CS 33

Introduction to C
Part 4

Lifetime

```
int count;

int main() {
   func();
   ...
   func(); // what's printed by func?
   return 0;
}

int func() {
   int a;
   if (count == 0) a = 1;
   count = count + 1;
   printf("a = %d\n", a);
   return 0;
}
```

```
% ./a.out
-38762173
```


undefined.

Lifetime (continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
2
```


2

Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
1
```


1

Lifetime (more ...)

```
int main() {
   int *a;
   a = func();
   printf("*a = %d\n", *a); // what's printed?
   return 0;
}

int *func() {
   int x;
   x = 1;
   return &x;
}
```

```
% ./a.out
23095689
```


undefined.

Lifetime (and still more ...)

```
int main() {
   int *a;
   a = func(1);
   printf("*a = %d\n", *a); // what's printed?
   return 0;
}

int *func(int x) {
   return &x;
}
```

```
% ./a.out
98378932
```

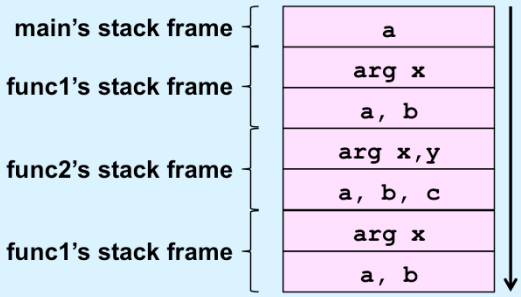

undefined.

Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the procedure
  - from call to return
  - each execution of a procedure results in a new instance of its arguments and local variables

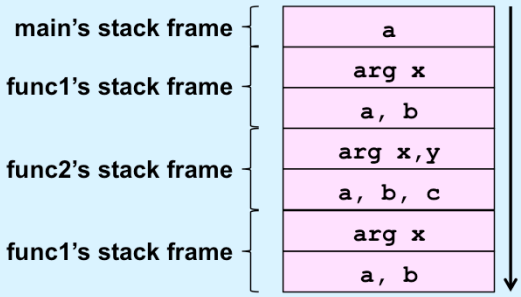
Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Quiz 1

```
void proc(int a) {
   int b=1;
   if (a == 1) {
      proc(2);
      printf("b=%d\n", b);
   } else {
      b = a*(b++)*b;
   }
}
int main() {
   proc(1);
   return 0;
}
```

- What's printed?
  - a) 0
  - b) 1
  - c) 2
  - d) 4

scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d %d", &i, &j);
}
```

Two parts

- **formatting instructions**
  - **whitespace in format string matches any amount of white space in input**
    - » **whitespace is space, tab, newline (‘\n’)**
- **arguments: must be addresses**
  - **why?**

#define (again)

```
#define CtoF(cent)  (9.0*cent)/5.0 + 32.0
```

Simple textual substitution:

```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```


Careful ...

```
#define CtoF(cent) (9.0*cent)/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;
```

```
#define CtoF(cent) (9.0*(cent))/5.0 + 32.0

float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```


Be careful with how arguments are used! Note the use of parentheses in the second version.

Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```


Pointers to Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x, *y;
x.real = 1.4;
x.imag = 3.65e-10;
y = &x;
y->real = 2.6523;
y->imag = 1.428e20;
```


Note that when we refer to members of a structure via a pointer, we use the "->" notation rather than the "." notation.

structs and Functions

```
struct ComplexNumber ComplexAdd(
          struct ComplexNumber a1,
          struct ComplexNumber a2) {
     struct ComplexNumber result;
     result.real = a1.real + a2.real;
     result.imag = a1.imag + a2.imag;
     return result;
}
```


Would This Work?

```
struct ComplexNumber *ComplexAdd(
          struct ComplexNumber *a1,
          struct ComplexNumber *a2) {
     struct ComplexNumber result;
     result.real = a1->real + a2->real;
     result.imag = a1->imag + a2->imag;
     return &result;
}
```


This doesn't work, since it returns a pointer to result, which would not be in scope once the procedure has returned. Thus the returned pointer would point to an area of memory with undefined contents.

How About This?

```
void ComplexAdd(
        struct ComplexNumber *a1,
        struct ComplexNumber *a2,
        struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```


This works fine: the caller provides the location to hold the result.

Using It …

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```

Arrays of *structs*

```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```

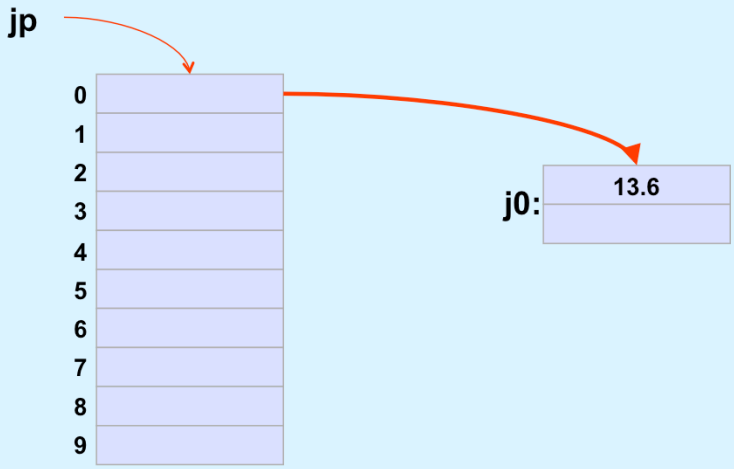
Arrays, Pointers, and *structs*

```
/* What's this? */
struct ComplexNumber *jp[10];


struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;
```


Subscripting (i.e., the "[]" operator) has a higher precedence than the "*" operator. Thus jp is an array of pointers to *struct ComplexNumber*s.

Memory View



Quiz 2

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a->val = 1;
   a->next = &b;
   b->val = 2;
   printf("%d\n", a->next->val);
   return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates

Quiz 3

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next.val);
   return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates

Quiz 4

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates

Quiz 5

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates

Structures vs. Objects

- Are structs objects?

(What's an object?)


```
for (;;)
  printf("C does not have objects!\n");
```

Structures Containing Arrays

```
struct Array {
   int A[6];
} S1, S2;

int A1[6], A2[6];

A1 = A2;
   // not legal: arrays don't know how big they are

S1 = S2;
   // legal: structures do
```


This seems pretty weird at first glance. But keep in mind that the name of an array refers to the address its first element, and does not represent the entire array. But the name of a structure refers to the entire structure.

A Bit More Syntax …

- **Constants**

```
const double pi =
  3.141592653589793238;

area = pi*r*r;     /* legal */
pi = 3.0;          /* illegal */
```


More Syntax …

```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

ptr_to_constant = &six;
   // ok
*ptr_to_constant = 7;
   // not ok
*constant_ptr = 7;
   // ok
constant_ptr = &six;
   // not ok
```


Note that constant_ptr_to_constant's value may not be changed, and the value of what it points to may not be changed.

And Still More …

- **Array initialization**

```
int FirstSixPrimes[6] = {2, 3, 5, 7, 11, 13};
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
int MagicSquare[][] = {{2, 7, 6},
                       {9, 5, 1},
                       {4, 3, 8}};
```