6.s096

Lecture 2

Today

The core of the language

- Control Structures
- Variables and Functions
- Scope
- Uninitialized Memory - and what to do about it!

Control Structures

Basic Control Structures

You’ve probably seen these…

while

do…while

for

if […else if], […else]

```
int i = 0;
while(i++ < 3){
    printf(“%d ”, i);
}

=> 1 2 3
```

Basic Control Structures

You’ve probably seen these…

while

<u>do…while</u>

for

if […else if], […else]

```
int i = 0;
do {
  printf(“%d “, i);
} while(i++ < 3);

=> 0 1 2 3
```

Basic Control Structures

You've probably seen these…

while

do…while

for

if […else if], […else]

```
// C99-style
for(int i = 0; i < 3; ++i){
    printf(“%d “, i);
}

=> 0 1 2
```

Basic Control Structures

You’ve probably seen these…

while

do…while

for

<u>if [...else if], [...else]</u>

```
int i = 0;
if(i < 3){
    printf(“It sure is.”);
} else if(i == 3){
    printf(“Nope.”);
} else {
    printf(“Still nope.”);
}
```

Slight variations

- Blocks / braces often optional (if, while, for):

```
    if(condition) expression;
```

- Empty `for` loop is an “infinite” `while`:

```
    for(;;) expression;
```

switch

-
  ```
  switch(i){
      case 1:
          printf(“It’s one!”);
          break;
      case 2:
          printf(“It’s two!!”);
          break;
      default:
          printf(“It’s something else!!!”);
  }
  ```

Jumps

```c
void foo(){
    for(int i = 0; i < 10; ++i){
        printf("%d ", i);
        if(i == 2){
            i = 3;
            continue;
        } else if(i == 6){
            break;
        }
    }

    goto end;
    printf("near the end\n");
    end:
    printf("the end\n");
    return;
    printf("or is it?");
}
```

Output:

```
0 1 2 4 5 6 the end
```

Jumps

Output:

```
0 1 2 4 5 6 the end
```

```
void foo(){
    for(int i = 0; i < 10; ++i){
        printf("%d ", i);
        if(i == 2){
            i = 3;
            continue;
        } else if(i == 6){
            break;
        }
    }

    goto end;
    printf("near the end\n");
    end:
    printf("the end\n");
    return;
    printf("or is it?");
}
```

Jumps

```c
void foo(){
    for(int i = 0; i < 10; ++i){
        printf("%d ", i);
        if(i == 2){
            i = 3;
            continue;
        } else if(i == 6){
            break;
        }
    }

    goto end;
    printf("near the end\n");
    end:
    printf("the end\n");
    return;
    printf("or is it?");
}
```

Output:

```
0 1 2 4 5 6 the end
```

Jumps

Output:

```
0 1 2 4 5 6 the end
```

```
void foo(){
    for(int i = 0; i < 10; ++i){
        printf("%d ", i);
        if(i == 2){
            i = 3;
            continue;
        } else if(i == 6){
            break;
        }
    }

    goto end;
    ~~printf("near the end\n");~~
    end:
    printf("the end\n");
    return;
    printf("or is it?");
}
```

Jumps

```
void foo(){
    for(int i = 0; i < 10; ++i){
        printf("%d ", i);
        if(i == 2){
            i = 3;
            continue;
        } else if(i == 6){
            break;
        }
    }

    goto end;
    printf("near the end\n");
    end:
    printf("the end\n");
    return;
    ~~printf("or is it?");~~
}
```

Output:

```
0 1 2 4 5 6 the end
```

```
int main(int argc, char ** argv){
    foo();
    return 0;
}
```

The **goto** statement in detail

- Syntax:
  goto ***label***;
  … where ***label*** refers to an earlier or later labelled section of code.

- Target label ***<u>must</u>*** be in the same function as the goto statement.

- Notorious for creating hard-to-read code, but the concept is critical to how computers operate.

Variables and Functions

Variables and constants

```
int a = 1;
a = 2; // cool
```

```
const int b = 1;
b = 2;
// error:
   read-only variable
   is not assignable
```

static Variables

Static variables retain their value throughout the life of the program.

```
void foo(){
    static int count = 0;
    printf("%d ", count++);
}
```

```
for(int i = 0; i < 5; ++i){
    foo();
}
```

Output: `0 1 2 3 4`

Functions in Variables

We'll examine part of this syntax in more depth in later lectures.

```
int foo(int a, int b){
    return a + b;
}

int bar(int c, int d){
    return c - d;
}
```

```
int (*func)(int, int) = &foo;
int result = func(2, 2);
printf("%d ", result); // 4

func = &bar;
result = func(2, 2);
printf("%d", result); // 0
```

Scope

Scope

A variable has a ***scope*** in which it is said to be defined.

```
void bar(){
    int a = 0;
    if(3 > 0){
        int b = 0;
        b = 2; // okay
    }
    a++; // okay
    b++; // error:
         // use of undeclared
         // identifier 'b'
}
```

```
void foo(){
    int a = 0;
}
```

In **foo** and **bar**,
a is "in scope" for
the entire function.
b is "in scope" only within
the if statement's block in **bar**.

Anonymous Blocks

Anonymous blocks demonstrate the concept of ***block scope***.

```
void foo(){
    { int a = 0; }
    {
        double a = 3.14; // no problem!

        {
            char * a = "3.14"; // no problem!
        }
    }
    // no 'a' defined in this scope
}
```

Uninitialized Memory

When you see that gibberish output…

Program memory, simplified...

```
int a = 0;
```

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

(Common)

Sources

Avoid these situations if you can help it!

- Uninitialized variables:
```
int i;
printf(“%d”, i);
```
- Out-of-bounds array access:
```
char reversed[20];
char out_of_bounds = reversed[21];
```
- Variables passed out of their defining function’s scope.
- `malloc` (coming up in a later lecture)