NWEN 241
Derived & User Defined Types

Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

Background

- Basic data types
  - `int` : integer ✓
  - `char` : character ✓
  - `float` : floating point number ✓
  - `double` : double-precision floating point number ✓
- Derived data types
  - Arrays ✓
  - Strings ✓
  - Structures and Unions
- User defined data types
  - New “types” including *enumeration* types

Background

- Derived types
  - Arrays – all elements must be of the same data type
  - Strings – array of characters with null \0 character at end
- What if you need a collection / group of information consisting of different data types?
  - E.g. student record that comprises name (last, first, middle and preferred), student ID, course, type, etc.
  - Use a composite ***struct****ure* or record that is made up different basic/derived data types;
  - Use a composite ***union*** if different types do not exist at the same time;
  - Use enumeration ***enum*** to define list of constants.

Enumeration

- Enumeration is a user-defined data type. It is defined using the keyword **`enum`** and the syntax is:

```
enum tag_name {name_0, …, name_n} ;
```

- The **`tag_name`** is not used directly. The names in the braces are symbolic constants that take on integer values from **zero** through **n**. As an example, the statement:

```
enum colors { red, yellow, green } ;
```

- creates three constants. **`red`** is assigned the value 0, **`yellow`** is assigned 1 and **`green`** is assigned 2.

enum example

```
/* This program uses enumerated data types to access
   the elements of an array */

#include <stdio.h>

int main( ) {
   int August[5][7] = {{0,0,1,2,3,4,5},
                   {6,7,8,9,10,11,12},
                   {13,14,15,16,17,18,19},
                   {20,21,22,23,24,25,26},
                   {27,28,29,30,31,0,0}};
   enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
   enum week {week_one, week_two, week_three, week_four,
                   week_five};

   printf ("Monday the third week of August "
    "is August %d\n", August[week_three][Mon]);
 }
```

Structures

- A ***`struct`*** is a derived data type composed of members that are each fundamental or derived data types.

- A single ***`struct`*** would store the data for one object. An array of ***`struct`***s would store the data for several objects.

- A ***`struct`*** can be defined in several ways as illustrated in the following examples:

Declaring structure types

- Syntax of the structure type:

```
struct struct_type {
    type1 id1;
    type2 id2;
    …
};
```

- E.g.,

```
struct student_info { // named struct
    char name [20];
    int student_id;
    int age;
}; // does not reserve any space
```

Using structures

- Declaring a variable **`current_student`**

```
struct student_info current_student;
```

- Above statement reserves space for:
  - 20 character array,
  - integer to store student ID, and
  - integer to store age.
- Declaring array of structures to store information of enrolled students in a class

```
struct student_info nwen241class[250];
```

- Reserves space for 250 element array of records (structs) for students enrolled in NWEN241.

Creating new user defined types

- Instead of saying **`struct student_info`** every time we declare a variable, we can define it as a new data type, e.g.

```
typedef struct { // unamed struct
 char name [20];
 int student_id;
 int age;
} StudentInfo;
```

- This makes **`StudentInfo`** a new user-defined type, and you can declare a variable as follows:

```
StudentInfo current_student;
```

New struct and data type

- If student_info has been previously defined, then we can create a new data type using typedef :

```
typedef struct student_info StudentInfo;
```

- Or, we can also do this:

```
typedef struct student_info {
    char name [20];
    int student_id;
    int age;
} StudentInfo;
```

Accessing and manipulating structs

- We can reference a component of a structure by the **direct component selection operator**, which is a **period**, e.g.

```
strcpy(student1.name, "John Smith");
student1.age = 18;
printf("%s is in age %d\n", student1.name,
   student1.age);
```

- The **direct component selection operator** has the highest priority in the operator precedence.
- The copy of an entire structure can be easily done by the assignment operator.

```
student1 = student2;
```

Example – struct and typedef (1)

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

int main() {

   typedef struct student_info {
        char name[20];
        int student_id;
        int age;
   } StudentInfo;

   StudentInfo current_student; // declare new variable using
                                // new type StudentInfo
   struct student_info new_student; // declare using struct
                                    // format
   // do stuff - see next slide
}
```

Example – struct and typedef (2)

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

int main() {
   // declarations in previous slide
   …
   // create new student record
   strcpy(new_student.name , "John Smith");
   new_student.student_id = 300300300;
   new_student.age = 22;

   current_student = new_student;

   printf("Student name : %s\n", current_student.name);
   printf("Student ID   : %.9d\n", current_student.student_id);
   printf("Student Age  : %d\n", current_student.age);

}
```

`struct` as function input parameter (1)

- Suppose there is a structure defined as follows.

```
typedef struct {
   char name[20];
   double diameter;
   int moons;
   double orbit_time,
             rotation_time;
} planet_t;
```

`struct` as function input parameter (2)

- When a structure variable is passed as an input argument to a function, all its component values are copied into the local structure variable.

```c
/*
 * Displays with labels all components of a planet_t structure
 */
void
print_planet(planet_t pl) /* input - one planet structure */
{
      printf("%s\n", pl.name);
      printf("  Equatorial diameter: %.0f km\n", pl.diameter);
      printf("  Number of moons: %d\n", pl.moons);
      printf("  Time to complete one orbit of the sun: %.2f years\n",
             pl.orbit_time);
      printf("  Time to complete one rotation on axis: %.4f hours\n",
             pl.rotation_time);
}
```

Source: Hanly and Koffman, *Problem Solving and Program Design in C*, Pearson, 2006.

`struct` as function input/output parm (1)

- If we define a variable as follows to store data to be read in:

```
planet_t current_planet;
```

- For the following function, we call it by passing the parameter by reference:

```
scan_planet(&current_planet);
```

where the input argument is also used to store the result.

struct as function input/output parm (2)

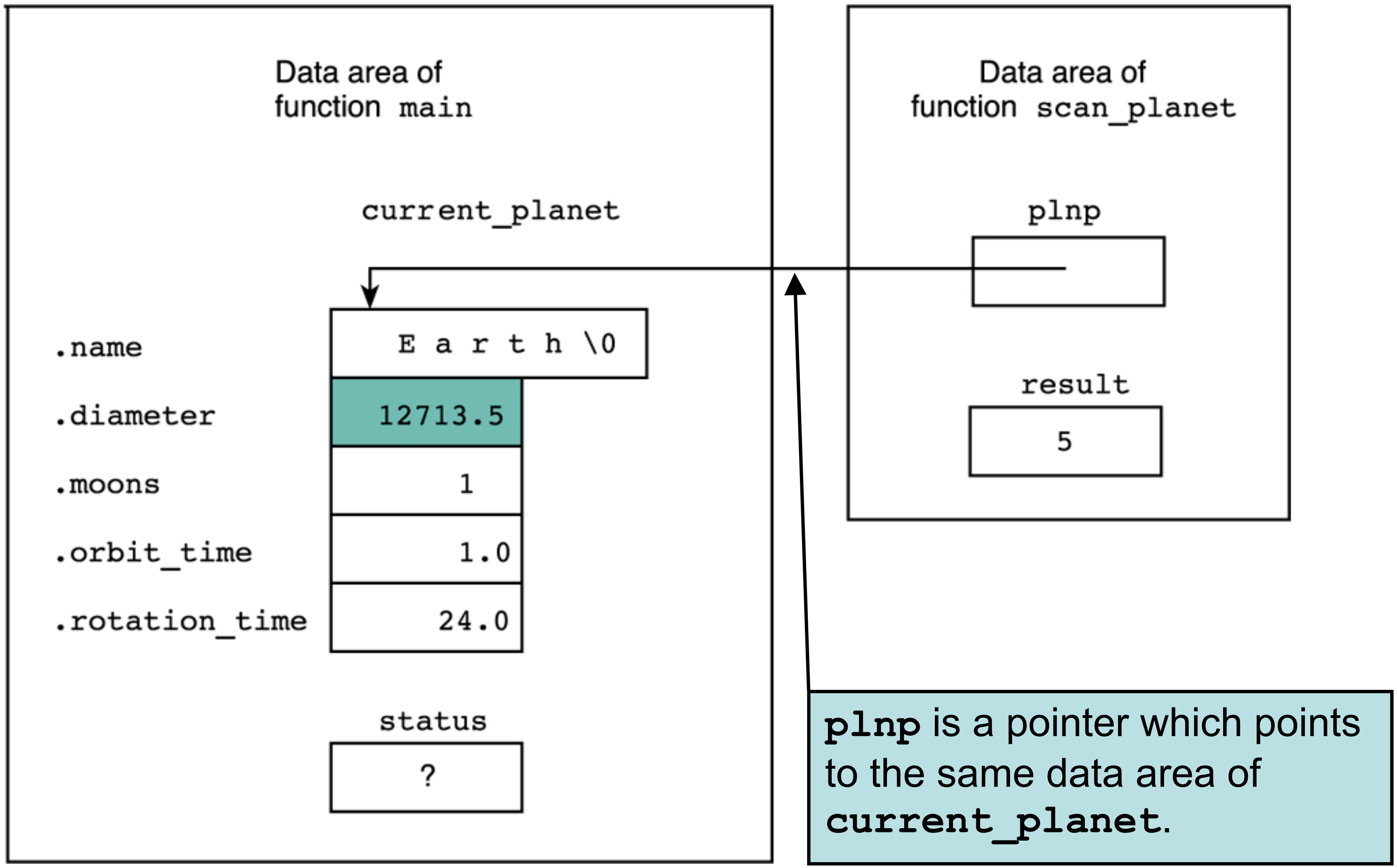
```
int
scan_planet(planet_t *plnp) /* output - address of planet_t structure
                                        to fill                          */
{
      int result;

      result = scanf("%s%lf%d%lf%lf",  (*plnp).name,
                                       &(*plnp).diameter,
                                       &(*plnp).moons,
                                       &(*plnp).orbit_time,
                                       &(*plnp).rotation_time);

      if (result == 5)
            result = 1;
      else if (result != EOF)
            result = 0;

      return (result);
}
```

Why no **&** ?

"***plnp**" is parenthesized because **&** operator has higher precedence.

Source: Hanly and Koffman, *Problem Solving and Program Design in C*, Pearson, 2006.

Data Areas of function call



Source: Hanly and Koffman, *Problem Solving and Program Design in C*, Pearson, 2006.

Indirect referencing steps

- `&(*plnp).diameter` is evaluated as shown in the following:

Reference	Type	Value
`plnp`	`planet_t*`	Address of structure that refers to `current_planet`
`*plnp`	`planet_t`	Real structure of `current_planet`
`(*plnp).diameter`	`double`	12713.5
`&(*plnp).diameter`	`double *`	Address of diameter of `current_planet` structure

- In the above example, we use direct component selection operator: period, e.g.,

```
&(*plnp).diameter
```

- C also provides **indirect component selection operator** : **->** , e.g.

```
&plnp->diameter
```

is the same as

```
&(*plnp).diameter
```

Function returning a `struct` result type

- **`struct`** variable can also be used as a return value of a function

```
/*
 * Computes a new time represented as a time_t structure
 * and based on time of day and elapsed seconds.
 */
time_t
new_time(time_t time_of_day,    /* input - time to be
                                   updated                      */
         int    elapsed_secs)   /* input - seconds since last update   */
{
      int new_hr, new_min, new_sec;

      new_sec = time_of_day.second + elapsed_secs;
      time_of_day.second = new_sec % 60;
      new_min = time_of_day.minute + new_sec / 60;
      time_of_day.minute = new_min % 60;
      new_hr = time_of_day.hour + new_min / 60;
      time_of_day.hour = new_hr % 24;

      return (time_of_day);
}
```

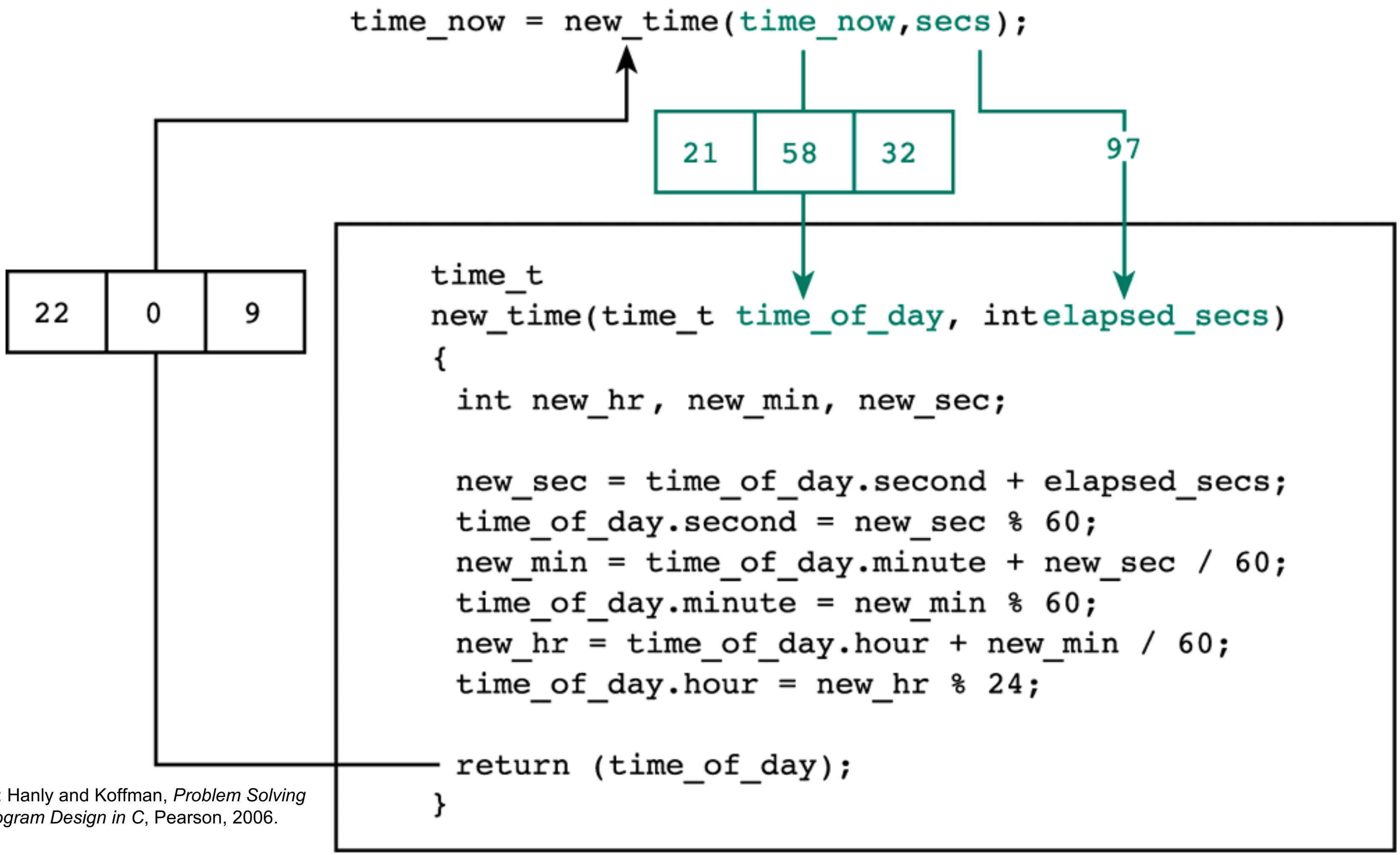
Use direct component selector

Return the **`struct`** value

Source: Hanly and Koffman, *Problem Solving and Program Design in C*, Pearson, 2006.

Function returning a `struct` result type e.g.

- Suppose the current time is 21:58:32, and the elapsed time is 97 seconds.



Source: Hanly and Koffman, *Problem Solving and Program Design in C*, Pearson, 2006.

Array of Structures (1)

- An array of structures can be defined as follows:

```
typedef struct {
   int student_id;
   double gpa;
} student_t;

student_t student_list[50];

student_list[3].student_id = 300922023;
student_list[3].gpa = 8.0;
```

Array of Structures (2)

- Can be simply manipulated as arrays of simple data types

	.student_id	.gpa
student_list[0]	300981683	6.5
student_list[1]	300961592	5.1
student_list[2]	300182652	7.3
student_list[3]	300922023	8.0
…	…	…
student_list[49]	300139414	9.0

student_list[0].gpa

student_list[3].student_id

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

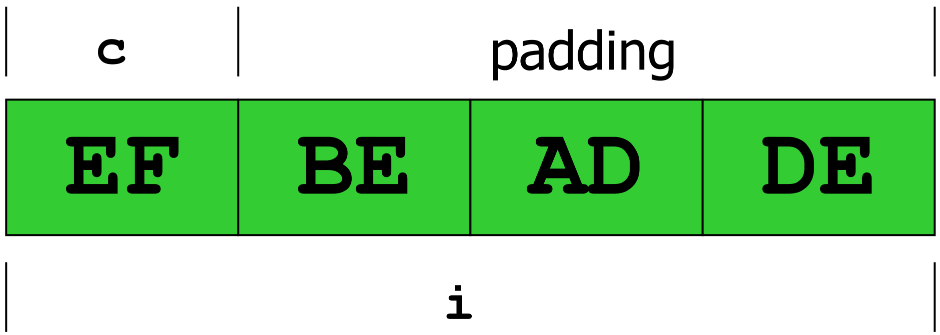
```
union space {
    int i;
    float f;
    char c[4];
};
```

- `sizeof(union space) =`

  max (`sizeof(i), sizeof(f), sizeof(c))`

`union` example

```
union AnElt {
    int   i;
    char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```



`union` doesn't know what it contains…

- How should your program keep track whether **`elt1`**, **`elt2`** hold an **`int`** or a **`char`**?

```
union AnElt {
   int   i;
   char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;

if (elt1 currently has a char)
  …
```

- Basic answer: Another variable holds that info

Tagged `unions`

- *Tag* every value with its case
- Pair the type info together with the union – implicit in other programming languages like Java.

```
enum Union_Tag { IS_INT, IS_CHAR };
struct TaggedUnion {
   enum Union_Tag  tag;
   union {
      int   i;
      char  c;
   } data;
};
```

`enum` must be external to **`struct`**, so constants are globally visible.

`struct` field must be named.