CS443: Compiler Construction

Lecture 8: Structs

Stefan Muller

Based on material from Stephen Chong, Steve Zdancewic and Greg Morrisett

2D (3D, etc.) Arrays

- By convention: “row major” order

```
int a[3][5]
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]

2D (3D, etc.) Arrays

- By convention: “row major” order

```
int a[3][5]
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]

Addr of a[y][x] = 5 * y + x

Structs in C and LLVM IR

```
struct person
{
  char name[];
  int age;
};

{ i8*, i32 }
```

Recursive structs in C and LLVM IR

```
struct node
{
  int hd;
  node *tl;
};
```

```
%Tnode = type { i32, %Tnode* }
```

(Can name non-recursive structs too)

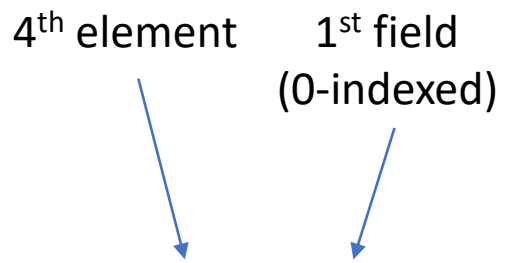
`getelementptr` (general)

```
%elptr = getelementptr <ty>, <ty>* %ptr, <intty1> <val1>, …, <inttyN> <valN>
```

- <ty> is a (possibly structured) type
- %ptr is a pointer to an array of <ty>s (might just have one element)
- <val1> is the index into the array
- <val2> is the index of a field in the structure (if <ty> is a structure)
- <val3> is the index of the field in *that* structure (if the <val2>th element of <ty> is a structure)…

(For structs, indices must be `i32` *constants*)

```
struct person { char name []; int age; };
person classlist[] = person[10];
classlist[4].age = 20;
```




```
%Tperson = { i8*, i32 }
%p4age = getelementptr %Tperson, %Tperson* %classlist, i32 4, i32 1
store i32 20, i32* %p4age
```

GEP Example

```
struct RT {
     int A;
     int B[10][20];
     int C;
}
struct ST {
     struct RT X;
     int Y;
     struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `size_ty(%ST)`.
3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.
5. Index into the 2d array.

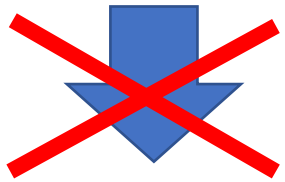
```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
     %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
     ret i32* %arrayidx
}
```

Final answer: ADDR + `size_ty(%ST)` + `size_ty(%RT)` + `size_ty(i32)` + `size_ty(i32)` + `5*20*size_ty(i32)` + `13*size_ty(i32)`

Adapted from the LLVM reference by Stephen Chong, Harvard University

Getelementptr does not access memory (ever!)

```
struct node { int hd; node *tl; };
node tltl = list.tl->tl;
```



```
%tltl = getelementptr %Tnode, %Tnode* %list, i32 0, i32 1, i32 1
```

Getelementptr does not access memory (ever!)

```
struct node { int hd; node *tl; };
node tltl = list.tl->tl;
```

```
%tlptr = getelementptr %Tnode, %Tnode* %list, i32 0, i32 1
%tl = load %Tnode, %Tnode* %tlptr
%tltlptr = getelementptr %Tnode, %Tnode* %tl, i32 0, i32 1
%tltl = load %Tnode, %Tnode* %tltlptr
```

MiniC Syntax

t ::= void | bool | char | int | t[] | s | t((t id,)*)

b ::= + | - | * | / | && | || | > | >= | < | <= | != | ==

u ::= - | !

c ::= n | ‘alpha’

l ::= x | x[e] | x.f

e ::= c | x | e b e | u e | l = e | new(t) | e((e,)*) | e[e] | e.f | (t) e

s ::= t x [= e] | { (s;)* } | e | if e s else s | while e s | break | continue
| return [v]

d ::= (t x [= e])* | t id ((t id,)*) | struct id {(t id;)*}

Arrays in MiniC

```
int a[] = new(int[20]);
a[4] = 42;
a[5] = a[4] + 1;
```

Structs in MiniC

```
struct person
{
  char name [];
  int age;
};

int main () {
  char my_name [] = new(char[6]);
  my_name[0] = ‘s’; my_name[1] = ‘t’; …;
  person stefan = new(person);
  stefan.name = my_name;
}
```

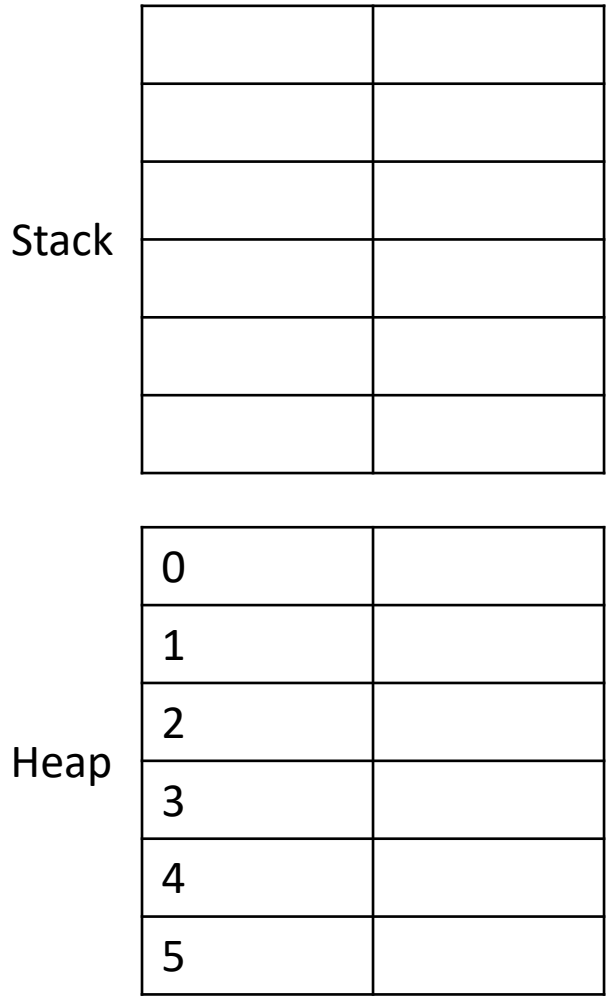
You can pass structs and arrays around

```
void print_arr(char a[]) {
  int i = 0;
  while ((int)(a[i]) != 0) { printf(“%c”, a[i]); }
  return;
}

void print_name(person p) {
  print_arr(p.name);
}
```

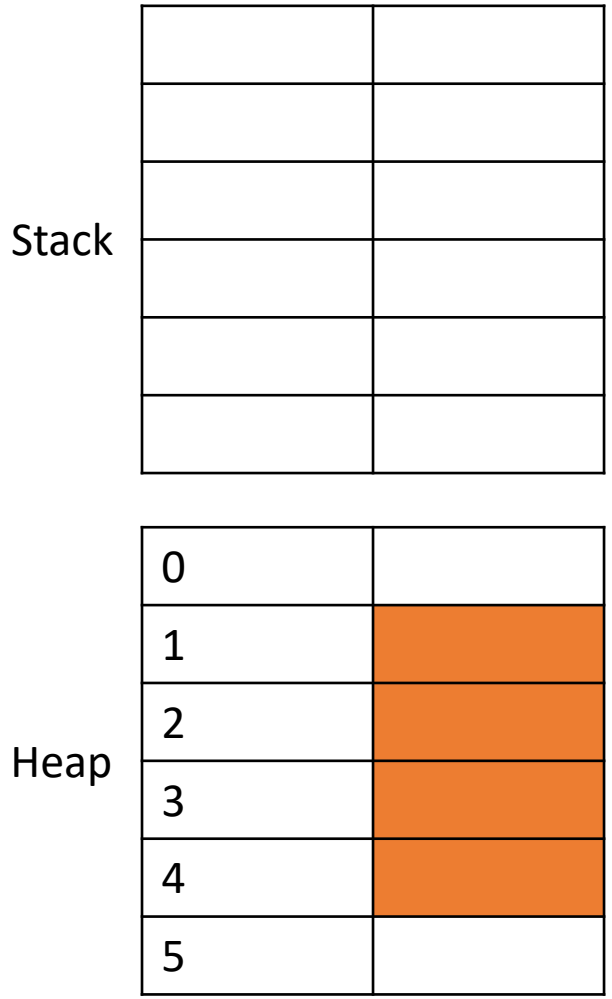
Arrays and structs are heap-allocated

```
int a[] = new(int[4]);
a[1] = 42;
a[2] = a[1] + 1;
```



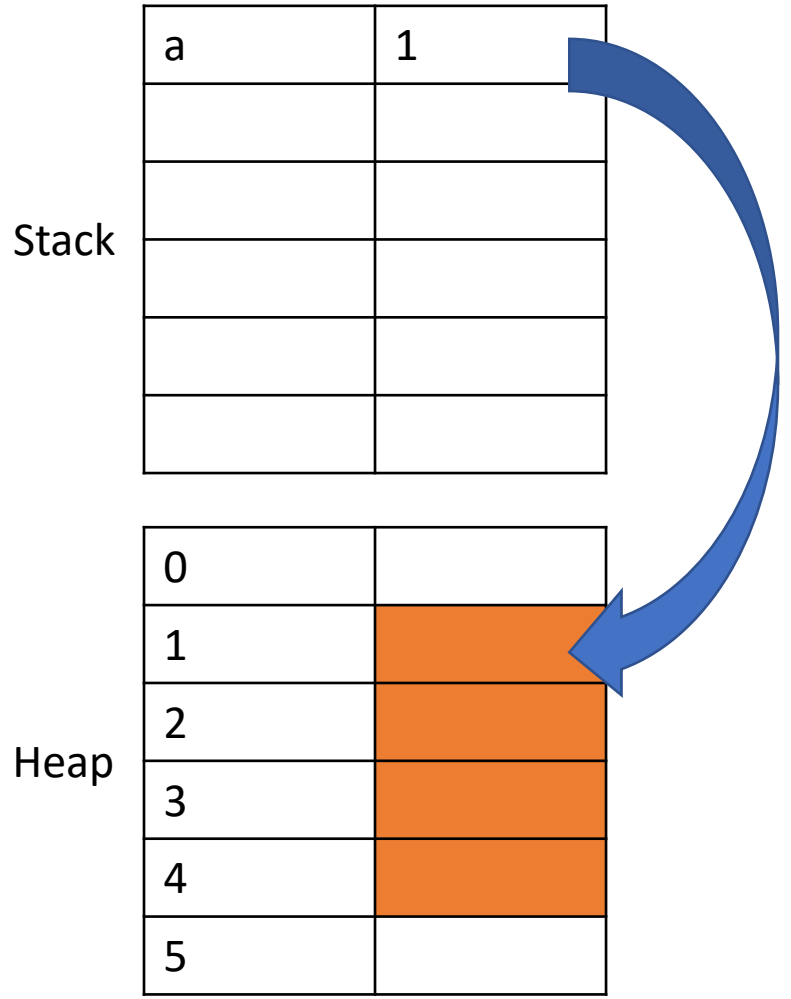
Arrays and structs are heap-allocated

```
int a[] = new(int[4]);
a[1] = 42;
a[2] = a[1] + 1;
```



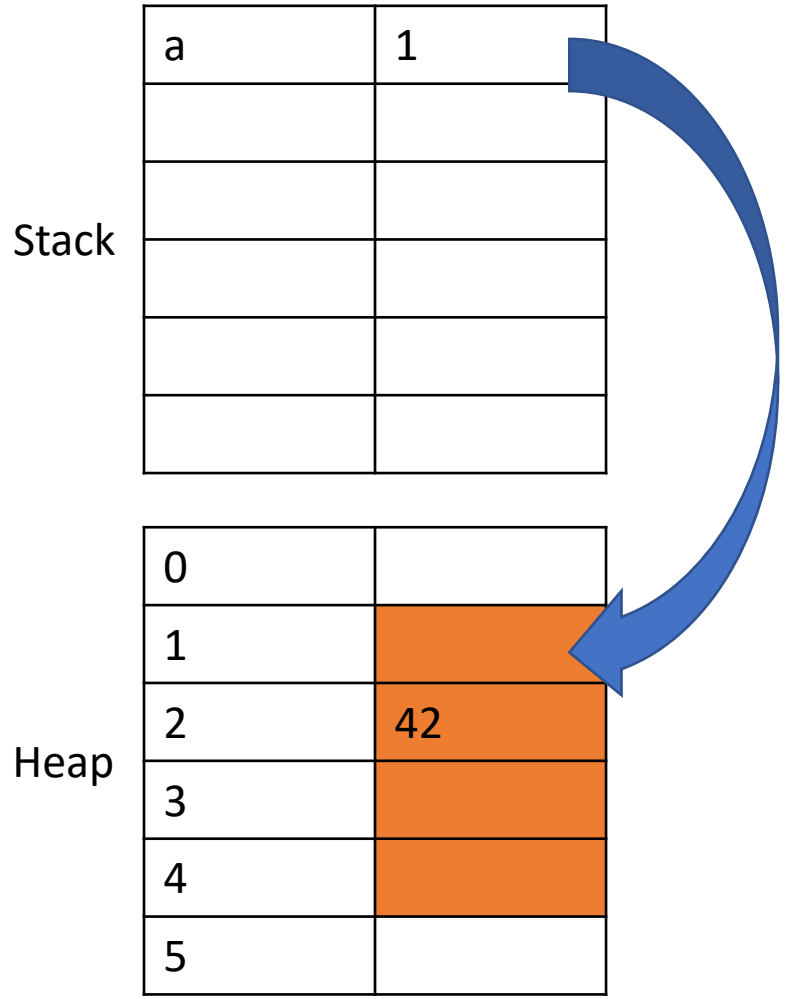
Arrays and structs are heap-allocated

```
int a[] = new(int[4]);
a[1] = 42;
a[2] = a[1] + 1;
```



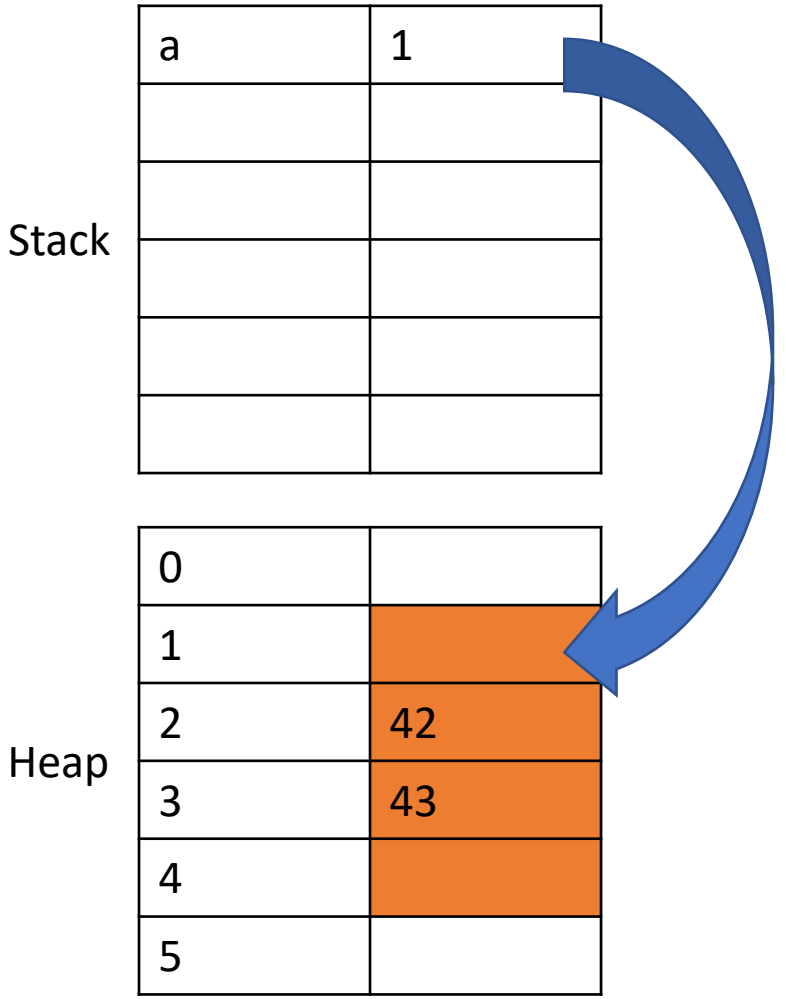
Arrays and structs are heap-allocated

```
int a[] = new(int[4]);
a[1] = 42;
a[2] = a[1] + 1;
```



Arrays and structs are heap-allocated

```
int a[] = new(int[4]);
a[1] = 42;
a[2] = a[1] + 1;
```



Not stack allocated

```
int[] init_array() {
  int a[] = new(int[4]);
  a[1] = 42;
  a[2] = a[1] + 1;
  return a;
}

int main() {
  int a[] = init_array();
  foo();
  return a[1];
}
```

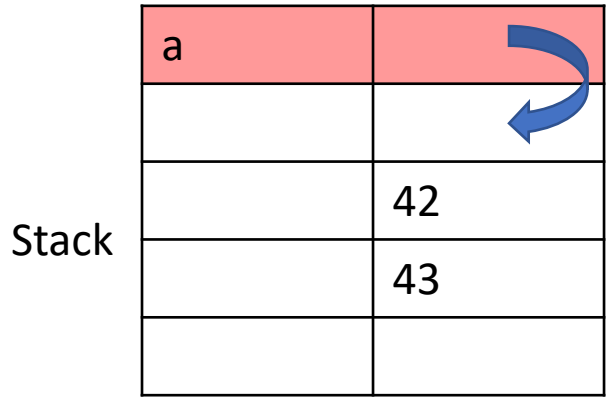
Stack

a	
a[0]	
a[1]	42
a[2]	43
a[3]	

Not stack allocated

```
int[] init_array() {
  int a[] = new(int[4]);
  a[1] = 42;
  a[2] = a[1] + 1;
  return a;
}

int main() {
  int a[] = init_array();
  foo();
  return a[1];
}
```

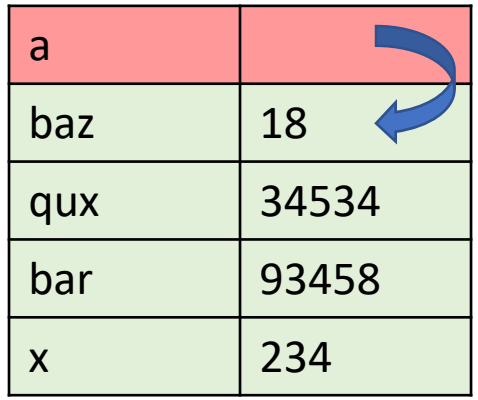


Not stack allocated

```
int[] init_array() {
  int a[] = new(int[4]);
  a[1] = 42;
  a[2] = a[1] + 1;
  return a;
}

int main() {
  int a[] = init_array();
  foo();
  return a[1];
}
```

Stack

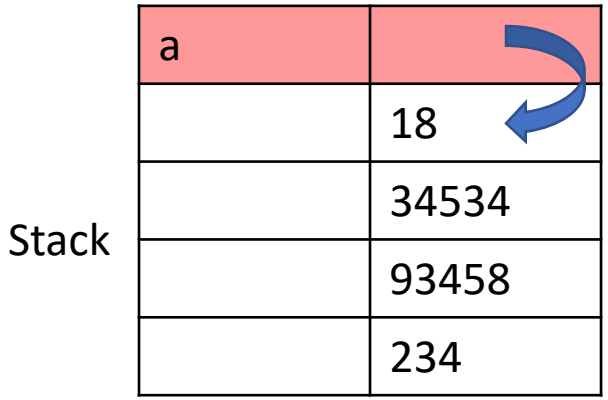


Not stack allocated

```
int[] init_array() {
  int a[] = new(int[4]);
  a[1] = 42;
  a[2] = a[1] + 1;
  return a;
}

int main() {
  int a[] = init_array();
  foo();
  return a[1];
}
```

Stack



Compiling new

```
dest = new(t);
```

```
%dest = call i8* @malloc(i32 [size of t])
```

(Will also need to bitcast %dest to whatever t* compiles to)