HIGH-LEVEL PROGRAMMING I

Intro to Structures

by Prasanna Ghali

Introduction

- Structure represents *aggregation* or *composition* of data items having *heterogeneous* types
- Recall arrays are also aggregation of data items except data items have *homogeneous* type

Structure definition syntax (1/3)

- Usual way to group stuff together in C is to put in braces:

```
{ stuff ... }
```

- Syntax for structures is similar and likewise easy to remember

Structure definition syntax (2/3)

- Keyword `struct` goes in at front so that compiler can distinguish from code block followed by semicolon at back:

```
struct {
  stuff ...
};
```

- *stuff* can be any data declarations: individual data items, arrays, other structures, pointers, ...

Structure definition syntax (3/3)

- Can follow structure definition by some variable names:

```
struct {
  stuff ...
} mango, pear, apple;
```

- Not useful because there's no way to refer to structure in other parts of program

Definition of structure types

- Instead, optional *structure tag* can be added after keyword `struct` as shorthand to later refer to *`stuff`*:

```
struct fruit_tag {
  stuff ...
};
```

- Somewhere else in your code, shorthand can now be used to define objects of type `struct fruit_tag`:

```
struct fruit_tag mango, pear, apple;
```

General form of struct


```
struct optional_tag {
  type-1 identifier-1;
  type-2 identifier-2;
  ...
  type-N identifier-N;
} optional-variable-definitions;
```

Example declaration (1/2)

- This is a *declaration* - no space is allocated in memory at this point

```
#define NAME_LEN 81

struct Weapon {
   char  name[NAME_LEN];
   int   damage;
   float range;
};
```

Example declaration (2/2)

- Now, *define* a variable of type `struct Weapon`:

```
struct Weapon w;
```

- Data members of variable `w` are given memory storage in declaration order:

	char name[81]	padding	int damage	float range
w	???	???	???	???
	81 bytes	3 bytes	4 bytes	4 bytes

Initialization

- Structures can be initialized much like arrays

```c
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};

struct Weapon w = {"rifle", 10, 5.2f};
```

Initialization – more examples


```c
#define NAME_LEN 81
struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
struct Weapon w1 = {"rifle", 10, 5.2f};
// zero-out range ...
struct Weapon w2 = {"zilch", 5};
// zero-out damage and range ...
struct Weapon w3 = {"zilch"};
// zero-out everything ...
struct Weapon w4 = {""};
struct Weapon w5 = {0};
```

Structure member operator

- Use *structure member operator* . to access structure members

```c
#define NAME_LEN 81
struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
```

```c
struct Weapon w1, w2;

strcpy(w1.name, "Dagger");
w1.damage = 1;
w1.range = 5.1f;

strcpy(w2.name, "Sabre");
w2.damage = 8;
w2.range = 12.5f;
```

Pointer to structure member operator (1/2)

- Use *pointer to structure member operator* -> to access structure members

```c
#define NAME_LEN 81
struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
```

```c
struct Weapon w1, *pw;

strcpy(w1.name, "Dagger");
w1.damage = 1;
w1.range = 5.1f;

pw = &w1;
(*pw).damage *= 2;
pw->range += 1.2f;
```

Pointer to structure member operator (2/2)

- Structure member `.` operator has higher precedence than indirection `*` operator
  - `*pw.damage` equivalent to `*(pw.damage)`
  - `(*pw).damage` corrects this
- *Pointer to structure member operator* (or *arrow operator*) is shorthand for `(*ptr).`
  - Syntactic sugercoating: `pw->damage`

Precedence and associativity

- Structure member operator has higher precedence than most other operators

Level 1

Operator	Meaning	Associativity
++ --	Postfix increment/decrement	
( )	Function call	
[ ]	Array subscripting	L-R
.	Structure member operator	
->	Pointer to structure member operator aka Right arrow selection operator	

Structures and operator =

- Unlike arrays, structures can be assigned to each other

```
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
```

```
struct Weapon w1, w2;

strcpy(w1.name, "Dagger");
w1.damage = 1;
w1.range = 5.1f;

w2 = w1;
/*
now objects w2 and w1 have
same bit pattern in memory
*/
```

Structures and other operators

- Structure object can be operand to only these operators:
  - Structure member operator .
  - Assignment operator =
  - Address-of operator &

Structures and functions

- Structures can be passed to functions and returned from functions just like any object
  - Like any other object, structures are passed by value

Passing struct objects to functions (1/2)


```
struct Vector {
  double x, y;
};
```

```
struct Vector add_vectors(struct Vector v0,
                          struct Vector v1) {
  struct Vector res;
  res.x = v0.x + v1.x;
  res.y = v0.y + v1.y;
  return res;
}

struct Vector start = {10.1, 20.2};
struct Vector end   = {30.2, 40.3};
struct Vector v = add_vectors(start, end);
```

Passing struct objects to functions (2/2)


```
double dot_product(struct Vector v0,
                   struct Vector v1) {
  return v0.x*v1.x + v0.y*v1.y;
}

struct Vector start = {10.1, 20.2};
struct Vector end   = {30.2, 40.3};
double d = dot_product(start, end);
```

Returning structure object from functions (1/2)


```
struct Vector {
  double x;
  double y;
};
```

```
struct Vector new_vector(double x, double y) {
  struct Vector v;
  v.x = x;
  v.y = y;
  return v;
}

struct Vector v0;
v0 = new_vector(1.3, 2.5);
```

Returning structure object from functions (2/2)


```
struct Weapon
set_weapon(char const *n, int d, float r) {
   struct Weapon w;
   strcpy(w.name, n);
   w.damage = d;
   w.range  = r;

   return w;
}

struct Weapon w1 = {"rifle", 10, 5.2f};
w1 = set_weapon("sabre", 20, 10.4f);
```

Structures and functions

- Structures are passed and returned by value to and from functions
  - Passing structure parameters and returning can be expensive operation
  - Different than arrays in that entire structure argument is copied to parameter while only address of first array element is copied
- In practice, pointers to structure objects are more common when structure objects are to be passed to functions

Pointers to structure objects and functions


```
void
set_weapon(struct Weapon *pw, char const *n,
           int d, float r) {
  strcpy(pw->name, n);
  pw->damage = d;
  pw->range  = r;
}

struct Weapon w1 = {"rifle", 10, 5.2f};
set_weapon(&w1, "sabre", 20, 10.4f);
```

Nested structures (1/2)

- Structures can contain any data type, including other structures

```
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
```

```
struct Armor {
  char name[NAME_LEN];
  int protection;
};
```

```
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};

struct Player hero;
```

Nested structures (2/2)

- C/C++ enforce one important restriction on nested structures: A structure of a given type cannot have, as a member, a structure variable of same type

```
// illegal declaration ...
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
  struct Player hero;
};
```

Initialization of nested structure members


```c
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};
```

```c
struct Player hero = {
  "snake",                // name
  {"fang", 80, 2.5f}, // weapon
  {"scale", 2},           // armor
  100
}; // Order is important!!!
```

Accessing elements of nested structures


```
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};
```

```
struct Player hero;

strcpy(hero.name, "snake");

strcpy(hero.weapon.name, "fang");
hero.weapon.damage = 80;
hero.weapon.range = 2.5f;

strcpy(hero.armor.name, "scale");
hero.armor.protection = 2;

hero.health = 100;
```

Data alignment

- For scalar data types, compilers assign addresses that are divisible by size of data type in bytes
  - Variables of type `int` are assigned storage at addresses divisible by 4 i.e., these addresses have least significant 2 bits cleared to 0
  - Variables of type `double` are assigned storage at addresses divisible by 8
- Compiler must therefore *pad* structures so that each structure element is naturally aligned

Sizes

- What is result of evaluation of `sizeof(struct Weapon)`, `sizeof(struct Armor)`, and `sizeof(struct Player)`?

88 bytes

92 bytes

268 bytes

```c
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
```

```c
struct Armor {
  char name[NAME_LEN];
  int protection;
};
```

```c
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};

struct Player hero;
```

Data padding: struct Player

- Memory layout of hero can only be determined by looking at individual data members of struct Player

```
struct Player {
  char name[NAME_LEN+1];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};
struct Player hero;
```

Data alignment: `struct Weapon` (1/2)

- Compilers assign storage for variables of scalar data types at addresses that are multiples of data type's size in bytes
- To ensure each structure element is naturally aligned, compilers *must* align structure objects based on *largest* member data type

Data alignment: struct Weapon (2/2)

Objects of type struct Weapon are given storage at addresses divisible by 4 since they contain int and float data members

```
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};
struct Weapon w;
```

	char name[81]	padding	int damage	float range
w	???	???	???	???
	81 bytes	3 bytes	4 bytes	4 bytes

Data alignment: struct Armor

Objects of type `struct Armor` are given storage at addresses divisible by 4

```c
#define NAME_LEN 81

struct Armor {
  char name[NAME_LEN];
  int protection;
};
struct Armor a;
```

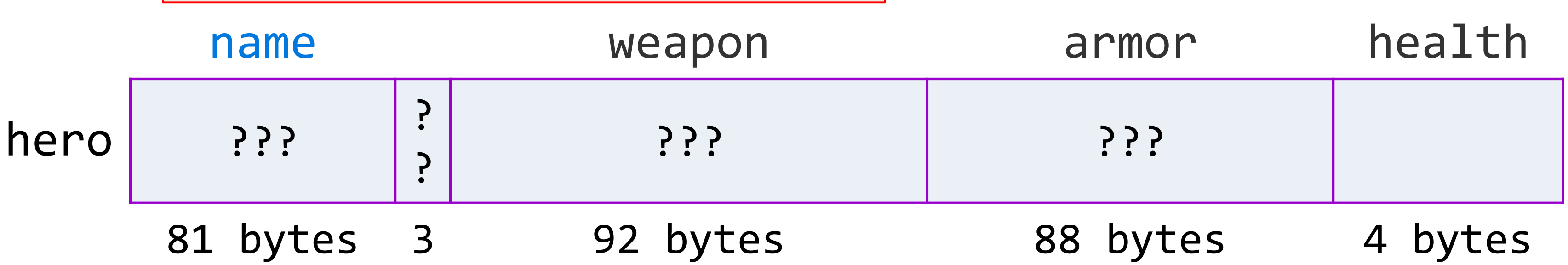
a

char name[81]	padding	int protection
???	???	???
81 bytes	3 bytes	4 bytes

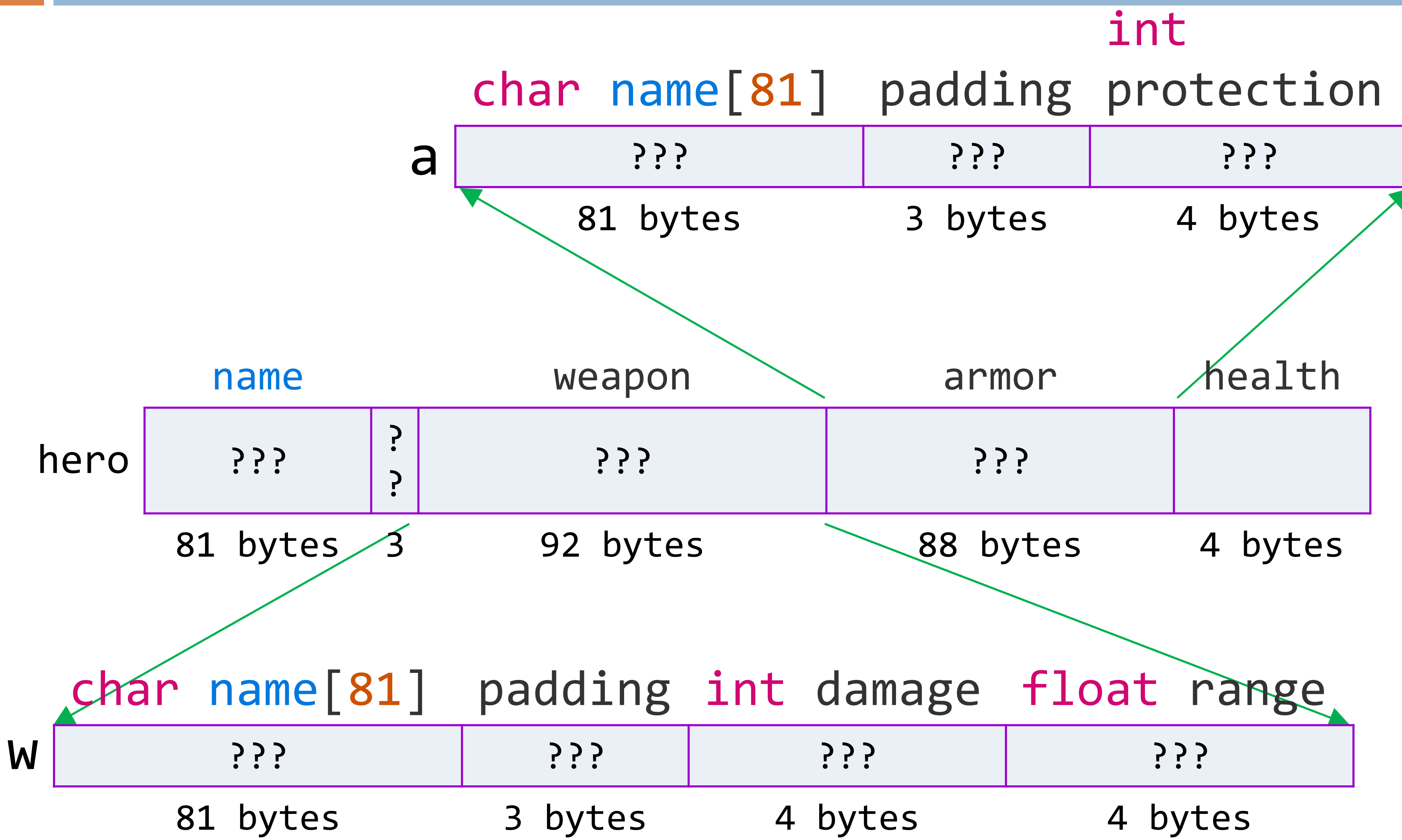
Data padding: struct Player (1/2)

```c
#define NAME_LEN 81

struct Weapon {
  char  name[NAME_LEN];
  int   damage;
  float range;
};

struct Armor {
  char name[NAME_LEN];
  int protection;
};
```

```c
struct Player {
  char name[NAME_LEN];
  struct Weapon weapon;
  struct Armor  armor;
  int health;
};

struct Player hero;
```

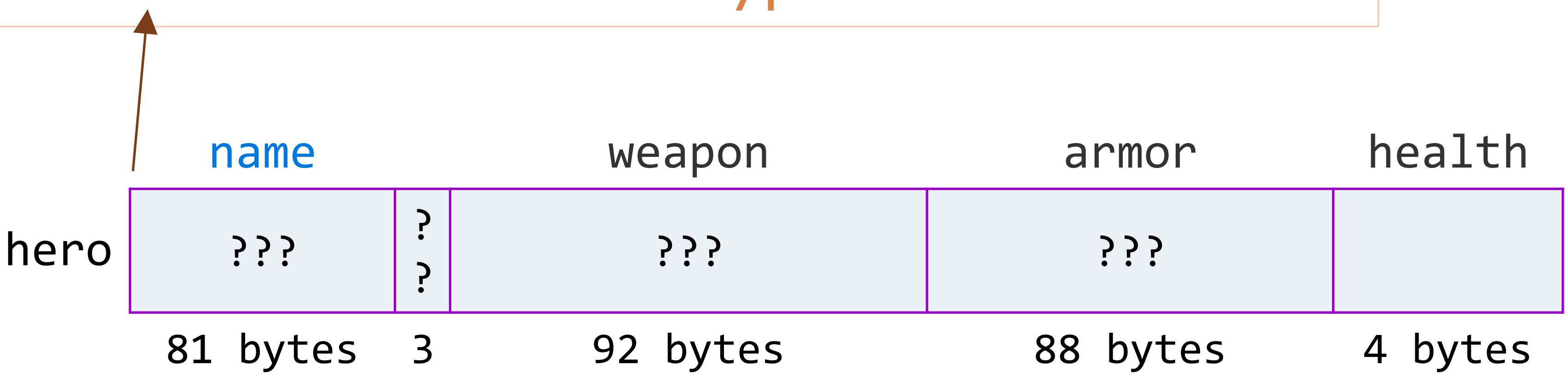


Data padding: struct Player (2/2)



Data alignment: `struct Player`

Objects of type `struct Player` are given storage at addresses divisible by 4 since it contains data members of type `int` and `float`

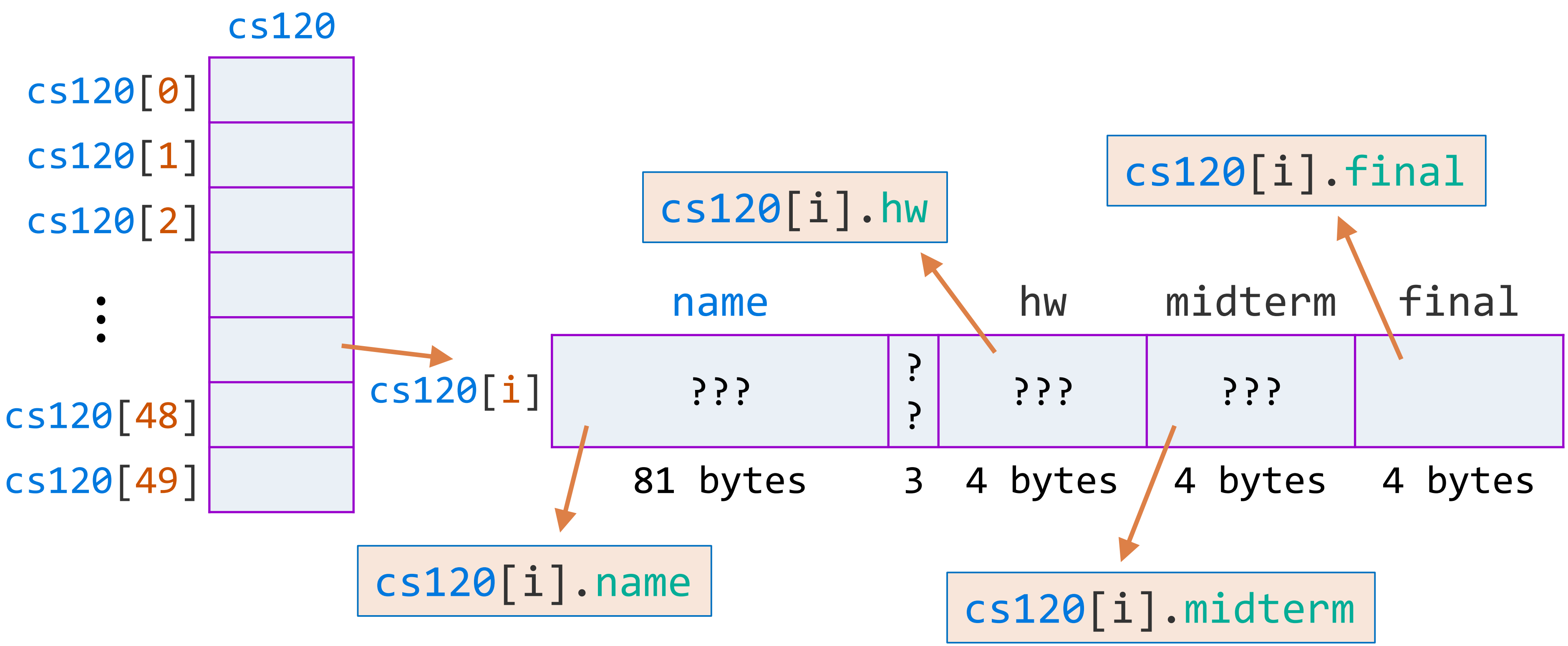


Array of structures

- Just as you can have structures with arrays, you can have arrays of structures

```
struct Student {
  char  name[81];
  float hw;
  int   midterm;
  int   final;
};

struct Student cs120[50];
```

Array of structures



Syntax review (1/8)

- Declare a structure called `struct TIME`

```c
/*
This declaration is in a header file
and doesn't trigger memory allocation!!!
*/
struct TIME {
  int hours;
  int minutes;
  int seconds;
};
```

Syntax review (2/8)

- Create two variables of type `struct TIME`

```
/*
These definitions are in a source file
that includes the header file containing
the declaration of struct TIME
Memory allocation takes place now!!!
*/
struct TIME now, later;
```

Syntax review (3/8)

- This is not very useful …

```c
/*
In one step, declare struct TIME
and define storage for variables
*/
struct TIME {
  int hours;
  int minutes;
  int seconds;
} now, later;
```

Syntax review (4/8)

- Leaving off *tag* creates anonymous structure – this is not useful at all …

```
// No name given to this struct
struct {
  int hours;
  int minutes;
  int seconds;
} now, later;

// Can't define more objects later ...
```

Syntax review (5/8)

- Using `typedef` – *tag* not required

```c
// Declare type TIME in header file
typedef struct {
   int hours;
   int minutes;
   int seconds;
} TIME;
```

Syntax review (6/8)

- Using typedef – not recommended because you're hiding struct keyword

```
// Declare type TIME in header file
typedef struct time_tag {
  int hours;
  int minutes;
  int seconds;
} TIME;
```

Syntax review (7/8)

- Create two variables of type `TIME`

```
/*
These definitions are in a source file
that includes the header file containing
the declaration of type TIME
Memory allocation takes place now!!!
*/
TIME now, later;
```

Syntax review (8/8)

- Using `typedef` – can also declare new name for pointer type

```
// in header file
typedef struct {
    int hours;
    int minutes;
    int seconds;
} TIME;

typedef TIME* TIMEPTR;
```