Structures

What are Structures?

- Like arrays, they are ***derived data types***
- Unlike arrays, they group together variables of ***different data types***
- Usually used to store information expressed in different data types that describe a single entity

Structures vs. Arrays

Structures

- Group together variables of different data types
- These variables store information related to a single entity

Arrays

- Group together variables of the same data types
- Arrays store information on the same characteristic of different entities

How to define structures? (1)

- Three ways to define structures

struct keyword

structure tag

(1)

```
struct s_tag
{
   char name[50];
   int age;
} s;
```

Separate variable declarations:

```
struct s_tag s1, s2;
struct s_tag s[100];
```

declares variable s with data type ***struct s_tag***; variable declarations at this point are ***optional***

How to define structures? (2)

(2)

```
typedef struct
{
   char name[50];
   int age;
} s_type;
```

s_type is now the ***type name***, NOT a variable declaration

Separate variable declarations:

```
s_type s1, s2;
s_type s[100];
```

How to define structures? (3)

(2)

```
typedef struct s_tag
{
   char name[50];
   int age;
} s_type;
```

structure tag

type name

Separate variable declarations:

```
s_type s1, s2;
s_type s[100];
```

OR

```
struct s_tag s1, s2;
struct s_tag s[100];
```

How to use structures?

- Structures would be useless unless we could access their ***fields/members***

```
struct s_tag
{
   char name[50];
   int age;
};
```

fields/members of the structure

The dot Operator (.)

- Used to access the fields/members of the array
- Is a simple dot (.)
- Usage:

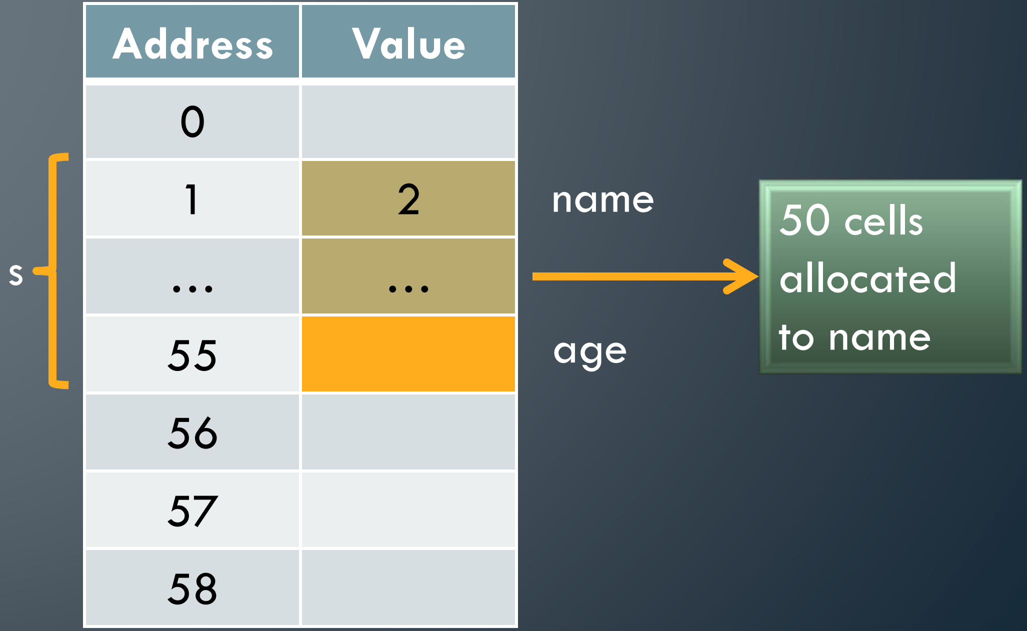
```
    s_type s;

    printf(“Enter your name: ”);
    scanf(“%s”, s.name);
    printf(“Enter your age: ”);
    scanf(“%d”, &(s.age));
```

once the dot operator has been used to access a field, the resulting expression can be used as a normal variable

Memory Allocation of Structure Variables (1)

- Recall: Arrays are allocated to contiguous memory cells
- Structures are allocated by allocating memory cells for each of its fields
  - Memory cells for structure fields are allocated one after another in the order in which they are declared
- Example:

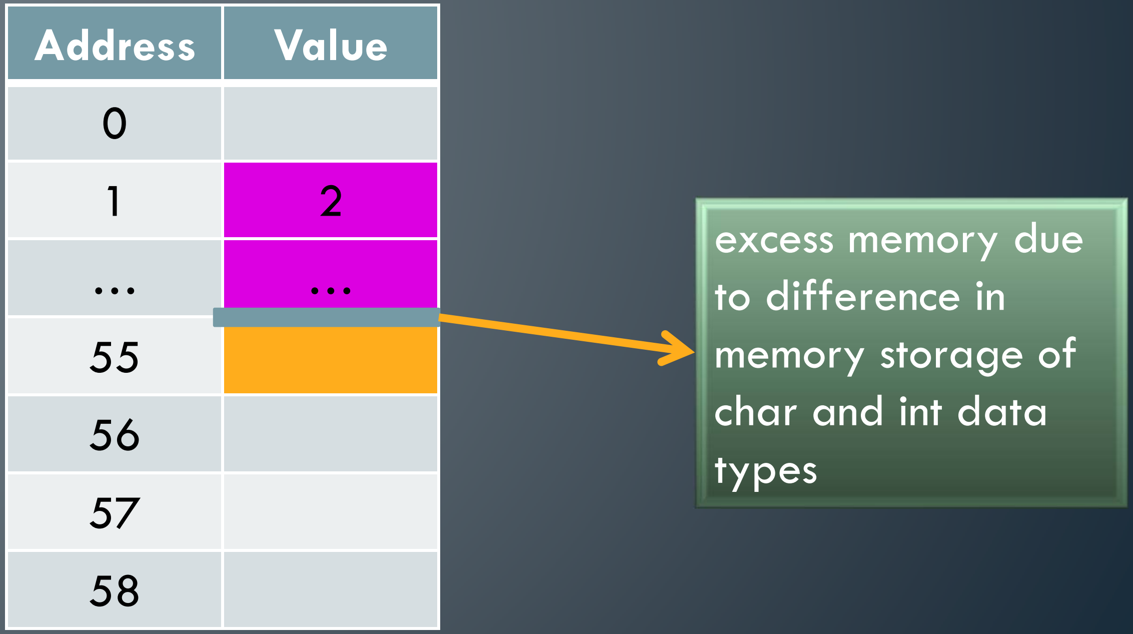
```
struct s_tag
{
  char name[50];
  int age;
} s;
```



Address	Value
0	
1	2
…	…
55	
56	
57	
58	

Memory Allocation of Structure Variables (2)

- However, because structure fields have different types, there are small bits of memory that are unallocated in between fields



Address	Value
0	
1	2
…	…
55	
56	
57	
58	

excess memory due to difference in memory storage of char and int data types

Initialization of Structure Variables

- Similar syntax as with array initialization
- Example:

```
struct s_type s = {“Kei”, 21};
```

  - Initializes the string “Kei” to s.name and the value 21 to s.age

Operations on Structures

- The only operations that can be conducted on structures are…
  - Assignment of one structure variable to another
  - Getting the address of a structure
  - Getting the number of bytes needed for a single structure of a particular type (`sizeof` operator)
- ***NO ARITHMETIC OPERATIONS!***
- ***NO RELATIONAL OPERATIONS!***

The arrow (->) Operator (1)

- When using a pointer to a structure, we still need to use the indirection operator before using the dot operator to access the field
- Example

```
    s_type s, *p;
    p = &s;
    scanf(“%s”, (*s).name);
```

The arrow (->) Operator (2)

- To shorten this expression, we have the arrow operator
- `(*s).name` ↔ `s->name`
- Example

```
    s_type s, *p;
    p = &s;
    scanf(“%s”, s->name);
```