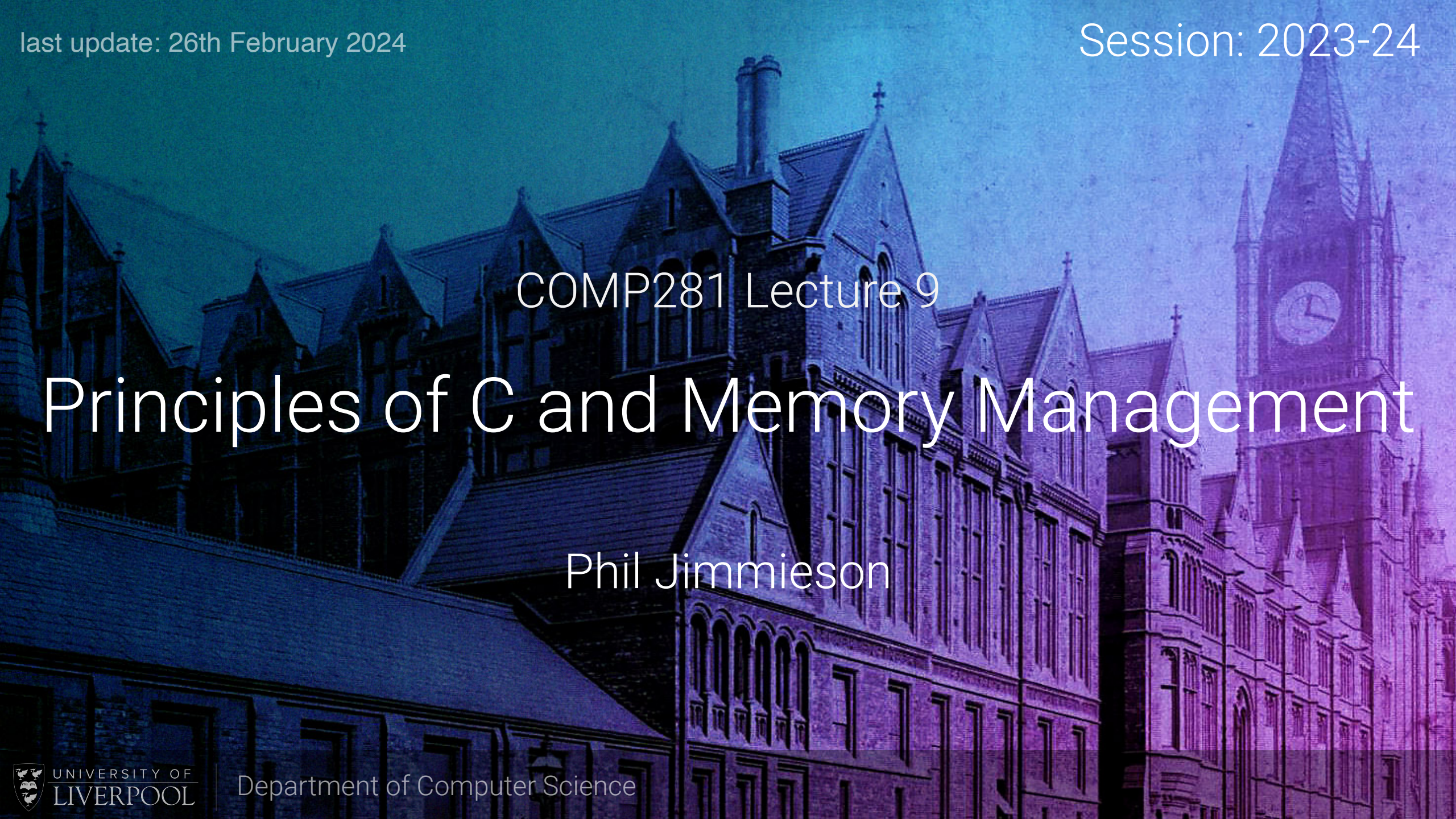
last update: 26th February 2024

Session: 2023-24



COMP281 Lecture 9

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL | Department of Computer Science

Last Lecture

- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions

Today

- `struct`
- `typedef`
- `union`
- String

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 highly recommended.

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 highly recommended.

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 room[20];
    int age;
};
int main(void)
{
    struct Lecturer lec;
    strcpy(lec.name, “Phil Jimmieson");
    lec.room = "Ashton 1.20";
    printf("Name: %s\n", lec.name);
    printf("Room: %c\n", lec.room);
    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 room[20];
    int age;
};
int main(void) {
    struct Lecturer lec;
    strcpy(lec.name, "Phil Jimmieson");
    lec.room = "Ashton 1.20";
    printf("Name: %s\n", lec.name);
    printf("Room: %c\n", lec.room);
    return 0;
}
```

Output

```
Name: Phil Jimmieson
Room: A1.20
```

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(void) {
    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(void) {
    for (int i = 0; i < 2; i
        printf("\nEnter recor
        printf("Enter name:\t
        scanf("%s", stu[i].na
        printf("Enter grade:\
        scanf("%d", &stu[i].g
    }
    printf("\n==============
    printf("Displaying recor
    printf("==============
    for (int i = 0; i < 2; i
        printf("Student %d",
        printf("\nName is %s"
        printf("\nGrade is %d
        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;
        char[9] postcode;
    } 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(void) {
    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(void) {
    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 location {
    int x;
    int y;
    double dist;
};
```

```
void init_point(struct location *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    /* syntactic sugar: */
    p->x = p->y = 0;
    p->dist = 0.0;
}

int main(void)
{
        struct location p;
        struct location *ptr = &p;
        init_point(ptr);
        printf("x=%d,y=%d\n", (*ptr).x, (*ptr).y);
        printf("x=%d,y=%d\n", ptr->x, ptr->y);
        return 0;
}
```

Pointers to `struct`

Example 4

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

```
void init_point(struct location *p) {
    (*p).x = (*p).y = 0;
    (*p).dist = 0.0;
    /* syntactic sugar: */
    p->x = p->y = 0;
    p->dist = 0.0;
}

int main(void)
{
        struct location p;
        struct location *ptr = &p;
        init_point(ptr);
        printf("x=%d,y=%d\n", (*ptr).x, (*ptr).y);
        printf("x=%d,y=%d
        return 0;
}
```

Output

```
X=0,  Y=0
X=0,  Y=0
```

typedef

`typedef`

- Typing `struct X` all the time is tedious
- `typedef` provides the facility 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 ;          /* no "struct" */
```

Note, this is an *anonymous* ***`struct`***

- Now we have a type **`Point`** (don't need to use **`struct Point`**)

Typedef, Struct and Struct pointers

Example 4b

```
typedef struct {/* no name for the struct */
    int x;
    int y;
    double dist;
} Point, *PointPtr;
…
p1.x = 5; p1.y = 6;
PointPtr p = &p1;
printf("%d:%d\n",p->x,p->y);
```

A new custom type: ***Point***

A new type of pointer to the custom type: ***PointPtr***

`typedef` and Pointers

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

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

No - one pointer to Int, and one Int.

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

Yes!

union

`union`

- `union` is a special datatype allowing us to store different datatypes in the same physical 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 is highly recommended.

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(void) {
    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(void) {
   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(void) {
   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(void) {
   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

```c
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main(void) {
    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
   da
   da
   st
   pr
   pr
   pr
   re
```

```
Output
data.i : 1886
data.f : 2939
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. This is the reason that the value of `str` member is printed correctly in this example.

Accessing union members

Example 6.2

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main(void) {
    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(void) {
    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 an additional null `char` `'\0'` placed at the end.
- 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 (void) {
   char msg1[ ] = {'H','e','l','l','o',',' ',','w','o','r','l','d','!','\0'};
   char msg2[ ] = "Hello, world!”; //get compiler to add the ‘\0’

   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 (void) {
   char msg1[ ] = {'H','e','l','l','o',',' ',','w','o','r','l','d','!','\0'};
   char msg2[ ] = "Hello, world!”; //get compiler to add the ‘\0’

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

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

   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(void) {
   /* 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, "was 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, "was established in 1882.");

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

   return 0;
}
```

Output

```
University of Liverpool was 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 char array where the content is to be copied.

  src: the string to be copied.

- Returns

  A pointer to the destination string.

Example 9

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

int main(void) {
   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(void) {
   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()`	Returns the 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(void) {
    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(void) {
    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(void) {
   char str1[50];
   char str2[50];
   int ret;
   strcpy(str1, “012345”);//starts with a zero
   strcpy(str2, “O12345”);//starts with a capital O
   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(void) {
   char str1[50];
   char str2[50];
   int ret;
   strcpy(str1, “012345”);//starts with a zero
   strcpy(str2, “O12345”);//starts with a capital O
   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.
```

- In other programming languages that you have used, strings are first-class types, and can be used in the way that other types are. e.g.

```swift
let user = "Phil"
let myLecturer = "Phil"
if user == myLecturer print(“True")
```
Swift

```
True
```

- But strings in C are not first-class types, they’re char arrays, and this is a cause of some problems (especially for those people who have used other programming languages). Unlike the true first-class types, Strings cannot be operated upon by the standard C operators in the way you might expect.

```c
char user[10] = "Phil";
char myLecturer[10] ="Phil";
if user == myLecturer { fprints("True");}
```
C

Note that you can’t compare two strings directly in C, or, at least, you **can**, but it won’t do what you think. What does the following code do?

```
#include <stdio.h>
int main(void) {
  char str[30] = {"Hello, World"};
  char str2[30] = {"Hello, World"};

  if (str == str2) {
    printf("The same\n");
  } else {
    printf("different\n");
  }
  return 0;
}
```

Note that you can't compare two strings directly in C, or, at least, you **can**, but it won't do what you think. What does the following code do?

```
#include <stdio.h>
int main(void) {
  char str[30] = {"Hello, World"};
  char str2[30] = {"Hello, World"};

  if (str == str2) {
    printf("The same\n");
  } else {
    printf("different\n");
  }
  return 0;
}
```

Output

```
different
```

Note that you can't compare two strings directly in C, or, at least, you **can**, but it won't do what you think. What does the following code do?

```
#include <stdio.h>
int main(void) {
  char str[30] = {"Hello, World"};
  char str2[30] = {"Hello, World"};

  if (str == str2) {
    printf("The same\n");
  } else {
    printf("different\n");
  }
  return 0;
}
```

warning: array comparison always evaluates to false

Output

```
different
```

String functions

`strchr()`	To search for the first occurrence of a 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>
```

Summary

Today

- `struct`
- `typedef`
- `union`
- String

Next

- Storage Classes
- Dynamic Memory Allocation
- Stack and Heap