Structures in Racket vs. C

Structures in C are similar to structures in Racket:

Racket:

\[
\begin{align*}
& (\text{struct posn } (x \ y)) \\
& (\text{define } p \ (\text{posn } 1 \ 2)) \\
& (\text{define } a \ (\text{posn-x } p)) \\
& (\text{define } b \ (\text{posn-y } p))
\end{align*}
\]

C:

\[
\begin{align*}
& \text{struct posn } \\
& \begin{cases}
& \text{int } x; \\
& \text{int } y;
\end{cases} \\
& \}; \\
& \text{const struct posn } p = \{1,2\}; \\
& \text{const int } a = p.x; \\
& \text{const int } b = p.y;
\end{align*}
\]

Racket generates selector functions when you define a structure.

Instead of selector functions, C has a structure operator (.) which selects the value of the requested field.

Call Stack and Stack Frames

Suppose the function main calls f, then f calls g, and g calls h.

As the program jumps from function to function, we need to remember the history of the return addresses, as well as all the parameters and local variables. This history is known as the call stack.

The entries that are pushed onto the call stack are known as stack frames.
Each function call creates a new stack frame that contains the following:

```c
// ===========================
// <function name>:
// <parameter one>: <value>
// <parameter two>: <value>
// ...
// <local variable one>: ( <value> || ??? )
// <local variable two>: ( <value> || ??? )
// ...
// return address: <caller function name>:<line>
// ===========================
```

If a question asks you to draw the call stack, then **for each** stack frame pushed onto the call stack you must provide the above.

For example:

```c
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

**Example: Draw the call stack**

```c
1 int g(void){
2     int x = 2;
3     return x;
4 }
5
6 int f(int x){
7     int a = g();
8     return x * a;
9 }
10
11 int main(void){
12     int x = 4;
13     int a = 1;
14     int y = f(x) + a;
15 }
```
Example: Draw the call stack

```
int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

---

```
Example: Draw the call stack

int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

---

```
Example: Draw the call stack

int g(void){
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```
**Example: Draw the call stack**

```c
int g(void)
{
    int x = 2;
    return x;
}

int f(int x){
    int a = g();
    return x * a;
}

int main(void){
    int x = 4;
    int a = 1;
    int y = f(x) + a;
}
```

---

**Structures on the Stack**

- Structures are just groups of regular variables.
- Also stored on the stack in the order fields are declared.
- Passed by value (one reason pointers are useful).

---

For Example:

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```
Example: Structures on the call stack

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```

Example: Structures on the call stack

```c
struct foo {
    int x;
    int y;
};

int baz(struct foo qux) {
    return qux.x + 1;
}

int main(void) {
    struct foo bar = {5, 6};
    int x = baz(bar);
    return 0;
}
```
Structures: Exercise

Define the following C functions:

// point_equal(p1, p2) determines if p1 == p2
// requires: p1 and p2 are not null
// effects: none

// triangle_exact_equal(t1, t2) determines
// if t1 exactly equals t2 for a,b,c
// requires: t1 and t2 are not null
// effects: none

// triangle_rot(t) rotates the points a, b, c => c, b, a
// requires: t is not null
// effects: none