Data Structures in C and C++

EECS 214, Fall 2018

Structs

Structs in DSSL2

```
struct posn:
    let x: num?
    let y: num?

struct circle:
    let center: posn?
    let radius: num?
```

Structs in DSSL2

```
struct posn:
    let x: num?
    let y: num?

struct circle:
    let center: posn?
    let radius: num?


def scale_circle(circ, factor):
    circ.radius = factor * circ.radius
```

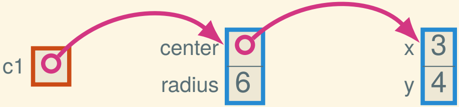
Structs in DSSL2

```
struct posn:
    let x: num?
    let y: num?

struct circle:
    let center: posn?
    let radius: num?



def scale_circle(circ, factor):
    circ.radius = factor * circ.radius

test 'parameter passing creates sharing':
    let c = circle(posn(3, 4), 6)
    scale_circle(c, 2)
    assert_eq c.radius, 12
```

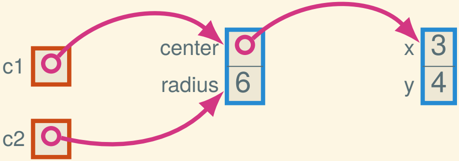
Structs in DSSL2

```
struct posn:
    let x: num?
    let y: num?

struct circle:
    let center: posn?
    let radius: num?



test 'assignment creates sharing':
    let c1 = circle(posn(3, 4), 6)
    let c2 = c1
    c2.radius = 12
    assert_eq c1.radius, 12
```

What is happening in memory (in DSSL2)

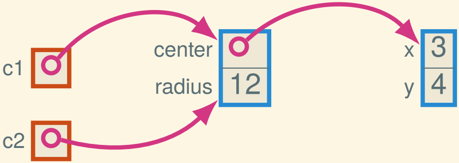


What is happening in memory (in DSSL2)



```
let c2 = c1
```

What is happening in memory (in DSSL2)



```
let c2 = c1

c2.radius = 12
```

Comparison between DSSL2 and C (and C++)

	DSSL2	**C(++)**
size of variables	always the same	depends on type
pointers	to every struct	only when requested

Comparison between DSSL2 and C (and C++)

	DSSL2	**C(++)**
size of variables	always the same	depends on type
pointers	to every struct	only when requested

Other languages like C and C++:

- C# (sort of)
- Swift (sort of)
- Rust

Other languages like DSSL2:

- Java
- Python
- JavaScript

The circle example in C

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

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

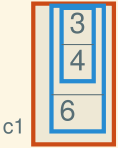
The circle example in C

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

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

void scale_circle(struct circle c2, double factor)
{
    c2.radius *= factor;
}
```

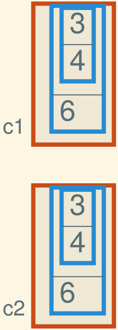
The circle example in C

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

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

int main()
{
    struct circle c1 = { .center = { .x = 3, .y = 4 },
                         .radius = 6 };
    struct circle c2 = c1;
    c2.radius = 12;
}
```

What is happening in memory (in C)

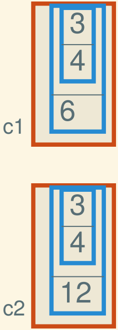


What is happening in memory (in C)



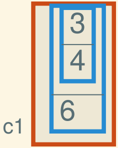
```
struct circle c2 = c1;
```

What is happening in memory (in C)

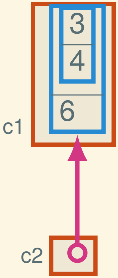


```
struct circle c2 = c1;

c2.radius = 12
```

Getting the address of a variable

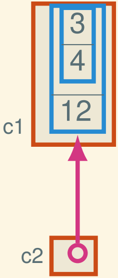


Getting the address of a variable



```
struct circle* c2p = &c1;
```

Getting the address of a variable

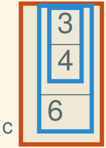


```
struct circle* c2p = &c1;

c2p->radius = 12;
```

Passing a pointer

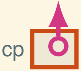
```
void scale_circle(struct circle* cp, double factor)
{
    cp->radius *= factor;
}

int main()
{
    struct circle c = {
            .center = { .x = 3, .y = 4 },
            .radius = 6
    };

    scale_circle(&c);
}
```

Stack allocation



```
struct circle* bad()
{
    struct circle c = { .center = { .x = 3, .y = 4 },
                        .radius = 6 };
    return &c;
}
```

Stack allocation

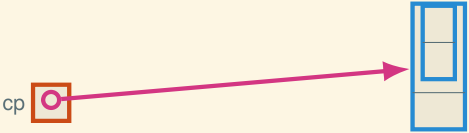


```
struct circle* bad()
{
    struct circle c = { .center = { .x = 3, .y = 4 },
                        .radius = 6 };
    return &c;
}

int main()
{
    struct circle* cp = bad();
}
```

Heap allocation

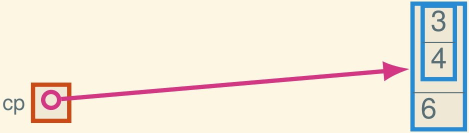
```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



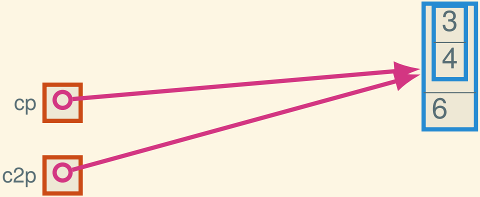
```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



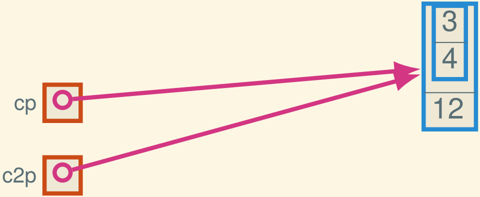
```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



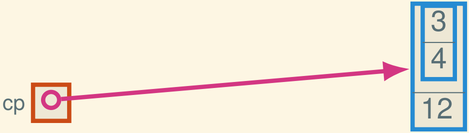
```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



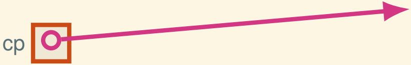
```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Heap allocation



```
int main()
{
    struct circle* cp = malloc(sizeof(struct circle));
    if (cp == NULL) return 1;
    cp->center.x = 3; cp->center.y = 4; cp->radius = 6;

    scale_circle(cp, 2);

    free(cp);
}
```

Arrays

Fixed-sized arrays

Global (statically allocated):

```
int array[10];

void f(size_t i)
{
    array[i] = 7;
}
```

Fixed-sized arrays

Global (statically allocated):

```
int array[10];

void f(size_t i)
{
    array[i] = 7;
}
```

Local (stack allocated):

```
void g(size_t i)
{
    int array[10];
    array[i] = 7;
}
```

Returning an array

This function returns a dangling pointer:

```
int* bad(size_t i)
{
    int array[10];
    array[i] = 7;
    return array;
}
```

Returning an array

This function returns a dangling pointer:

```c
int* bad(size_t i)
{
    int array[10];
    array[i] = 7;
    return array;
}
```

But you can `malloc` up a heap array:

```c
int* good(size_t i)
{
    int* array = malloc(10 * sizeof(int));
    if (array == NULL) out_of_memory();
    array[i] = 7;
    return array;
}
```

Be careful!

```c
int* f(size_t i)
{
    int* array = malloc(10 * sizeof(int));
    if (array == NULL) out_of_memory();
    array[i] = 7;
    return array;
}

void g()
{
    int* array = f(5);

    // prints "7\n"
    printf("%d\n", array[5]);

    free(array);
}
```

Be careful!

```c
int* f(size_t i)
{
    int* array = malloc(10 * sizeof(int));
    if (array == NULL) out_of_memory();
    array[i] = 7;
    return array;
}

void g()
{
    int* array = f(5);

    // UNDEFINED BEHAVIOR!
    printf("%d\n", array[6]);

    free(array);
}
```

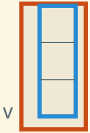
Be careful!

```c
int* f(size_t i)
{
    int* array = malloc(10 * sizeof(int));
    if (array == NULL) out_of_memory();
    array[i] = 7;
    return array;
}

void g()
{
    int* array = f(5);

    // prints "12\n"
    array[6] = 12;
    printf("%d\n", array[6]);

    free(array);
}
```

Be careful!

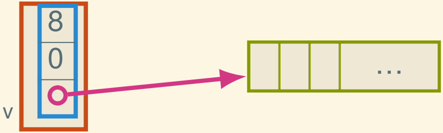
```c
int* f(size_t i)
{
    int* array = malloc(10 * sizeof(int));
    if (array == NULL) out_of_memory();
    array[i] = 7;
    return array;
}

void g()
{
    int* array = f(5);

    // UNDEFINED BEHAVIOR!
    array[10] = 17;

    free(array);
}
```

Representing dynamic arrays in C



```
struct int_da
{
    size_t capacity;
    size_t size;
    int* data;
};

struct int_da v;
```

Representing dynamic arrays in C



```
struct int_da
{
    size_t capacity;
    size_t size;
    int* data;
};

struct int_da v;

v.capacity = 8;
v.size     = 0;
v.data     = malloc(v.capacity * sizeof(int));
```

The pimpl pattern

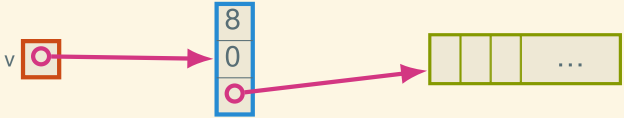
```
// int_da.h
typedef struct int_da* int_da_t;
int_da_t int_da_create(size_t capacity);
```

The pimpl pattern

```
// int_da.h
typedef struct int_da* int_da_t;
int_da_t int_da_create(size_t capacity);

// int_da.c
struct int_da
{
    size_t capacity;
    size_t size;
    int* data;
};
```

The pimpl pattern

```
// int_da.h
typedef struct int_da* int_da_t;
int_da_t int_da_create(size_t capacity);

// int_da.c
struct int_da
{
    size_t capacity;
    size_t size;
    int* data;
};
```



```
    int_da_t v = int_da_create(8);
```

To `int_da.h`, `int_da.c`, `str_da.h`, `str_da.c`, …

Moving to C++

Why C++?

- Generics
- Privacy
- Stuff happens automatically

C++ generics

```
struct int_da
{
    size_t capacity, size;
    int* data;
};

struct str_da
{
    size_t capacity, size;
    char** data;
};
```

C++ generics

```
struct int_da
{
    size_t capacity, size;
    int* data;
};

struct str_da
{
    size_t capacity, size;
    char** data;
};

template <class T>
struct Dyn_array
{
    size_t capacity, size;
    T* data;
}
```

C++ privacy

```
template <class T>
class Dyn_array
{
public:
    Dyn_array();

    size_t size() const;
    void push_back(T const&);
    T& operator[](size_t);

    // …

private:
    size_t capacity_, size_;
    T* data_;
}
```

C++ stuff that happens automatically

```
template <class T>
class Dyn_array
{
public:
    // …

    ~Dyn_array();
    Dyn_array(Dyn_array const&);
    Dyn_array& operator=(Dyn_array const&);

    // …
}
```

To `raii.cpp`, `Dyn_array.hpp`, `Hash_map.hpp`, …

STL: The C++ Standard Template Library

```
#include <vector>           // dynamic array

std::vector<size_t> v1;
std::vector<std::string> v2;
std::vector<std::vector<bool>> v3;
```

STL: The C++ Standard Template Library

```
#include <vector>           // dynamic array

std::vector<size_t> v1;
std::vector<std::string> v2;
std::vector<std::vector<bool>> v3;

#include <unordered_map>    // hash table dictionary
#include <set>              // BST set

template <class T>
class Graph
{
    std::unordered_map<T, std::set<T>> adj_sets_;

    // …
};
```

A less silly way to represent a graph

```
#include <unordered_map>
#include <set>

template <class T>
class Graph
{
public:
    // …
    size_t new_node(T const& name);

private:
    std::unordered_map<T, size_t> name_to_index_;
    std::vector<T*> index_to_name_;
    std::vector<std::set<size_t>> adj_sets_;
};
```

Adding a node

```
template <class T>
size_t Graph<T>::new_node(T const& name)
{
    size_t index = index_to_name_.size();

    auto pair = name_to_index_.insert({name, index});
    if (!pair.second)
        throw name_already_exists(name);

    index_to_name.push_back(&pair.first->first);
    adj_sets_.emplace_back();

    return index;
}
```

Next time: Bloom filters