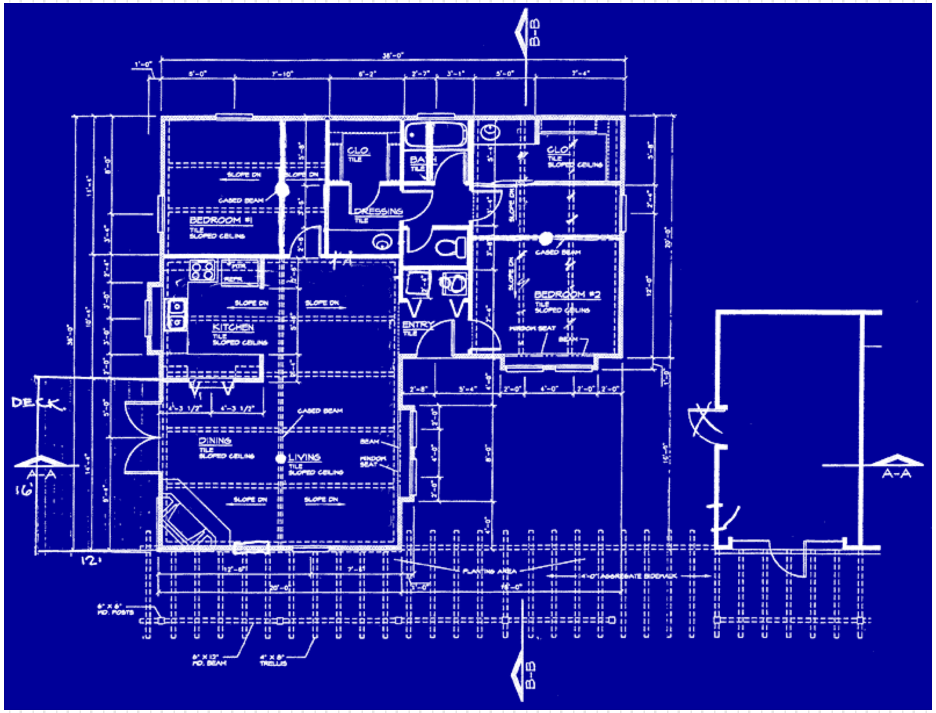
Programming in C

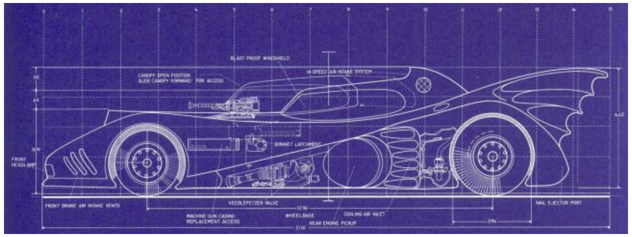
Chapter 8

Structures



Structures

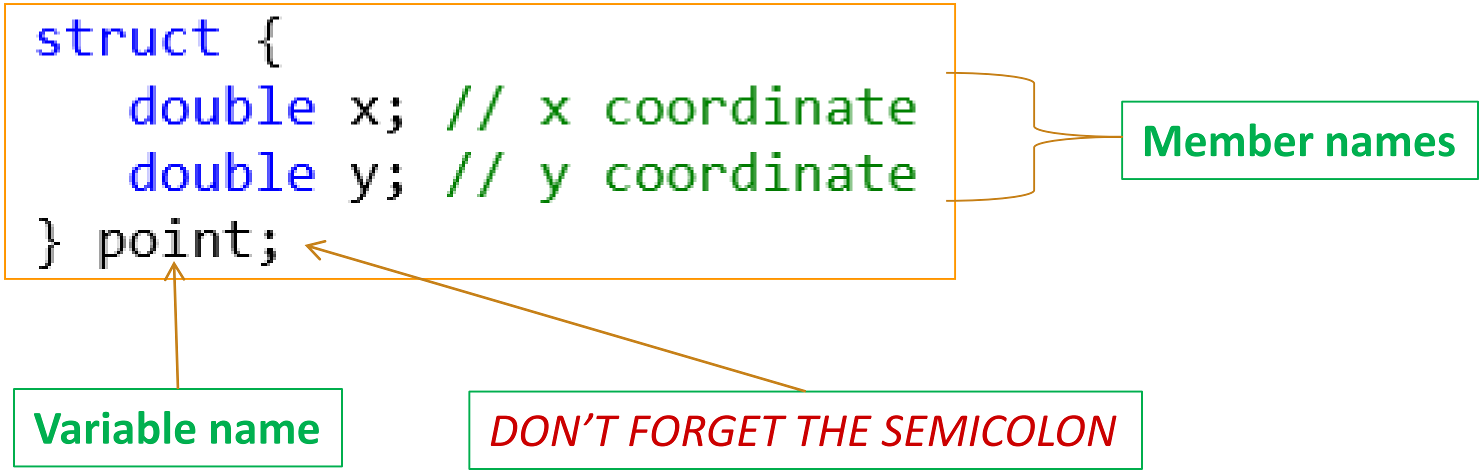
- A structure can be used to define a new data type that combines different types into a single (compound) data type
  - Definition is similar to a template or blueprint
  - Composed of members of previously defined types



- Structures must defined before use
- C has three different methods to define a structure
  - variable structures
  - tagged structures
  - type-defined structures

1) Struct variable

- A variable structure definition defines a struct variable



```
struct {
    double x; // x coordinate
    double y; // y coordinate
} point;
```

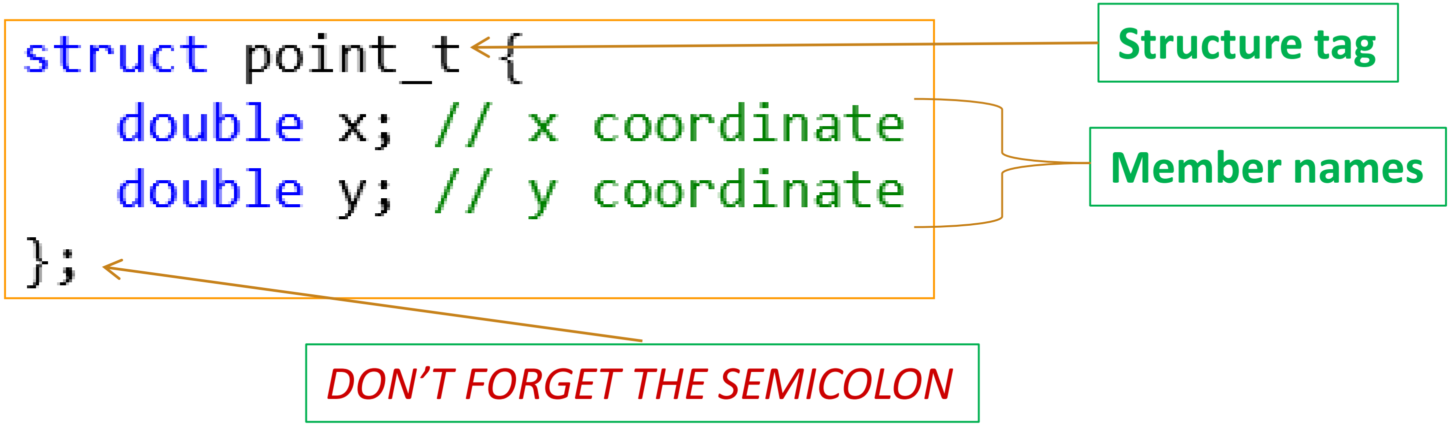
Member names

Variable name

DON'T FORGET THE SEMICOLON

2) Tagged Structure

- A tagged structure definition defines a type
- We can use the tag to define variables, parameters, and return types



```
struct point_t {
   double x; // x coordinate
   double y; // y coordinate
};
```

Structure tag

Member names

DON'T FORGET THE SEMICOLON

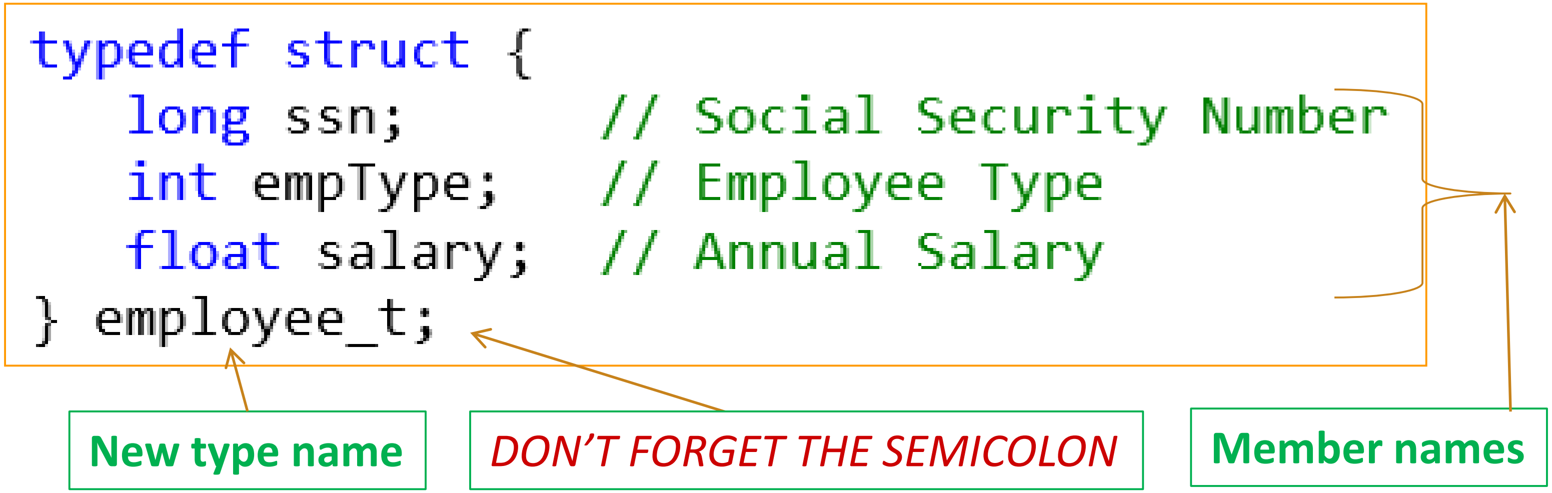
- Variable definitions:

```
struct point_t point1, point2, point3;
```

- Variables point1, point2, and point3 all have members x and y.

3) Typedef Structure

- A typed-defined structure allows the definition of variables without the struct keyword.
- We can use the tag to define variables, parameters, and return types.



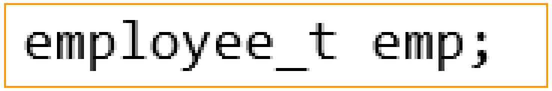
```
typedef struct {
   long ssn;        // Social Security Number
   int empType;     // Employee Type
   float salary;    // Annual Salary
} employee_t;
```

New type name

DON'T FORGET THE SEMICOLON

Member names

- Variable definition:

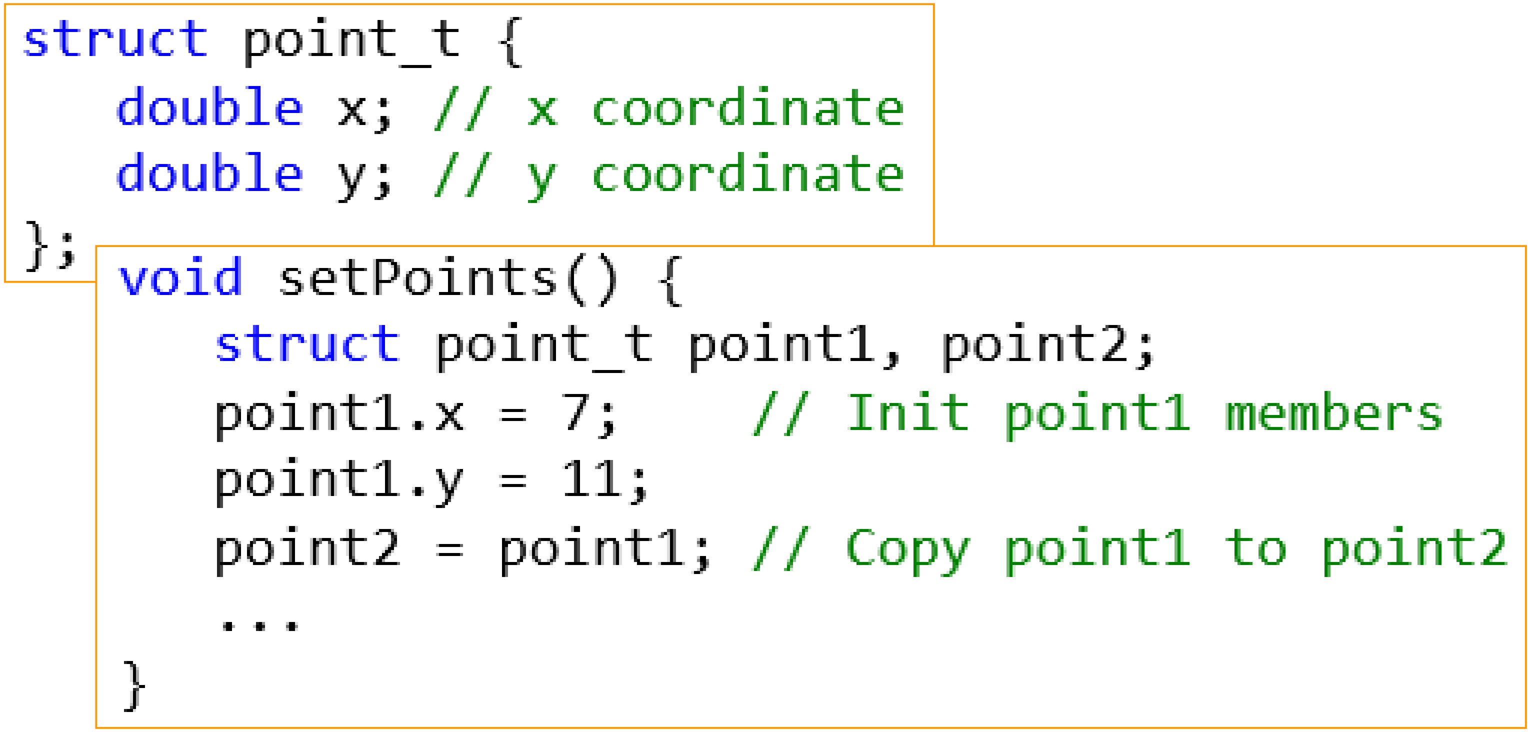


```
employee_t emp;
```

- Variable emp has members ssn, empType, and salary.

Dot Operator (.)

- Used to access member variables
  - Syntax:
    `structure_variable_name.member_name`
  - These variables may be used like any other variables

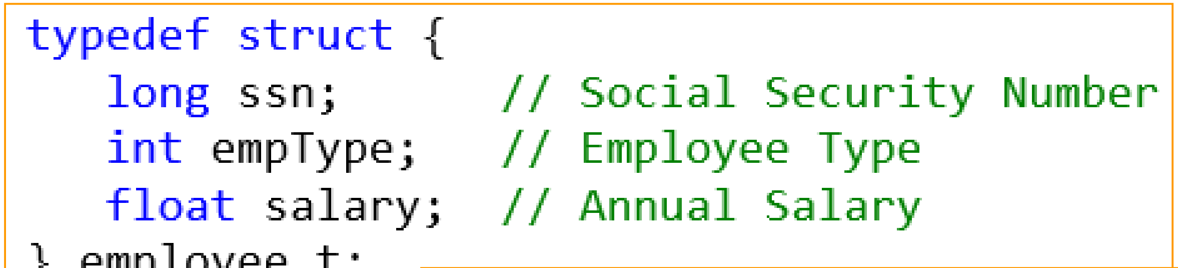


```
struct point_t {
   double x; // x coordinate
   double y; // y coordinate
};
```

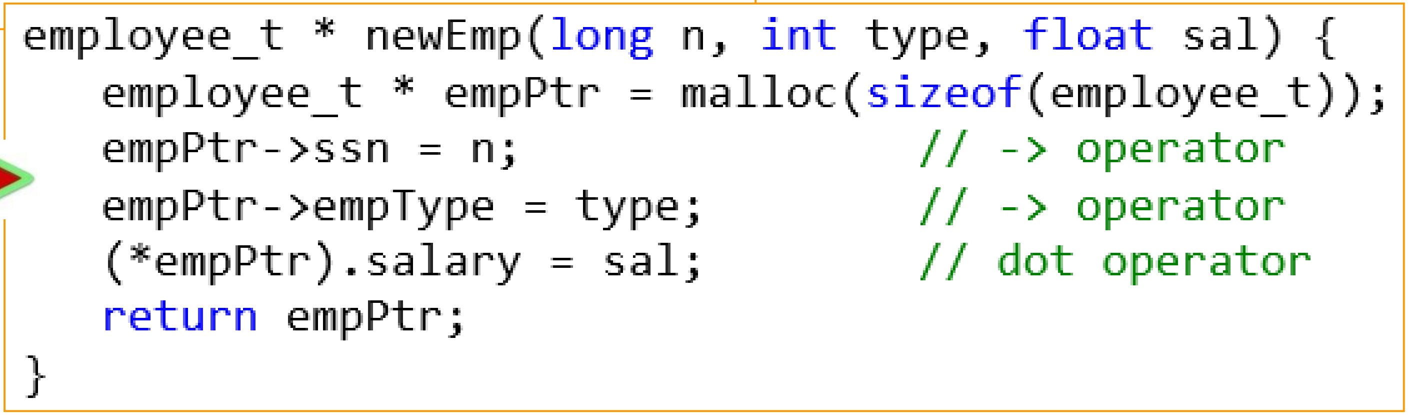
```
void setPoints() {
   struct point_t point1, point2;
   point1.x = 7;     // Init point1 members
   point1.y = 11;
   point2 = point1; // Copy point1 to point2
   ...
}
```

Arrow Operator (->)

- Used to access member variables using a pointer
  - Arrow Operator Syntax:
    structure_variable_pointer->member_name
  - Dot Operator Syntax:
    (*structure_variable_pointer).member_name



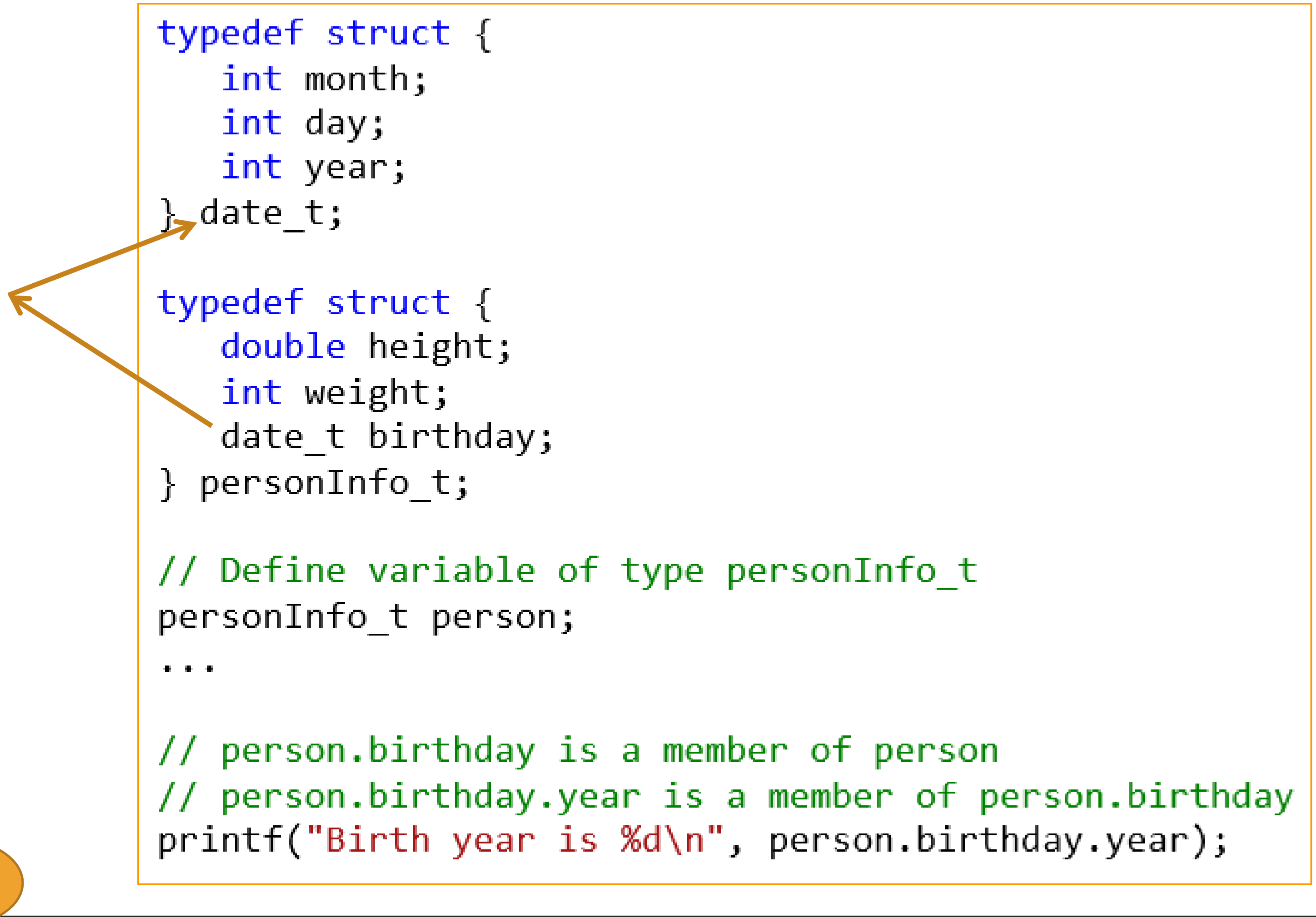
```
typedef struct {
   long ssn;        // Social Security Number
   int empType;     // Employee Type
   float salary;    // Annual Salary
} employee_t;
```



```
employee_t * newEmp(long n, int type, float sal) {
   employee_t * empPtr = malloc(sizeof(employee_t));
   empPtr->ssn = n;                  // -> operator
   empPtr->empType = type;           // -> operator
   (*empPtr).salary = sal;           // dot operator
   return empPtr;
}
```

Nested Structures

- A member that is of a structure type is nested

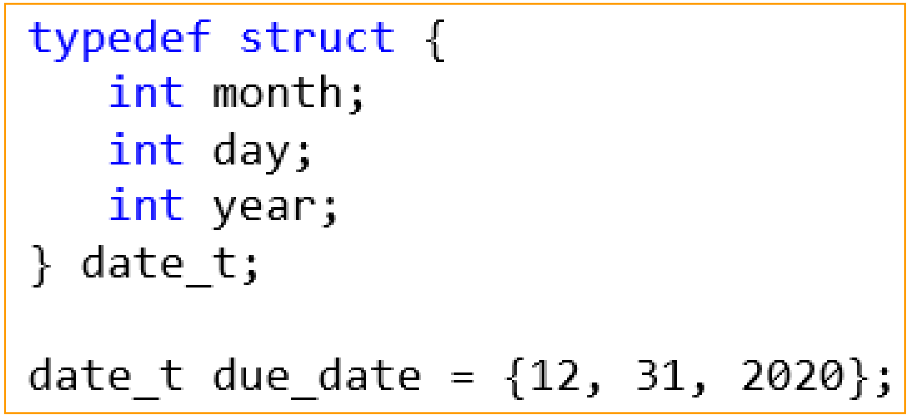


```c
typedef struct {
   int month;
   int day;
   int year;
} date_t;

typedef struct {
   double height;
   int weight;
   date_t birthday;
} personInfo_t;

// Define variable of type personInfo_t
personInfo_t person;
...

// person.birthday is a member of person
// person.birthday.year is a member of person.birthday
printf("Birth year is %d\n", person.birthday.year);
```

Initializing...

Initializing Structures

- A structure may be initialized at the time it is declared
- Order is essential
  - The sequence of values is used to initialize the successive variables in the struct
- It is an error to have more initializers than members
- If fewer initializers than members, the initializers provided are used to initialize the data members
  - The remainder are initialized to 0 for primitive types



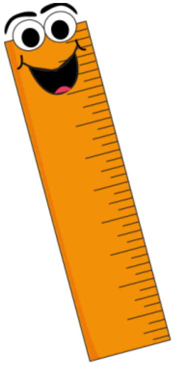
```
typedef struct {
    int month;
    int day;
    int year;
} date_t;

date_t due_date = {12, 31, 2020};
```

Dynamic Allocation of Structures

- The *sizeof()* operator should always be used in dynamic allocation of storage for structured data types and in reading and writing structured data types

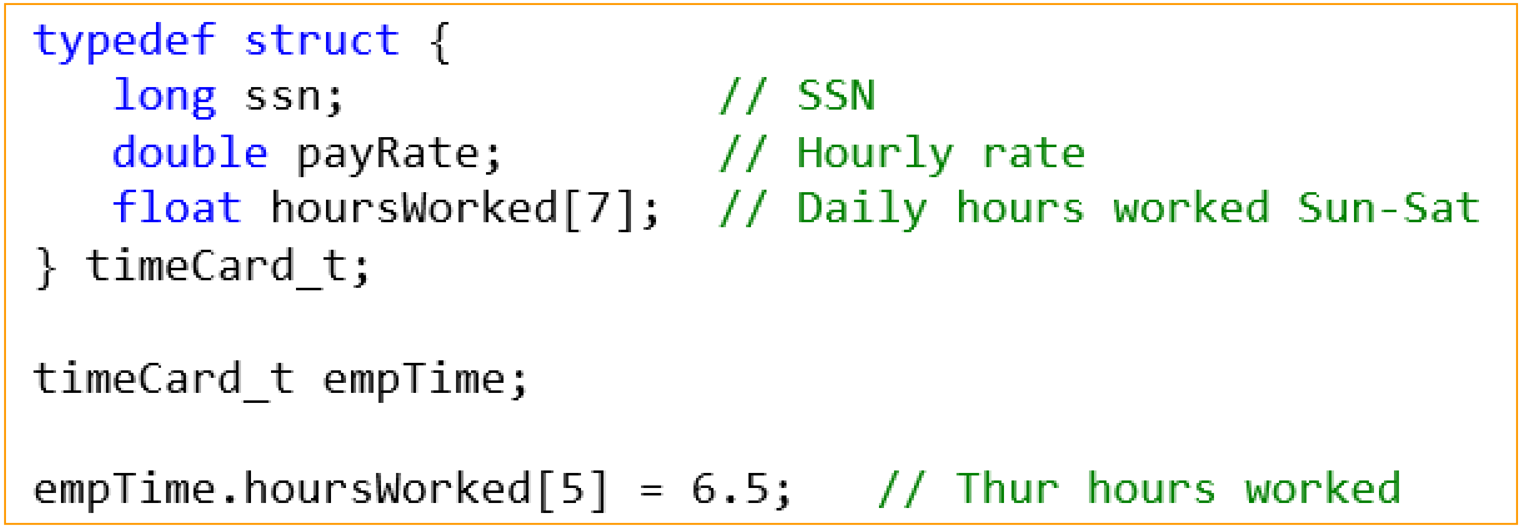
```
typedef struct {
    int month;
    int day;
    int year;
} date_t;

date_t due_date;

int date_t_len = sizeof(date_t);       // sizeof type
int due_date_len = sizeof(due_date);   // sizeof variable

printf("sizeof(date_t)=%d\n", date_t_len);
printf("sizeof(due_date)=%d\n", due_date_len);

date_t * due_dates = calloc(100, sizeof(date_t));
```

```
sizeof(date_t)=12
sizeof(due_date)=12
```

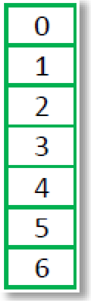


Arrays Within Structures

- A member of a structure may be an array

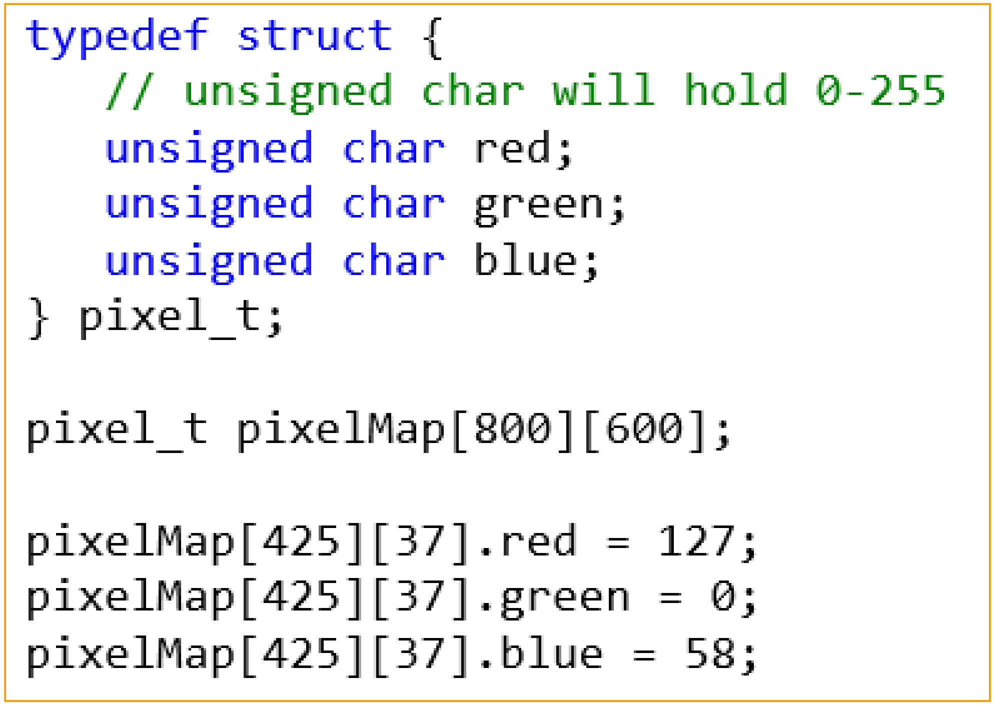


```
typedef struct {
   long ssn;                // SSN
   double payRate;          // Hourly rate
   float hoursWorked[7];  // Daily hours worked Sun-Sat
} timeCard_t;

timeCard_t empTime;

empTime.hoursWorked[5] = 6.5;    // Thur hours worked
```



Arrays of Structures

- We can also create an array of structure types



```
typedef struct {
   // unsigned char will hold 0-255
   unsigned char red;
   unsigned char green;
   unsigned char blue;
} pixel_t;

pixel_t pixelMap[800][600];

pixelMap[425][37].red = 127;
pixelMap[425][37].green = 0;
pixelMap[425][37].blue = 58;
```

Arrays of Structures Containing Arrays

- We can also create an array of structures that contain arrays

```
typedef struct {
   long ssn;                 // SSN
   double payRate;           // Hourly rate
   float hoursWorked[7];     // Daily hours worked Sun-Sat
} timeCard_t;

timeCard_t empTime[1000];

// Thur hours worked, emp # 10

empTime[9].hoursWorked[5] = 6.5;
```

Structures as Parameters

- A struct, like an int, may be passed to a function
- The process works just like passing an int, in that:
  - The complete structure is copied to the stack
  - Called function is unable to modify
    the caller's copy of the variable

Structures as Parameters

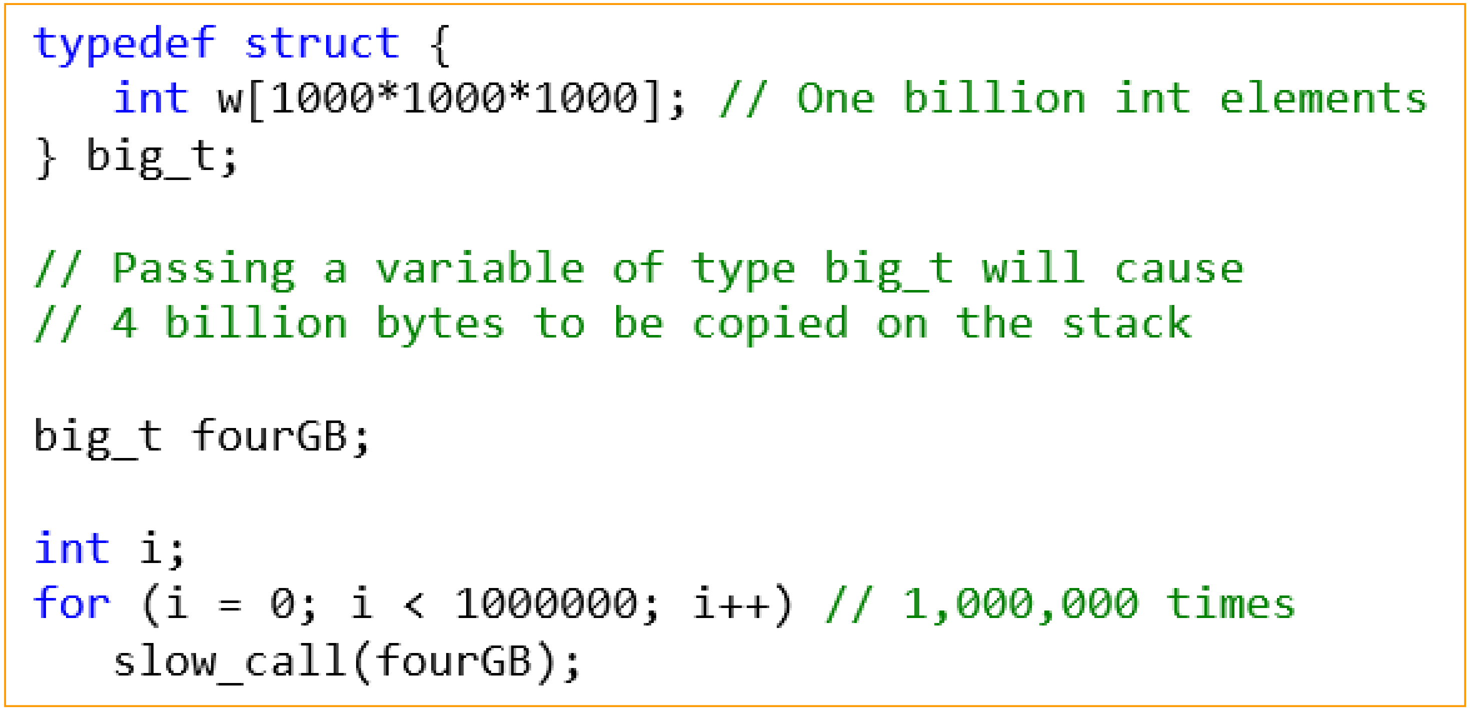
```c
typedef struct {
   double x; // x coordinate
   double y; // y coordinate
} point_t;

void changePoint(point_t p) {
   printf("x=%.1lf, y=%.1lf\n", p.x, p.y);
   //
   p.x = 3.4;
   p.y = 4.5;
}

void mainPoint() {
   point_t point = {1.2, 2.3};
   changePoint(point);
   printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
   //
}
```

```
x=1.2, y=2.3
```

```
x=1.2, y=2.3
```

Structures as Parameters

- Disadvantage of passing structures by value: Copying large structures onto stack
  - Is inefficient
  - May cause stack overflow

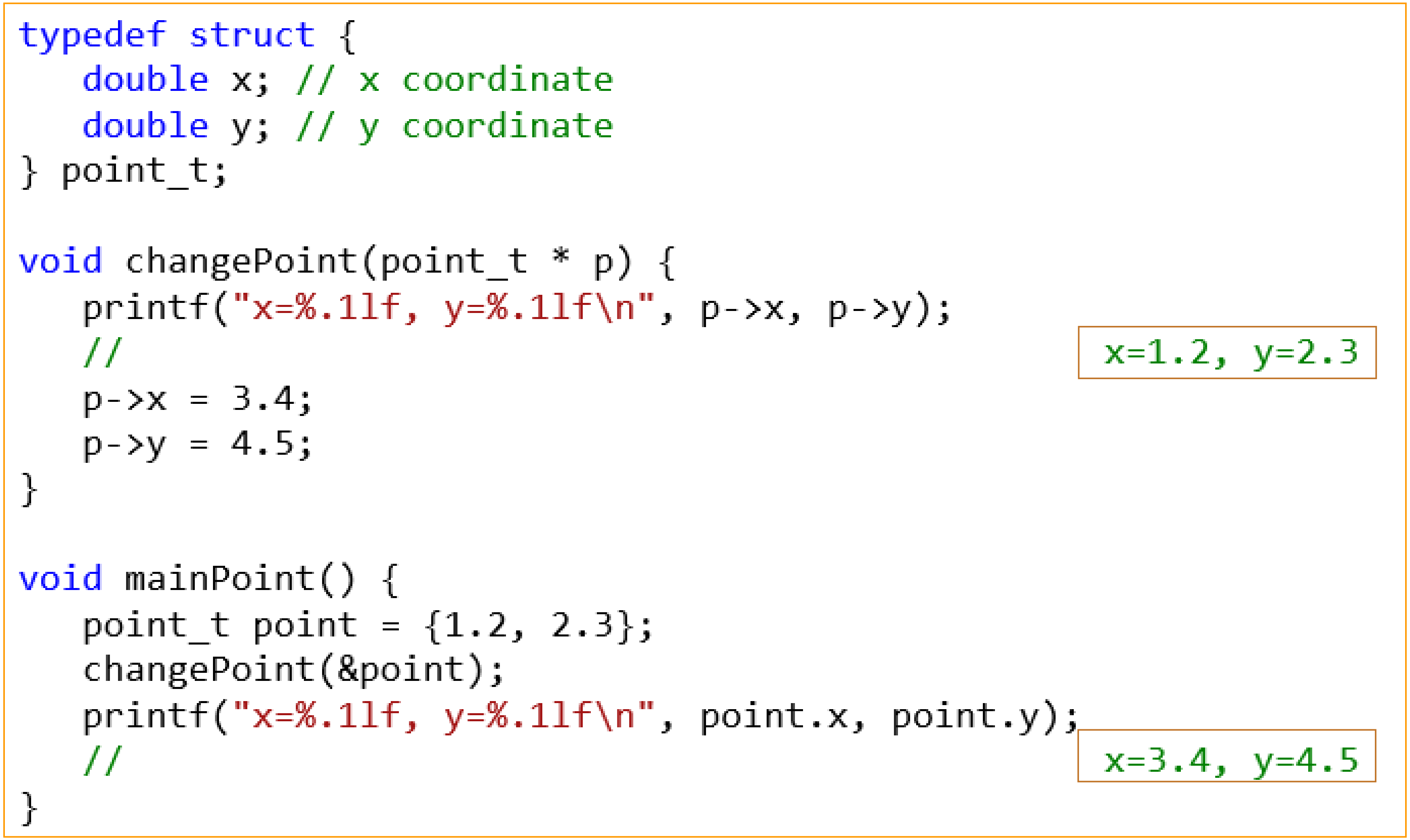


```
typedef struct {
   int w[1000*1000*1000]; // One billion int elements
} big_t;

// Passing a variable of type big_t will cause
// 4 billion bytes to be copied on the stack

big_t fourGB;

int i;
for (i = 0; i < 1000000; i++) // 1,000,000 times
   slow_call(fourGB);
```

Structure Pointers as Parameters

- More efficient: Pass the address of the struct
- Passing an address requires that only a single word be pushed on the stack, no matter the size
  - Called function can then modify the structure.

Structure Pointers as Parameters



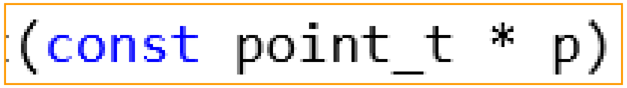
```c
typedef struct {
   double x; // x coordinate
   double y; // y coordinate
} point_t;

void changePoint(point_t * p) {
   printf("x=%.1lf, y=%.1lf\n", p->x, p->y);
   //
   p->x = 3.4;
   p->y = 4.5;
}

void mainPoint() {
   point_t point = {1.2, 2.3};
   changePoint(&point);
   printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
   //
}
```

```
x=1.2, y=2.3
```

```
x=3.4, y=4.5
```

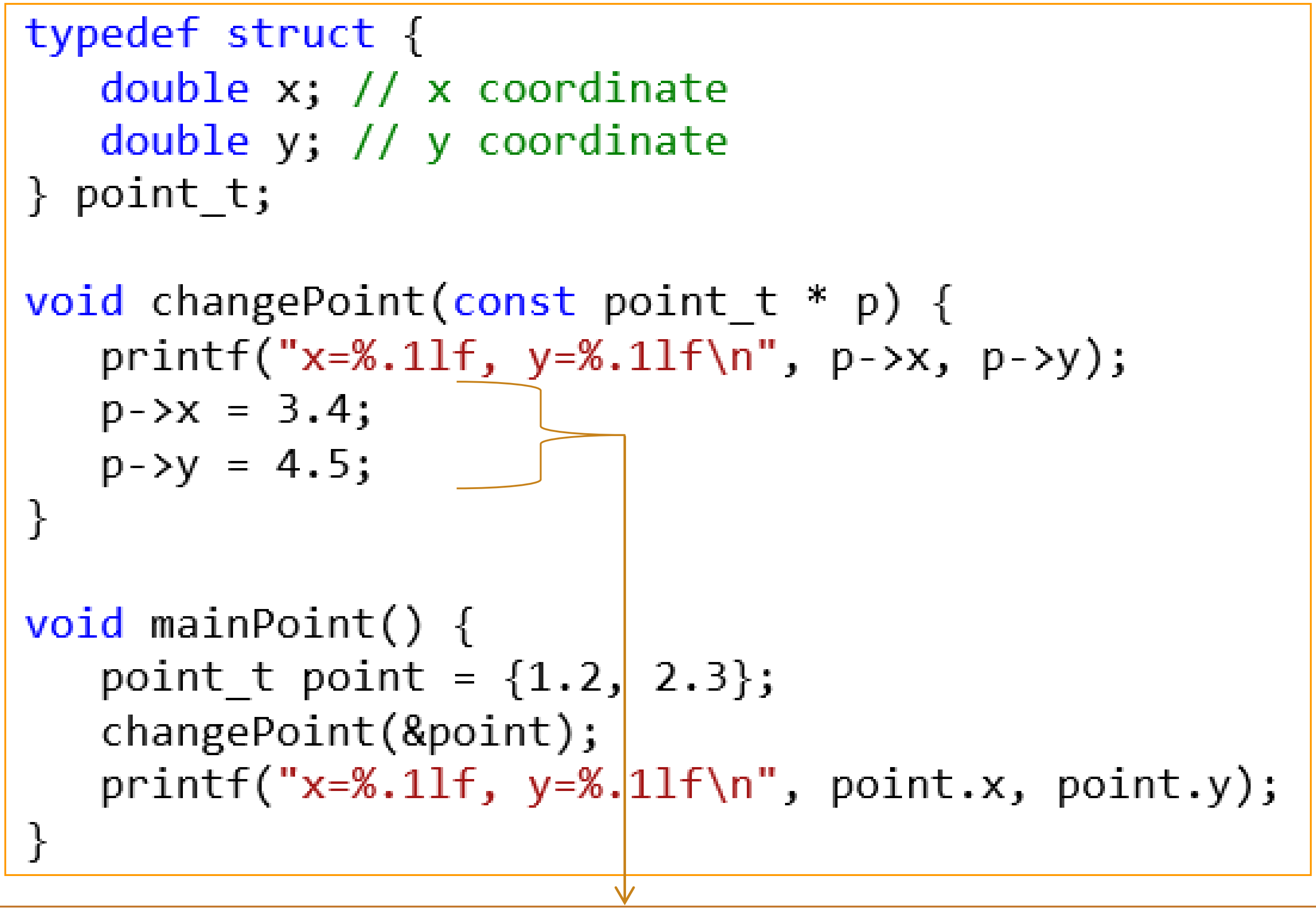
Const Struct Parameter

- What if you do not want the recipient to be able to modify the structure?
  - Use the const modifier



```
(const point_t * p)
```

Using the const Modifier



```c
typedef struct {
   double x; // x coordinate
   double y; // y coordinate
} point_t;

void changePoint(const point_t * p) {
   printf("x=%.1lf, y=%.1lf\n", p->x, p->y);
   p->x = 3.4;
   p->y = 4.5;
}

void mainPoint() {
   point_t point = {1.2, 2.3};
   changePoint(&point);
   printf("x=%.1lf, y=%.1lf\n", point.x, point.y);
}
```

```
ch08.c: In function âchangePointâ:
ch08.c:213:7: error: assignment of member âxâ in read-only object
ch08.c:214:7: error: assignment of member âyâ in read-only object
```

Return Structure

- Scalar values (*int, float, etc)* are efficiently returned in CPU registers
- Historically, the structure assignments and the return of structures was not supported in C
- But, the return of *pointers (addresses)*, including pointers to structures, has always been supported

Return Structure Pointer to Local Variable

```
typedef struct {
   // unsigned char will hold 0-255
   unsigned char red;
   unsigned char green;
   unsigned char blue;
} pixel_t;

pixel_t * getEmptyPixel() {
   // empty pixel = zeros
   pixel_t p = {0, 0, 0};

   // return pointer to empty pixel
   return &p;
}

pixel_t ePixel;
pixel_t * pixelPtr;

pixelPtr = getEmptyPixel();

// Immediately use return
ePixel = *pixelPtr;
```

```
ch08.c: In function âgetEmptyPixelâ:
ch08.c:293:7: warning: function returns address of local variable
```

Return Structure Pointer to Local Variable

- Reason: function is returning a pointer to a variable that was allocated on the stack during execution of the function

  - Such variables are subject to being wiped out by subsequent function calls

Function Return Structure Values

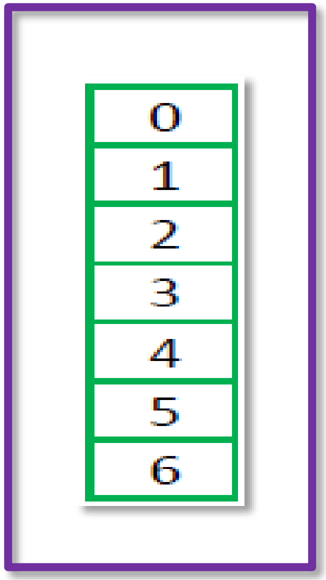
- It is possible for a function to return a structure.
- This facility depends upon the structure assignment mechanisms which copies one complete structure to another.
  - Avoids the unsafe condition associated with returning a pointer, but
  - Incurs the possibly extreme penalty of copying a very large structure

Function Return Structure Values

```c
typedef struct {
   // unsigned char will hold 0-255
   unsigned char red;
   unsigned char green;
   unsigned char blue;
} pixel_t;

pixel_t getEmptyPixel() {
   // empty pixel = zeros
   pixel_t p = {0, 0, 0};

   // return pointer to empty pixel
   return p;
}

pixel_t ePixel;

ePixel = getEmptyPixel();
```

Arrays as Parameters & Return

- Array's address is passed as parameter
  - Simulates passing by reference
- Embedding array in structure
  - The only way to pass an array by *value* is to embed it in a structure
  - The only way to return an array is to embed it in a structure
  - Both involve copying
    - Beware of size



Programming in C

Chapter 9

Structures

THE END