#include <stdio.h>
int number; /* global variable */

void fun1() {
    number = 20;
    printf("I am in function fun1. My value is %d.\n", number);
}

void fun2() {
    printf("I am in function fun2. My value is %d.\n", number);
}

int main() {
    number = 10;
    printf("I am in main function. My value is %d\n", number);
    fun1(); /* function calling
    fun2(); /* function calling
    return 0;
}
```

Output

```
I am in main function. My value is 10
I am in function fun1. My value is 20.
I am in function fun2. My value is 20.
```

Keyword extern

The `extern` keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.

file1.c

```
#include <stdio.h>
/* global variable */
int a = 7;

void func() {
    a++;
    printf("%d", a);
    … …
}
```

file2.c

```
#include "file.c";

int main() {
    extern int a;
    func();
    … …
    return 0;
}
```

Problem when extern is not used

```
int main() {
   a = 10; /* Error: cannot find definition of variable 'a' */
   printf("%d", a);
   return 0;
}
```

Example using extern in same file

```
int main() {
   extern int x; /* informs the compiler that it is defined
                    somewhere else */
   x = 10;
   printf("%d", x);
}
int x; //Global variable x
```

Static variables

Tells the compiler to persist/save the variable until the end of program

- **Scope**: local to the block in which the variable is defined.
- **Default initial value**: 0(zero).
- **Life time**: Till the whole program doesn't finish its execution.

Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialised only once and remains into existence till the end of the program.

Example 14

```c
#include <stdio.h>

void test();

int main() {
   test();
   test();
   test();
   return 0;
}
void test() {
   static int a = 0; /* a static variable */
   a = a + 1;
   printf("%d\t",a);
}
```

Output

```
1	2	3
```

Register variables

Tells the compiler to persist/save the variable until the end of program

- **Scope**: local to the function in which it is declared.
- **Default initial value**: Any random value, i.e. garbage value.
- **Life time**: Till the end of function/method block, in which the variable is defined.

Syntax

```
register int number;
```

- Register variables inform the compiler to store the variable in **CPU** register instead of **memory**.
- Register variables have faster accessibility than a normal variable.
- We can never get the address of such variables.

Which storage class should be used and when?

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables:

- Use **static** storage class only when we want the value of the variable to remain same every time we call it using different function calls.
- Use **register** storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.
- Use **external (global)** storage class only for those variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above mentioned storage classes, then we should use the **automatic** storage class.

Summary

Today

- `struct`
- `typedef`
- `union`
- String
- `Storage classes`

Next

- Dynamic Memory Allocation
- Stack and Heap