SYSC 2006
Fall 2019

Carleton
UNIVERSITY
Canada's Capital University

C Structures - Part 2

Pointers and Structures

Revised by C.-H. Lung, Oct 8th, 2019

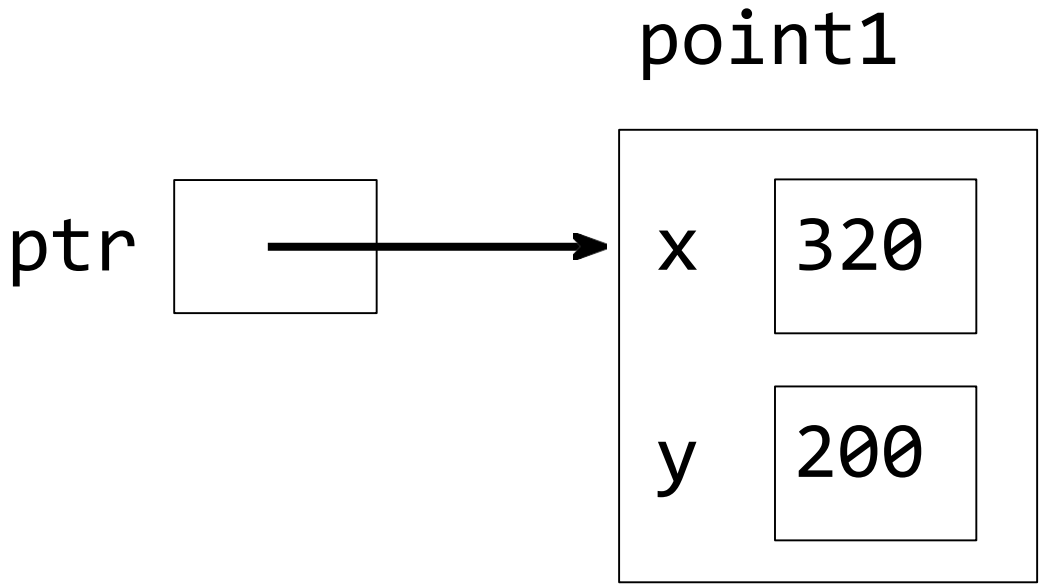
& and Structures

- The **& operator** can be applied to structures

```
point_t* ptr;
point_t point1 = {320, 200};
...
ptr = &point1;
```

- `ptr` now points to `point1`:
  contains the address of the first byte in the memory allocated to `point1`

Memory Diagram



Pointer Dereferencing

- `*ptr` is the entire structure

- `(*ptr).x` and `(*ptr).y` are the members
  - read as: "the `x` member of the structure pointed to by `ptr`"
  - parentheses are required, because the dot operator has higher precedence than *

Pointer Dereferencing

- `*ptr.x` means `*(ptr.x)`, which causes a compilation error
- `ptr` isn't a structure
- it's a ***pointer*** to a structure, so we can't apply the dot operator to it

-> Operator

- If `ptr` is a pointer to a structure, **`ptr->member`** is a shorthand for `(*ptr).member`

```
point_t* ptr;
point_t point1;
...
ptr = &point1;
ptr->x = 320;  // (*ptr).x = 320;
ptr->y = 200;  // (*ptr).y = 200;
```

Function Arguments

- Don't pass large structures as function arguments
  - pass-by-value semantics
  - **copying an entire structure requires time and memory** (the function parameter)

- Instead, **pass pointers to structures** as function arguments

addpoints (Version 1)

- This function changes the structure pointed to by parameter ptr1

```
void addpoints(point_t* ptr1,
               const point_t* ptr2)
{
    ptr1->x = ptr1->x + ptr2->x;
    ptr1->y = ptr1->y + ptr2->y;
}

// pointer a constant: cannot change the value they
// pointing to.
// Why using pass by reference then?
```

addpoints (Version 1)

- Typical call:

```
point_t point1, point2;
point1 = makepoint(320, 200);
point2 = makepoint(30, 40);
addpoints(&point1, &point2);
```

- When addpoints returns, point1 contains the sum of the two points, i.e., point1 gets changed.

Memory Diagram Exercise

- Draw the activation frames for `addpoints` and the calling function
  - just before `addpoints` returns
  - after the statement `addpoints(&point1, &point2)` is executed
- Use C Tutor to check your diagrams

addpoints (Version 2)

- What if we don't want the function to modify point1?
- Rewrite addpoints so it is passed a pointer to the structure where the **sum** will be stored

```
void addpoints(const point_t* ptr1,
               const point_t* ptr2,
               point_t* sum)
{
    sum->x = ptr1->x + ptr2->x;
    sum->y = ptr1->y + ptr2->y;
}
```

addpoints (Version 2)

- Typical call:

```
point_t point1, point2, result;
point1 = makepoint(320, 200);
point2 = makepoint(30, 40);
addpoints(&point1, &point2,
          &result);
```

- When `addpoints` returns, `result` contains the sum of the two points

Memory Diagram Exercise

- Draw the activation frames for `addpoints` and the calling function
  - just before `addpoints` returns
  - after the statement
    `addpoints(&point1, &point2, &result)`
    is executed
- Use C Tutor to check your diagrams

Returning a Pointer

- Rewrite `addpoints` so that it returns a pointer to a `point_t` structure containing the sum of the two points:

```
point_t* addpoints(const point_t* ptr1,
                   const point_t* ptr2);
```

addpoints (Version 3)

```
point_t* addpoints(const point_t* ptr1,
                   const point_t* ptr2)
{
    point_t sum;

    sum.x = ptr1->x + ptr2->x;
    sum.y = ptr1->y + ptr2->y;
    return &sum;
}

point_t* result = addpoints(&point1, &point2);
```

- What is the dangerous flaw in this code?

Memory Diagram Exercise

- Draw the activation frames for `addpoints` and the calling function
  - just before the statement
    `return &sum;`
    is executed
  - after the statement
    ```
    point_t* result =
                    addpoints(&point1, &point2);
    ```
    is executed
- Use C Tutor to check your diagrams

Returning a Pointer

- Is there a way to write `addpoints` so that it returns a pointer to a `point_t` structure containing the sum of its two arguments, without the flaw?
- Yes, if the structure is allocated on the **heap**
- We'll see how to do this, later