Computer Systems Principles

C Structures

UMASSCS
SCHOOL OF COMPUTER SCIENCE

Learning Objectives

- To learn and apply C structures
- To understand a little about alignment
- To learn and apply C enums
- To learn and apply C unions
- To understand and apply C typedef

C Structures

- **Essential**
  - For building up interesting data structures
- **Definition**
  - A **C structure** is a collection of one or more variables, typically of different types, grouped together under a single name for convenient handling
  - Kind of like a Java class with public instance variables and no methods

C `struct`

- **Defines a new type**
  - A new kind of data type that the compiler regards as a unit or aggregate of variables/types.
- **Example:**

```
struct Date {
  int day;
  int month;
  int year;
};
```

Structure Properties

- Individual components of a struct type are called **members** (or **fields**).
- Members can be of **different types** (primitive, array, or struct)
- A struct is *named* as a whole while individual members are named using field identifiers
- Complex data structures can be formed by defining *arrays of structs*.

More `struct` Examples

- **Examples:**

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentGrade {
  char name[25];
  char course[9];
  int lab[5];
  int homework[7];
  int exams[2];
};
```

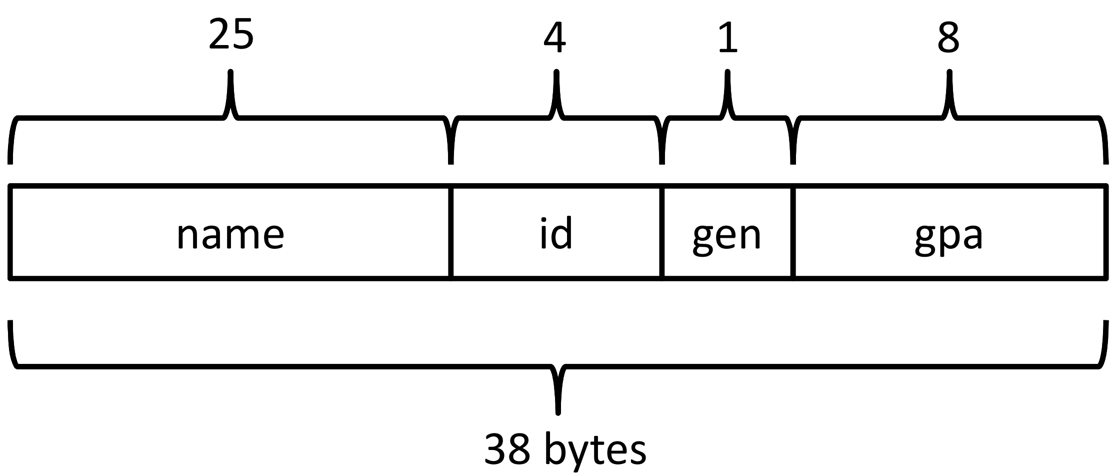
Declaring a `struct` Variable

- Declaration of a variable of struct type:

  <struct type> <identifier list>;

- **Example:**

```
struct StudentRecord student1;
```



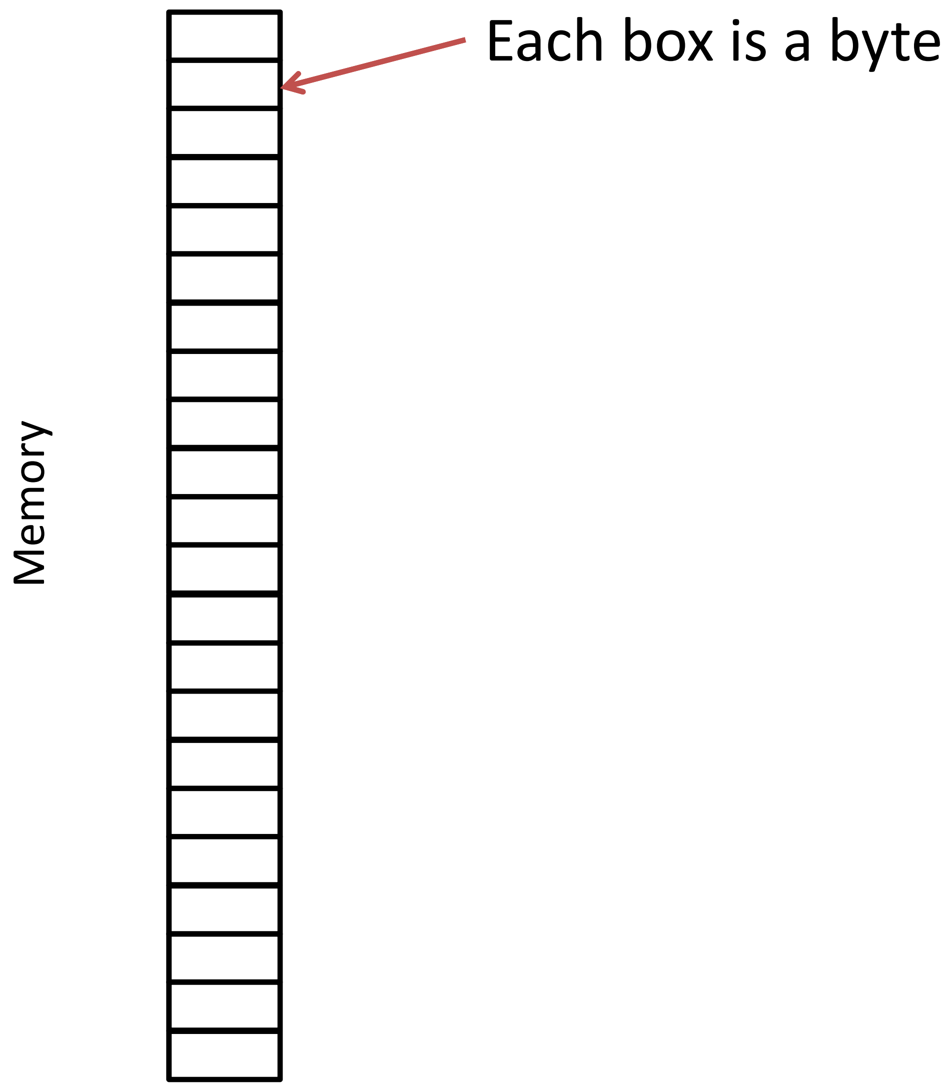
student-01.c example

- **Let us compile this example**
  - What do you noticed about the output of this program as compared with the previous slide?
  - Is the size of this struct the same as we predicted?
  - Why is this or is this not the case?

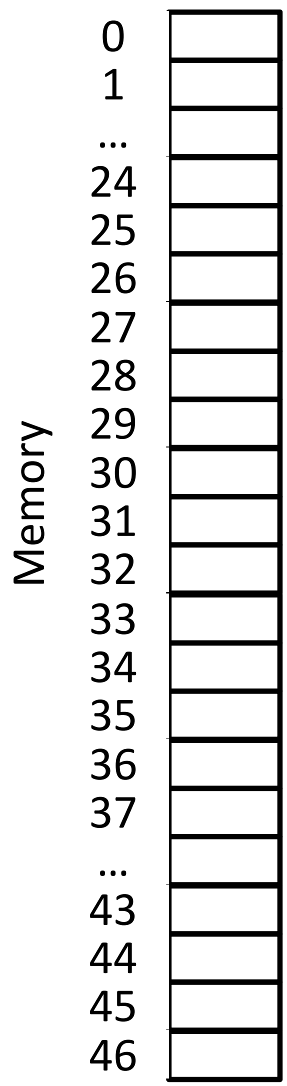
Data Allocation and Alignment

- **Data Allocation**
  - Each variable definition is allocated bytes in memory according the type of that variable
  - e.g., char = 1 byte, int = 4 bytes, double = 8 bytes
  - This is allocated in a special place in memory known as the **stack**.
- **Data Alignment**
  - Machines are more efficient if allocated data is on a **word boundary**.
  - A word is typically 4 bytes

Alignment in Memory



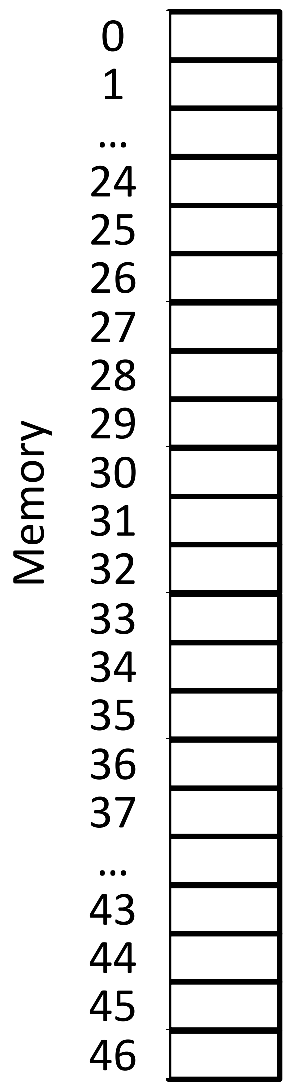
Alignment in Memory



Each box is a byte
and has a location.

Memory is very much like a a
giant character array!

Alignment in Memory

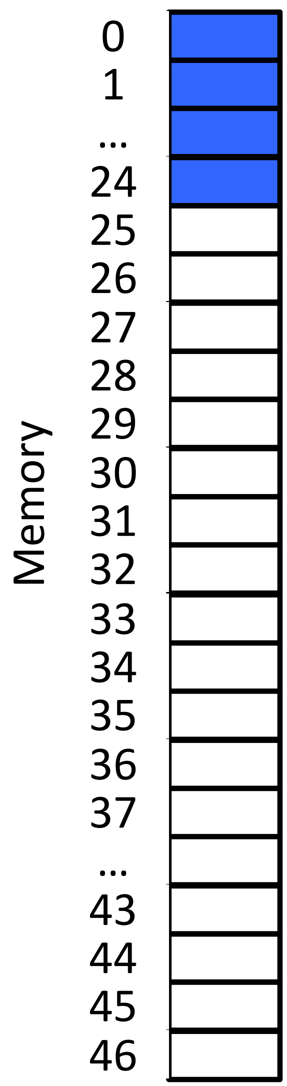


So, how do we allocate the structure definition below?

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

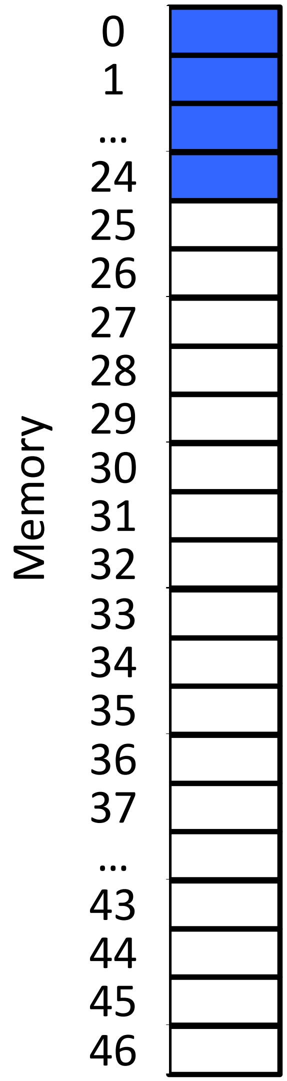


We allocate 25 bytes for the character array **name**. In this example we start from 0, however, we could start from anywhere in memory.

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory



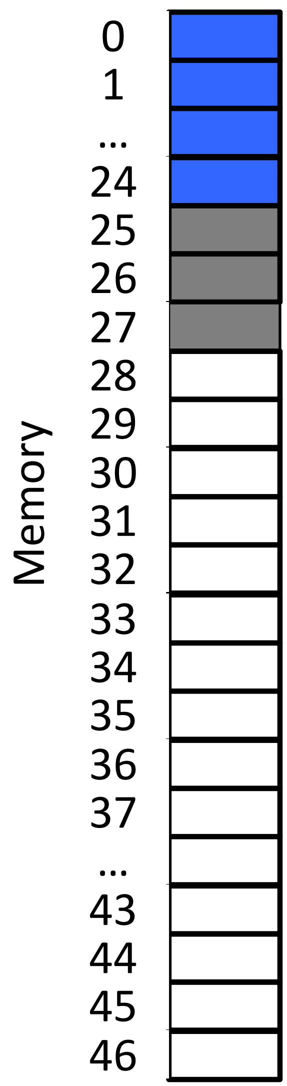
But, machines are typically **more efficient** if data is allocated on the start of 4 byte “word boundaries” (every 4th byte) e.g., 4, 8, 12, 16, 20, 24, …

That is, the starting memory index for allocating the next data type should be (index % 4) = 0

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

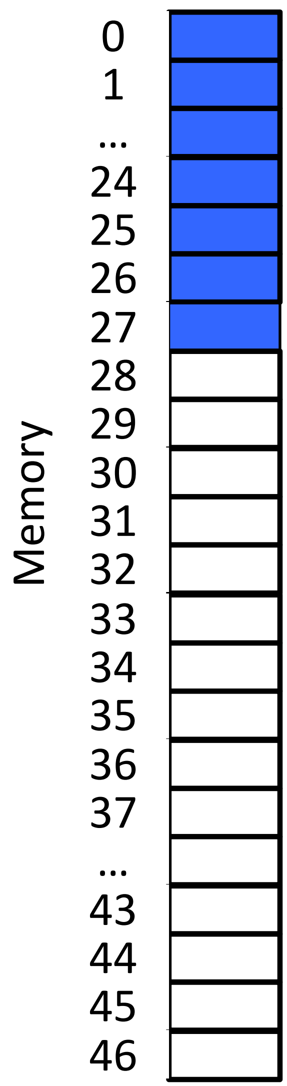


So, the compiler will make the allocation of your structures more efficient by **padding** the bytes so the next allocation will be **4-byte aligned**!

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

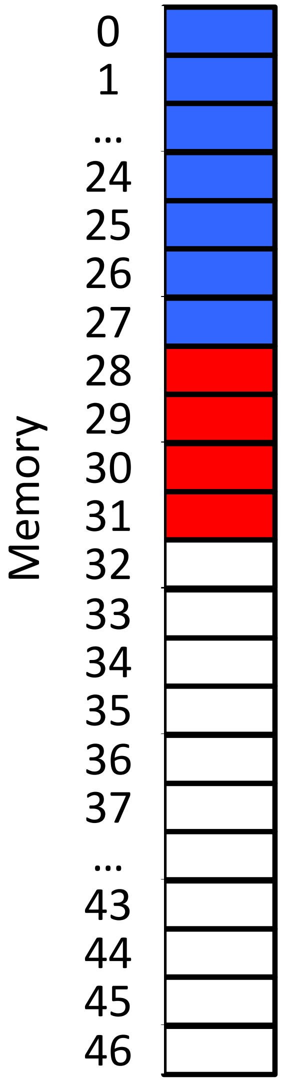


So, the compiler will make the allocation of your structures more efficient by **padding** the bytes so the next allocation will be **4-byte aligned**!

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

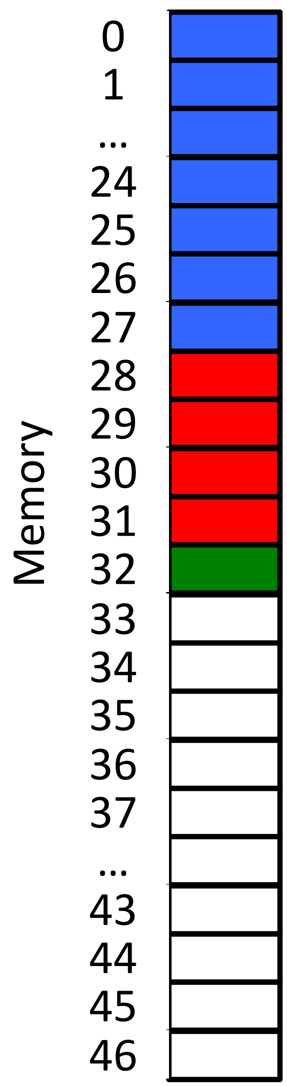


Next, we allocate bytes for the next type – this is a 4-byte integer, so it is already aligned properly.

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

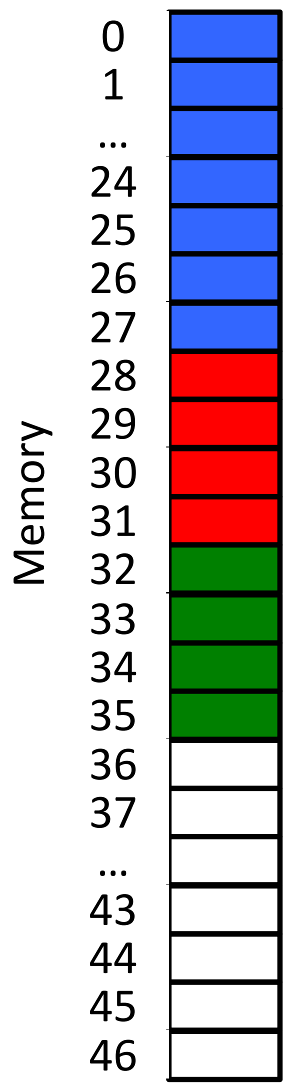


Next is the character. Will the next allocation be aligned properly?

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory

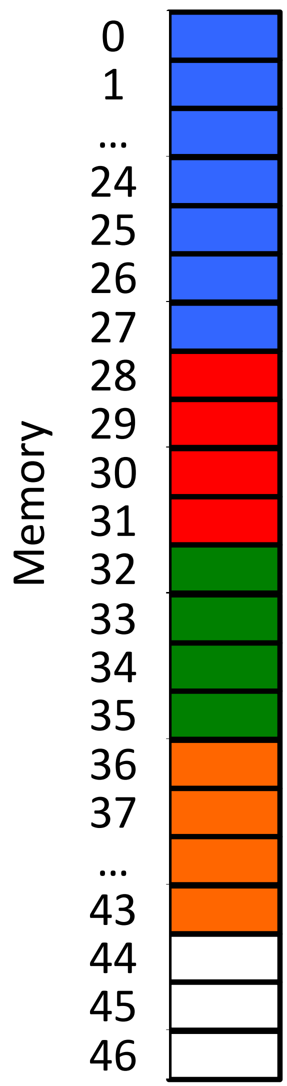


Nope! So the compiler will pad this out with 3 additional bytes

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

Alignment in Memory



Lastly, we allocate the last field in the structure which is an 8-byte double which is 4-byte aligned.

So, instead of 38 bytes – we get 44 bytes!

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};

struct StudentRecord student1;
```

What if…

We change around the fields in a structure?

Let us take a look at this example:

student-01-alt.c

iClicker Question

The size of this struct will be:

A. 38 bytes
B. 40 bytes
C. 44 bytes
D. 48 bytes
E. None of the above

What Happened?

- Previously, we had 44 bytes.
- Now, we have 40 bytes allocated!
- Same fields ?!

```
struct StudentRecord {
  char name[25];
  int  id;
  char gender;
  double gpa;
};
```

```
struct StudentRecord {
  int  id;
  double gpa;
  char name[25];
  char gender;
};
```

Think about this for a moment and write down an explanation.
I will randomly ask for four responses and read them!

Rounding up …

- The new layout uses 38 bytes, packing better than the old one, but …
- The compiler rounds struct sizes up to a multiple of 4 bytes, giving 40 as the length.
- Why?
  - The main reason is that each struct in an array of these structs needs to *start* on a 4 byte boundary, so the individual structs need to be a multiple of 4 bytes in size.

Structure Initialization

- **There are three ways to initialize a struct**
  - Positional initialization
  - Named initialization
  - Copy initialization
  - Initialize individual fields

Positional Initialization

Positional initialization allows you to provide the values for each of the fields based on the position of each structure member:

```
struct StudentRecord student1 = {
  “John Doe”, 1234567, “M”, 3.95
};
```

student-02.c example

- **Let us compile this example**
  - Notice that each value in the structure initializer is exactly the same position as the structure definition.
  - Look at how we use the ‘.’ operator to access the individual fields in a structure. This should be reminiscent of how you access fields in Java.

Named Initialization

Named initialization allows you to provide the values for each of the fields based on the name of each structure member:

```
struct StudentRecord student1 = {
  .id   = 1234567,
  .gpa    = 3.95,
  .gender = ‘M’,
  .name = “Harry Potter”
};
```

student-03.c example

- **Let us compile this example**
  - Notice that each value in the structure initializer can occur in any position.
  - Do not forget to use the ‘.’ prefix for each field name in the initializer!

Copy Initialization

Copy initialization allows you to initialize a structure by assigning an existing structure:

```
struct StudentRecord student1 = {
  .id      = 1234567,
  .gpa     = 3.95,
  .gender  = ‘M’,
  .name = “Harry Potter”
};

struct StudentRecord student2 = student1;
```

student-04.c example

- **Let us compile this example**
  - Notice that we can initialize by using the assignment operator ‘=‘.
  - This will automatically copy the memory from the structure on the right-hand side to the structure on the left-hand side.

Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```
struct StudentRecord student1;

student1.id        = 1234567;
student1.gender    = ‘M’;
student1.gpa       = 3.95;
```

Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```
struct StudentRecord student1;

student1.id       = 1234567;
student1.gender   = 'M';
student1.gpa      = 3.95;
student1.name     = "Harry Potter";
```

What about this one?

student-05.c example

- **Let us compile this example**
  - What problems do we encounter with this example?
  - Why can’t we assign a string to a character array?
    - The string is a char * type
    - Arrays are not modifiable values, that is, you can’t reassign them to “point” to different locations in memory.
    - Huh? This will be more clear when we talk about pointers ☺

Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```
struct StudentRecord student1;

student1.id         = 1234567;
student1.gender     = 'M';
student1.gpa        = 3.95;
student1.name       = "Harry Potter";
```

So, how do we fix this?

strncpy

- **Copying Strings**
  - #include <string.h>
    A library for manipulating C strings

  - To assign a new string value to a C string (e.g., character array) you must use the *strncpy* function to **copy** the bytes into the array.

Field Initialization

Field initialization allows you to initialize a structure by assigning to its fields:

```
struct StudentRecord student1;

student1.id         = 1234567;
student1.gender     = 'M';
student1.gpa        = 3.95;
strncpy(student1.name,"Harry Potter",25);
```

Size of the *destination*

We use the strncpy function!

student-05-fix.c example

- **Let us compile this example**
  – Notice that need to specify the number of bytes.
- **Why do I need to specify the number of bytes?**
  – There is also a *strcpy* function that does not require the number of bytes.
  – but, this function is dangerous because it is possible to copy a larger string into a smaller destination array **overwriting** adjacent memory!
  – This is called **buffer overflow** and can be used to exploit system vulnerabilities!

student-06.c example

- **One last example…**
  - Creating a “constructor” function for creating new structures is a useful pattern in C.
- **Let us compile this example**
  - Notice that need to specify the number of bytes.
- **Some new things:**
  - `strlen` for computing the length of a string
  - Functions in C are “call by value”!
    - But, what about passing arrays to functions?

Activity!

- **strlen(char s[])**
  strncpy(char dest[], char src[], int n)
  - Take a moment to implement these functions!
  - Work with the people around you!
  - Write it down on a piece of paper!