Typed Imperative Programming

CS 211

Winter 2020

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl -k $URL211/lec/02typed_imp.tgz | tar zxv
…
% cd 02typed_imp
```

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl -k $URL211/lec/02typed_imp.tgz | tar zxv
…
% cd 02typed_imp
```

- The *curl*(1) command downloads a URL and prints its contents to its standard output (*stdout*).

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl -k $URL211/lec/02typed_imp.tgz | tar zxv
…
% cd 02typed_imp
```

- The *curl*(1) command downloads a URL and prints its contents to its standard output (*stdout*).
- The *tar*(1) command extracts various forms of archives. (The "(1)" means you can get help by running `man 1 tar`.)

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl -k $URL211/lec/02typed_imp.tgz | tar zxv
…
% cd 02typed_imp
```

- The *curl*(1) command downloads a URL and prints its contents to its standard output (*stdout*).
- The *tar*(1) command extracts various forms of archives. (The "(1)" means you can get help by running `man 1 tar`.)
- The `|` character is a Unix "pipe," which attaches the first command's stdout to the second command's stdin.

Problem: Compute the nth Fibonacci number

Definition

$$fib(n) \begin{cases} n & \text{if } n < 2; \\ fib(n-2) + fib(n-1) & \text{otherwise.} \end{cases}$$

Definition

$$fib(n) \begin{cases} n & \text{if } n < 2; \\ fib(n-2) + fib(n-1) & \text{otherwise.} \end{cases}$$

n	$fib(n)$
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21

In C (don’t do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

In C (don’t do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}

long fib(int n)
{
    return (n < 2)? n : (fib(n - 2) + fib(n - 1));
}
```

In C (don’t do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}

long fib(int n){return n<2?n:fib(n-2)+fib(n-1);}
```

In C (don’t do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}

long fib(int n){
return n<2?n:fib
(n-2)+fib(n-1);}
```

In C (don't do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

Things to notice:

- Static types `int`[a] and `long`[b] must be given for variables (argument n) and function results.
- This function does computation but not input/output.

[a] a fixed-width machine "integer"

[b] also a fixed-width "integer," but maybe wider

In C (less weird but still slow [and weird])

```
long fib(int n)
{
    if (n < 2) {
        return n;
    } else {
        long b = fib(n - 1);
        return a + fib(n - 1);
    }
}
```

In C (less weird but still slow [and weird])

```
long fib(int n)
{
    if (n < 2) {
        return n;
    } else {
        long b = fib(n - 1);
        return a + fib(n - 1);
    }
}
```

Syntax of `if`:

```
if (⟨test-expr⟩) {     // evaluate ⟨test-expr⟩; then...
    ⟨then-stms⟩        // do these if ⟨test-expr⟩ was true
} else {
    ⟨else-stms⟩        // do these if ⟨test-expr⟩ was false
}
```

In C (less weird but still slow [and weird])

```
long fib(int n)
{
    if (n < 2) {
        return n;
    } else {
        long b = fib(n - 1);
        return a + fib(n - 1);
    }
}
```

Syntax of variable definition:

⟨*type*⟩ ⟨*var-name*⟩ = ⟨*init-expr*⟩;

Semantics: allocate space named ⟨*var-name*⟩ for a value of type ⟨*type*⟩; evaluate ⟨*init-expr*⟩ and store its result there.

In C (less weird but still slow [and weird])

```
long fib(int n)
{
    if (n < 2) {
        return n;
    } else {
        long b = fib(n - 1);
        return a + fib(n - 1);
    }
}
```

Syntax of `return`:

`return` ⟨*result-expr*⟩`;`

Semantics: evaluate ⟨*result-expr*⟩, then return that value from this function immediately.

In C (less weird but still slow [and weird])

```
long fib(int n)
{
    if (n < 2) {
        return n;
    } else {
        long b = fib(n - 1);
        return a + fib(n - 1);
    }
}
```

More syntax of `if`:

```
if (⟨test-expr⟩) {
    ⟨then-stms⟩
}
```

Everything nests

```
if (⟨first-test-expr⟩) {   // But don't write this.
    ⟨A-stms⟩
} else {
    if (⟨second-test-expr⟩) {
        ⟨B-stms⟩
    } else {
        ⟨C-stms⟩
    }
}
```

Everything nests

```
if (⟨first-test-expr⟩) {   // But don't write this.
    ⟨A-stms⟩
} else {
    if (⟨second-test-expr⟩) {
        ⟨B-stms⟩
    } else {
        ⟨C-stms⟩
    }
}

if (⟨first-test-expr⟩) {   // Do write this.
    ⟨A-stms⟩
} else if (⟨second-test-expr⟩) {
    ⟨B-stms⟩
} else {
    ⟨C-stms⟩
}
```

Problem: It’s super slow

Solution: Mutation (and iteration)

The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```

The essence of imperative programming

```
  int a = 5;
▶ int b = 8;
  int c;

  c = a + b;
  a = b;
  b = c;

  c = a + b;
  a = b;
  b = c;

  c = a + b;
  a = b;
  b = c;
```

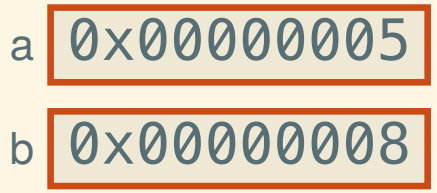
The essence of imperative programming

```
  int a = 5;
  int b = 8;
▶ int c;

  c = a + b;
  a = b;
  b = c;

  c = a + b;
  a = b;
  b = c;

  c = a + b;
  a = b;
  b = c;
```



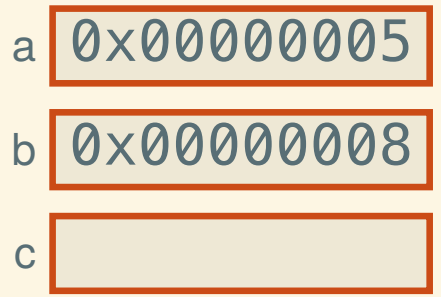
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



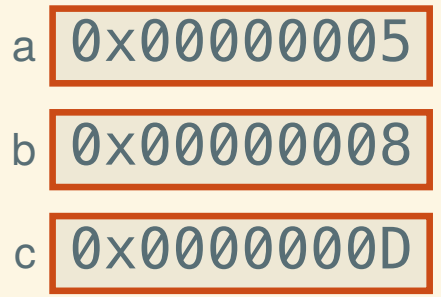
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



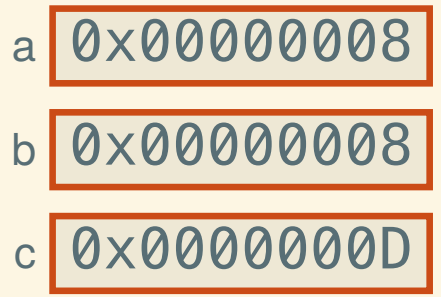
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



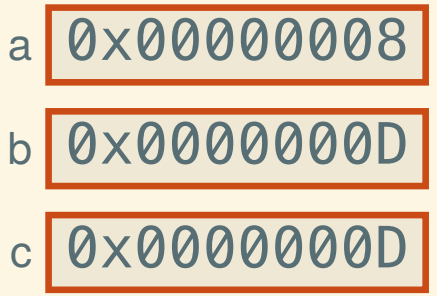
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



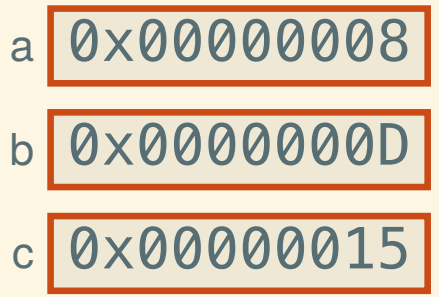
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



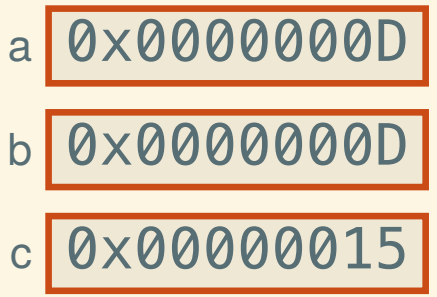
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



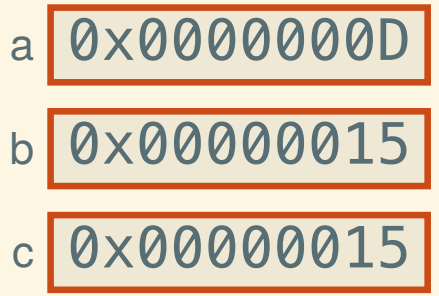
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

▶ c = a + b;
a = b;
b = c;
```



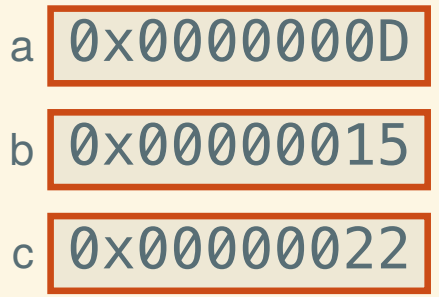
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



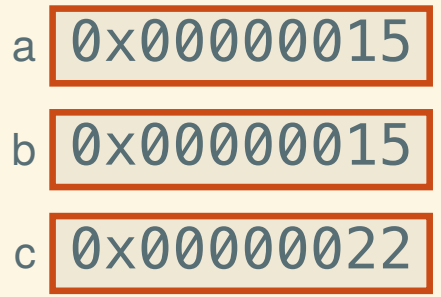
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



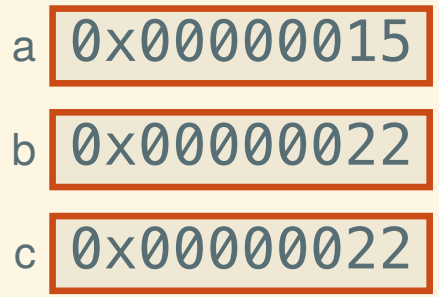
The essence of imperative programming

```
int a = 5;
int b = 8;
int c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;

c = a + b;
a = b;
b = c;
```



What's happening with variables, definitions, and assignments?

```
int z = 5; double d = 5; char c = 5;
```

What's happening with variables, definitions, and assignments?

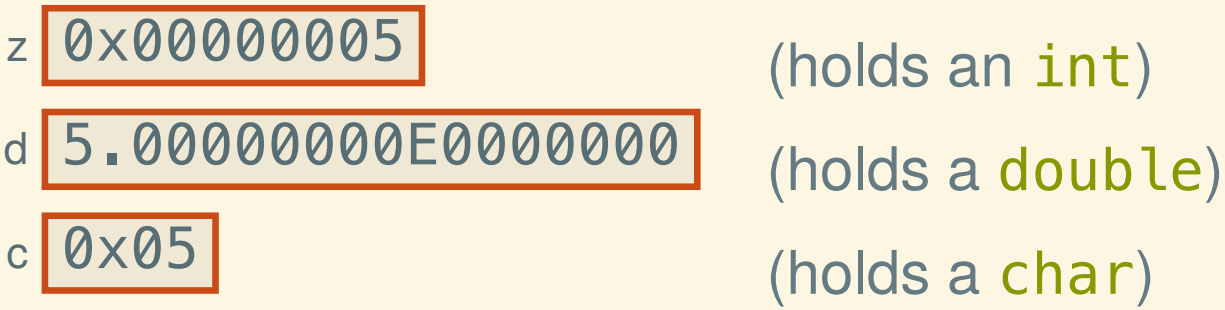
```
int z = 5; double d = 5; char c = 5;
```

- Numbers (*e.g.,* `0x000000A6` and `3.57e-99`) are *values*.

What's happening with variables, definitions, and assignments?

```
int z = 5; double d = 5; char c = 5;
```

- Numbers (*e.g.,* `0x000000A6` and `3.57e-99`) are *values*.
- An *object* is a location where you can store a particular type of value:



What's happening with variables, definitions, and assignments?

```
int z = 5; double d = 5; char c = 5;
```

- Numbers (*e.g.,* `0x000000A6` and `3.57e-99`) are *values*.
- An *object* is a location where you can store a particular type of value:

  z `0x00000005` (holds an `int`)

  d `5.00000000E0000000` (holds a `double`)

  c `0x05` (holds a `char`)
- A *variable* is the name of an object (like `z` and d).

What's happening with variables, definitions, and assignments?

```
int z = 5; double d = 5; char c = 5;
z += c;
```

- Numbers (*e.g.,* `0x000000A6` and `3.57e-99`) are *values*.
- An *object* is a location where you can store a particular type of value:

  z `0x00000005` (holds an `int`)

  d `5.00000000E0000000` (holds a `double`)

  c `0x05` (holds a `char`)
- A *variable* is the name of an object (like `z` and d).
- An assignment modifies the value stored in an object.

What's happening with variables, definitions, and assignments?

```
int z = 5; double d = 5; char c = 5;
z += c;
```

- Numbers (*e.g.,* `0x000000A6` and `3.57e-99`) are *values*.
- An *object* is a location where you can store a particular type of value:

z	`0x0000000A`	(holds an `int`)
d	`5.00000000E0000000`	(holds a `double`)
c	`0x05`	(holds a `char`)

- A *variable* is the name of an object (like `z` and d).
- An assignment modifies the value stored in an object.

The other ingredient: iteration with `while`

Syntax:

```
while (⟨test-expr⟩) {
    ⟨body-stms⟩
}
```

The other ingredient: iteration with `while`

Syntax:

```
while (⟨test-expr⟩) {
    ⟨body-stms⟩
}
```

Semantics:

1. Evaluate ⟨*test-expr*⟩ to a `bool`.
2. If the `bool` is false then the loop is finished, so jump to the next statement after the loop. after it
3. Execute ⟨*body-stms*⟩.
4. Go back to step 1.

In C, iteratively

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr  = next;
        next += prev;
        n    -= 1;
    }

    return curr;
}
```

In C, iteratively

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;    // variable definition

        curr  = next;        // assignment
        next += prev;        // add-to
        n    -= 1;           // subtract-from
    }

    return curr;
}
```

Counting upwards

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    int i = 0;

    while (i < n) {
        long prev = curr;
        curr  = next;
        next += prev;
        ++i;                    // equivalent to i += 1;
    }

    return curr;
}
```

Counting upwards with `for`

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    int i = 0;

    for (; i < n; ) {
        long prev = curr;
        curr  = next;
        next += prev;
        ++i;
    }

    return curr;
}
```

Counting upwards with `for`

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    int i = 0;

    for (; i < n; ++i) {
        long prev = curr;
        curr  = next;
        next += prev;
        // ++i
    }

    return curr;
}
```

Counting upwards with `for`

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    // int i = 0;

    for (int i = 0; i < n; ++i) {
        long prev = curr;
        curr  = next;
        next += prev;
        // ++i
    }

    return curr;
}
```

Counting upwards with `for`

```
long fib(int n)
{
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; ++i) {
        long prev = curr;
        curr  = next;
        next += prev;
    }

    return curr;
}
```

Adding I/O

Reading user input

src/get_input.c

```c
#include <stdio.h>

int main()
{
    int x = 0, y = 0;

    printf("Enter␣two␣integers:␣");
    scanf("%d%d", &x, &y);
    printf("%d␣*␣%d␣==␣%d\n", x, y, x * y);
}
```

Reading user input

src/get_input.c

```c
#include <stdio.h>

int main()
{
    int x = 0, y = 0;

    printf("Enter␣two␣integers:␣");
    scanf("%d%d", &x, &y);
    printf("%d␣*␣%d␣==␣%d\n", x, y, x * y);
}
```

- *scanf*(3) takes a *template* specifying what types of values to read and how many.

Reading user input

src/get_input.c

```c
#include <stdio.h>

int main()
{
    int x = 0, y = 0;

    printf("Enter␣two␣integers:␣");
    scanf("%d%d", &x, &y);
    printf("%d␣*␣%d␣==␣%d\n", x, y, x * y);
}
```

- *scanf*(3) takes a *template* specifying what types of values to read and how many.
- *printf*(3) takes a *template* with holes to fill in with the values of its remaining arguments.

Reading user input

src/get_input.c

```c
#include <stdio.h>

int main()
{
    int x = 0, y = 0;

    printf("Enter␣two␣integers:␣");
    scanf("%d%d", &x, &y);
    printf("%d␣*␣%d␣==␣%d\n", x, y, x * y);
}
```

- *scanf*(3) takes a *template* specifying what types of values to read and how many.
- *printf*(3) takes a *template* with holes to fill in with the values of its remaining arguments.
- `%d` means scan/print an `int` in decimal.

Checking for input errors

src/check_input.c

```c
#include <stdio.h>

int main()
{
    int x, y;
    printf("Enter two integers: ");

    // scanf(3) returns the number of *successful*
    // conversions:
    int count = scanf("%d%d", &x, &y);
    if (count == 2) {
        printf("%d * %d == %d\n", x, y, x * y);
    } else {
        printf("Input error\n");
        return 1;
    }
}
```

A `main` function for the *fib* program

src/fib_iter.c

```c
#include <stdio.h>

long fib(int n)
{ … }

int main()
{
    int input;

    while (scanf("%d", &input) == 1) {
        printf("%ld\n", fib(input));
    }
}
```

Structure types

Structure types in C

C (like BSL/ISL) uses structures to define new data types by composition of existing data types

A structure type has a name and some number of fields, each of which must be declared with a type

Syntax to define a `struct` type

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

struct circle
{
    struct posn center;
    double radius;
};
```

Syntax to define a `struct` type

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

struct circle
{
    struct posn center;
    double radius;
};
```

Note that the type defined by the `struct posn` definition, and used for field `center` of `struct circle` is `struct posn`, not merely `posn`. (In C++ you could refer to it either way, but not in C.)

Syntax to use a structure

Suppose we have a variable `p` whose type is `struct posn`. How do we access `p`'s fields?

Syntax to use a structure

Suppose we have a variable `p` whose type is `struct posn`.
How do we access `p`'s fields? `p.x` and `p.y`

Syntax to use a structure

Suppose we have a variable `p` whose type is `struct posn`. How do we access `p`'s fields? `p.x` and `p.y`

Let's write a function to compute the Manhattan distance between two points. Mathematically,

$$d_1((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

Syntax to use a structure

Suppose we have a variable `p` whose type is `struct posn`. How do we access `p`'s fields? `p.x` and `p.y`

Let's write a function to compute the Manhattan distance between two points. Mathematically,

$$d_1((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

```
// For the fabs(3) function:
#include <math.h>

// Finds the Manhattan distance between two points.
double manhattan_dist(struct posn p, struct posn q)
{
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
```

Defining and initializing a structure

Usually to get a structure in C, first you define a structure variable and then initialize it by *assigning* each field:

```
struct posn p;
p.x = 3.0;
p.y = 4.0;

struct circle c;
c.center.x = 7.0;
c.center.y = -9.2;
c.radius = 6.4;
```

C won't force you to initialize all the fields, but guess what happens if you a access a field that hasn't been initialized?

Factory functions

If you get tired of initializing structures as on the previous slide, you can always define a *factory function* to do the work:

```
struct circle
make_circle(struct posn center, double radius)
{
    struct circle result;
    result.center = center;
    result.radius = radius;
    return result;
}
```

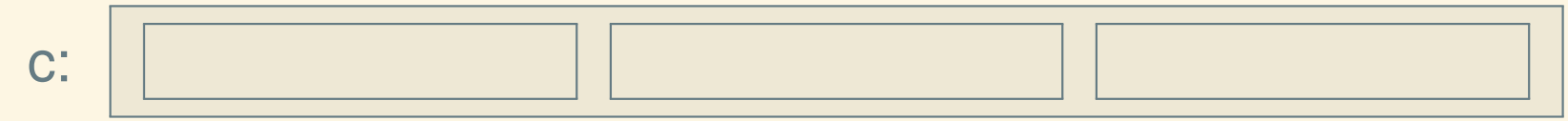
(Note that functions can both take and return structure values.)

Visualizing structure value layout

```
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```

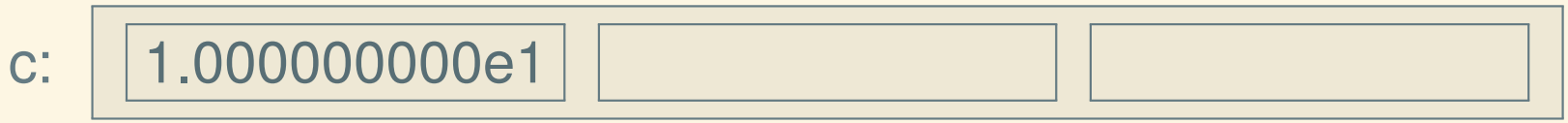
Visualizing structure value layout

```
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```



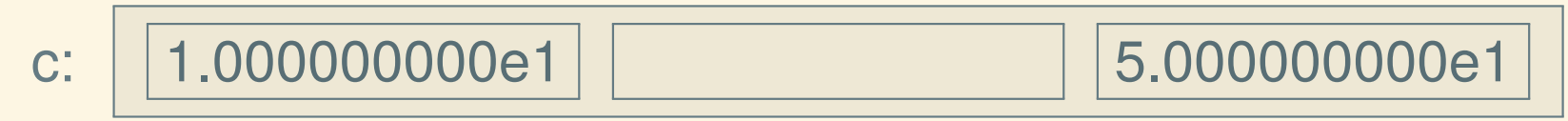
Visualizing structure value layout

```
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```



Visualizing structure value layout

```
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```



Visualizing structure value layout

```
struct circle c;
c.center.x = 10.0;
c.radius = 50.0;
c.center.y = -7.0;
```