CSCI 135
Scopes

Variable Naming

Consider: Real programs have thousands (or billions!) of lines of code, written by large teams of programmers.

? How do we prevent them from using the same location in memory?

Ex: Alice and Bob use the same variable name, age, to store ages of two different entities (or worse, both of them use a meaningless variable name like n or x).

! Decompose program into multiple **scopes** and require all variable references to refer to the version of the variable in scope.

Program Block

Def: Each compound statement is called a **block**, and each block is associated with a **scope**.

```
int y = 0;
int x = 5;
if (...)  {
    int x = 6;              Inner Scope
    y = x;
};
y = x;
```

- Two *distinct* variables named x, one for inner scope, and one outside it. Instances of x are stored in different memory locations and are NOT the same variable!
- Inside the inner scope, any references to x are to the inner scope's x.
- Any references to y are to the outer scope's y.

? What is y at end of inner scope above?

Program Block

Def: Each compound statement is called a **block**, and each block is associated with a **scope**.

```
int y = 0;
int x = 5;
if (...)  {
    int x = 6;                    Inner Scope
    y = x;
};
y = x;
```

- Two *distinct* variables named x, one for inner scope, and one outside it. Instances of x are stored in different memory locations and are NOT the same variable!
- Inside the inner scope, any references to x are to the inner scope's x.
- Any references to y are to the outer scope's y.

? What is y at end of inner scope above? ! 6

Program Block

Def: Each compound statement is called a **block**, and each block is associated with a **scope**.

```
int y = 0;
int x = 5;
if (...)  {
     int x = 6;                Inner Scope
     y = x;
};
y = x;
```

- Two *distinct* variables named x, one for inner scope, and one outside it. Instances of x are stored in different memory locations and are NOT the same variable!
- Inside the inner scope, any references to x are to the inner scope's x.
- Any references to y are to the outer scope's y.

? What is y at end of program?

Program Block

Def: Each compound statement is called a **block**, and each block is associated with a **scope**.

```
int y = 0;
int x = 5;
if (...)  {
    int x = 6;              Inner Scope
    y = x;
};
y = x;
```

- Two *distinct* variables named x, one for inner scope, and one outside it. Instances of x are stored in different memory locations and are NOT the same variable!
- Inside the inner scope, any references to x are to the inner scope's x.
- Any references to y are to the outer scope's y.

? What is y at end of program? ! 5

Scope

- Every variable is associated with a scope.
- A variable is **visible** from the point of its declaration to the end of its scope (including any nested scopes that don't define a variable of the same name).
- Scopes may be (and often are) nested.
- For any name, only one variable of that name may be defined in a scope.
- Statements have no way of accessing variables outside their scope.

⚠Just because you can have multiple variables with the same name in a small program doesn't mean you should. Readability matters!
BUT you shouldn't think about other blocks/functions (thus, scopes) when writing your block/function.

Some Scopes

Which constructs introduce a new scope?

- Global scope: all variables declared outside main() (we will talk about later – AVOID).
- Every program block (typically indicated by {})
- Main: all variables declared in main() (and outside other scopes)
- Every function

These are often [imprecisely] referred to as local or global scopes depending on whether they are declared inside a block or outside `main`.

Where Should I Declare a Variable?

General guideline: Declare a variable at top of the innermost scope where it is used.
Ex:

```
main() {
  int x;
  ...
  if (...) {
    int temp;          Declared here since it is
    temp = ...;        not used outside block
    x = temp*2;
    };
  ...                  temp is invisible here
}
```

⚠K&R C only supported variable declarations at top of main (and functions), and some programmers still follow old rules. You might maintain such code!

Scopes: An Exception

```
main() {
   ...
   for (int n=0; n<32; n++) {   declares n, initialized to 0
      ...
      };
   ...
};
```

? What is the scope of n?

Scopes: An Exception

```
main() {
   ...
   for (int n=0; n<32; n++) {   declares n, initialized to 0
      ...
      };
   ...
};
```

? What is the scope of n?

! ANSI C: any variables declared in a for loop initializer have scope local to the loop body.
BUT some older compilers don't follow this rule (*i.e.*, it might not be portable).

Example: Scopes Inside Loops

```
main() {
  while (...) {
    int x;      Scope is block
    x++;
    cout << "x is: " << x << endl;
    };
  }
```

Assume: all variables are initialized to 0 on your system (this is hypothetical, and NOT true in general).
Each declaration of x results in creation of a new variable (*i.e.*, possibly at a different memory location) named x.
⇒ The program creates one x per iteration, and repetitively prints

```
x is:  1
```

BUT, the value of x from a previous iteration is not accessible.

Scope vs. Lifetime

Careful: Don't confuse scope and lifetime!

The **scope** of a variable is the part of the program where the variable is visible (and can be accessed).

The **lifetime** of a variable is the time duration (with respect to program) in which the variable exists in memory.

Previous example (if lifetime of x extended outside the scope):
`x is: 1`, `x is: 2`, `x is: 3`, ...

So far: The lifetime of every variable is its scope

Future: Possible to specify (using storage classes) that a variable lives longer than its scope (*e.g.*, if x in next iteration of previous example is stored in the same location).

Exercise

Consider the code:

```
int main() {
  int n=2; int m=1;
  if (1==1) {
    int n;
    n = foo(m,n);
    return(n)}
int foo(int m, int n) {
  n++; m++;
  return (m+n)}
```

1. What are m and n after the above code executes (assuming you've added appropriate prototypes, headers, etc.)?
2. Repeat if foo's second argument is passed by reference.
3. Repeat if foo's first argument is passed by reference.
4. Identify those positions in the program where you can modify the value of the n declared at the top of `main`.