Week 5 Part 2

Kyle Dewey

Overview

- Scope
- Lifetime
- Testing
- Exam #1 overview

What's with the { . . . }?

Recall

- Function definitions look like this:

```
void foo() { ... }
```

- Conditionals (`if`) look like this:

```
if ( condition ) { ... }
```

- `while` loops look like this:

```
while ( condition ) { ... }
```

Brackets

- The `{ ... }` part is significant
- This is called a **block**
- Blocks have special meaning to C (and to the vast majority of languages)

Blocks

- As we’ve already seen, blocks can be nested:

```
void foo() {
  int x;
  for ( x = 0; x < 10; x++ ) {
    if ( x % 2 == 0 ) {
      printf( “Even: %i\n”, x );
      continue;
    }
    printf( “Odd: %i\n”, x );
  }
}
```

Blocks

- Importance of this lies in variable declaration
- A block nested at level N has access to variables defined at nesting levels 0 .. N - 1, but not the other way around

Example

```
void foo() {
  int x = 10;
  if ( x > 5 ) {
    int y = x * 4;
    // this block can access x
  }
  // ...but this block can’t access y
}
```

So what?

- This may seem obvious and/or insignificant
- This mechanism means that you don’t have to worry about what was defined in inner blocks, because they are inaccessible anyway

Variable Name Reusage

- Blocks help to prevent variable names from clashing
- A variable `foo` defined in a given block is distinct from all other variables named `foo` defined in other blocks

Example

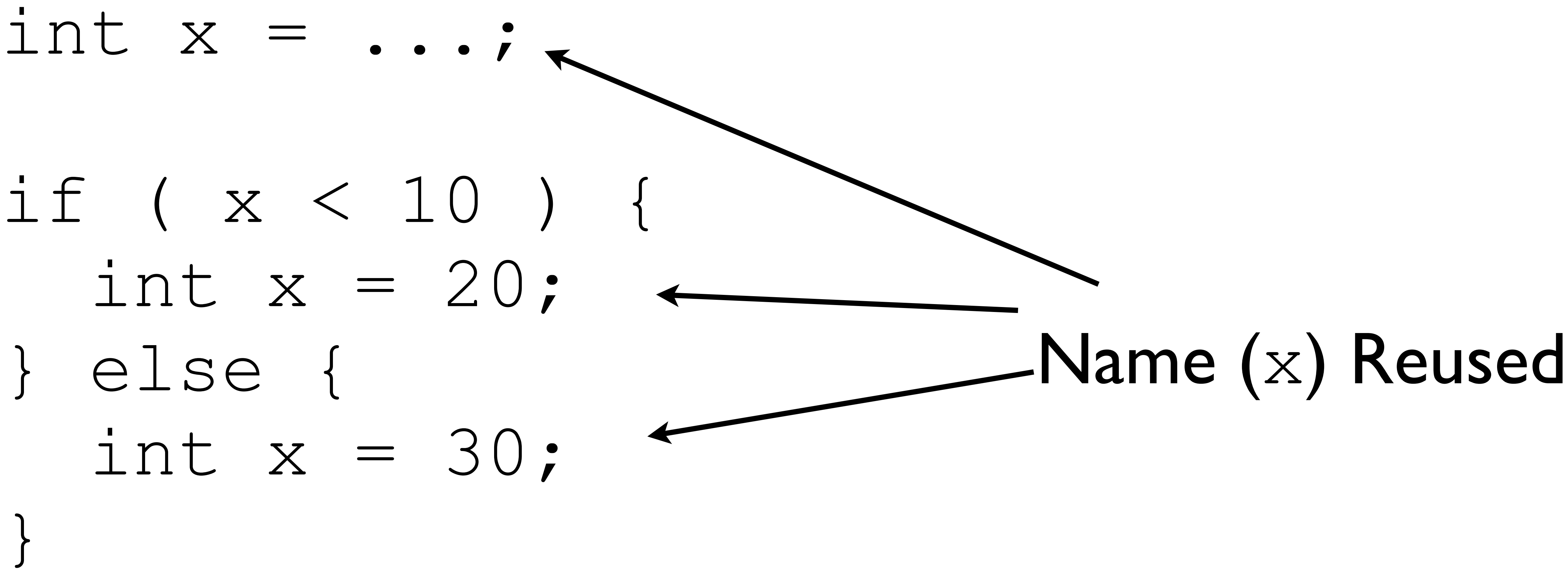
```
int x = ...;

if ( x < 10 ) {
  int y = 20;
} else {
  int y = 30;
}
```

Distinct variables

Variable Name Reusage

- Consider the following code:



```
int x = ...;

if ( x < 10 ) {
  int x = 20;
} else {
  int x = 30;
}
```

Name (x) Reused

Variable Name Reusage

- The original variable x **does not change**
- The old definition is **shadowed** by the new one, **not** overwritten

```
int x = ...;

if ( x < 10 ) {
  int x = 20;
} else {
  int x = 30;
}
```

Question

- What does this code print?

```
int x = 10;
if ( x == 10 ) {
  int x = 5;
  printf( “%i\n”, x );
}
printf( “%i\n”, x );
```

Block Advantage

- Focus only on one block at a time, not on previous blocks
    - Variables defined in previous blocks are shadowed

Scope

- **Scope** defines which variables can be accessed at any given point in the code
- Blocks manipulate the scope

```
if ( 1 < 2 ) {
  int x = 5;
  // x is now in scope
}
// x is no longer in scope
```

Scope Example

```
int x = 10;
// x is now in scope

if ( 1 < 2 ) {
  int x = 5;
  // x is in scope, but it
  // refers to the x = 5 definition
}

// x is in scope, but it refers to
// the x = 10 definition
```

Lifetime

- How long a variable exists in your program is the variable's **lifetime**
- Scope is **not** the same as lifetime
    - Scope: when you can access a variable
    - Lifetime: whether or not a variable is there

Scope vs. Lifetime

- A variable in scope is necessarily alive
- A variable that’s alive is not necessarily in scope

Example #1

```
int x = ...; // alive and in scope
if ( x < 10 ) {
  int y = 5; // alive and in scope
  ...
}
// y is not in scope and not alive
```

Example #2

```
int x = ...; // alive and in scope
if ( x < 10 ) {
  int y = 5; // x and y are alive
             // and in scope
  if ( x < y + 5 ) {
    int z = 20;
    // x, y, z alive and in scope
  }
  // x, y alive and in scope
}
// x alive and in scope
```

Example #3

```
int x = ...; // alive and in scope
if ( x < 10 ) {
  int x = 5; // x = ... is alive
             // but not in scope
             // x = 5 alive in scope
  if ( x < y + 5 ) {
    int x = 20;
    // x = ... and x = 5 alive
    // only x = 20 is in scope
  }
  // x = ... and x = 5 alive
  // only x = 5 in scope
}
// x = ... alive and in scope
```

Example #4

```
void bar() {
  // y is alive but not in scope
  int z = 5;
}

void foo() {
  int y = 10;
  bar();
}

void main() {
  foo();
}
```

Global Variables

- Consider the following code:

```
int x = 10;

void foobar() {
  printf( “%i\n”, x );
}

void barfoo() {
  x++;
}
```

Global Variables

- `x` is a global variable
- Always in scope (unless shadowed)
- Always alive

```
int x = 10;

void foobar() {
  printf( “%i\n”, x );
}

void barfoo() {
  x++;
}
```

Thought Question

- Global variables are seen as bad practice, and are usually avoided
- Why?

Answer

- Always in scope and always alive means everything in the file probably heavily relies on it
  - Another variable to keep track of for **everything** in the file
  - Can be error prone
  - Interdependent code

Aside: "In the File"

- Technically a "compilation unit"
- In this class, a file is a compilation unit
- However, it's possible to have multiple files in the same compilation unit

Testing

Recall...

- Testing is an important step in software development
  - Builds confidence that code works correctly
  - Modern software development heavily relies on testing

Testing

- Testing can confirm a bug exists
- ...but it cannot confirm that bugs do not exist
  - May not be testing for it
  - May need additional tests

Testing Weakness

```
int badMax( int x, int y ) {
  if ( x == 513 ) {
    return x;
  } else if ( x > y ) {
    return x;
  } else {
    return y;
  }
}
```

Testing Strength

- Code is not usually written like that
    - The goal is not to mess up the tests
- Simple (compared to **verification**, which attempts to prove that there are no bugs)

Additional Terminology

- White box testing: you can see the whole code, as with:

```
// get the max of x and y
int max( int x, int y ) {
  if ( x > y ) {
    return x;
  } else {
    return y;
  }
}
```

Additional Terminology

- Black box testing: you can see only the interfaces and what they do, as with:

```
// get the max of x and y
int max( int x, int y );
```

Exam #1 Overview