

Storage and Scope

C Programming and Software Tools

N.C. State Department of Computer Science

Computer Science
NC STATE UNIVERSITY

Blocks

- A *block* is a set of statements delimited by curly braces: { }
- i.e., body of a loop, function definition, or anywhere you want to define a new *scope*, for example...

```
int sum = 0;
int main (void)
{
    int i;
    for ( i=0 ; i<n ; i++ ) {
        ...
    }
    if (n > 60) {
        ...
    }
    ...
}
```

CSC230: C and Software Tools © 1

Computer Science
2 NC STATE UNIVERSITY

Scope of Variables

- (for the moment, this discussion is only for programs whose source code is contained entirely in *one file*)

A variable defined or declared **outside of any block** has **global scope**

- the variable is **visible** (read/writable) to all functions that appear **after** it in the source file

```
int main
(void)
{ ... }

int i = 15;

int f(...)
{...}

int g(...)
{...}
```

CSC230: C and Software Tools © NC State University Computer Science Faculty

3

Science
NC STATE UNIVERSITY

Scope... (cont'd)

- A variable defined **inside a block** has scope only **within** that block
- Variables with different scopes (even if they have the same name) are **independent** variables
- If two or more variables have the same name, to resolve a variable reference the rule is:

“most local scope wins”

CSC230: C and Software Tools © NC State University Computer Science Faculty

Computer Science
4 NC STATE UNIVERSITY

Example

```
...  
char c = 'a';  
int i = 15;  
int j = 0;  
  
int f(void)  
{  
    char c = 'b';  
    int i = 25;  
    int sum = 0;  
  
    for (int k = 1; k < 4; k++)  
        sum += k;  
    (void) printf("c=%c, i=%d, sum=%d\n", c, i, sum);  
    {  
        int i = 35;  
        j = i + 13;  
        (void) printf("c=%c, i=%d, j=%d\n", c, i--, j);  
    }  
    (void) printf("c=%c, i=%d, j=%d\n", c, i, j);  
    ...  
}
```

Diagram illustrating variable scope and shadowing:

- Global scope: `char c = 'a';`, `int i = 15;`, `int j = 0;`
- Function scope `f(void)`:
 - Local scope: `char c = 'b';`, `int i = 25;`, `int sum = 0;`
 - Loop scope: `for (int k = 1; k < 4; k++)`
 - Block scope: `{ int i = 35; j = i + 13; }`

Arrows indicate the flow of variable resolution and shadowing:

- From `int i = 15;` to the `i` in `printf("c=%c, i=%d, sum=%d\n", c, i, sum);`
- From `int i = 25;` to the `i` in `printf("c=%c, i=%d, sum=%d\n", c, i, sum);`
- From `int i = 35;` to the `i` in `printf("c=%c, i=%d, j=%d\n", c, i--, j);`
- From `int i = 35;` to the `i` in `j = i + 13;`
- From `int i = 35;` to the `i` in `printf("c=%c, i=%d, j=%d\n", c, i, j);`

CSC230: C and Software Tools © NC State University Computer Science Faculty

Scope... (cont'd)

- Is it a good idea to avoid reusing the same variable name?

⚠ common source of bugs ⚠
**confusion about scoping,
and use of common names**

Special Case: Scope of Labels

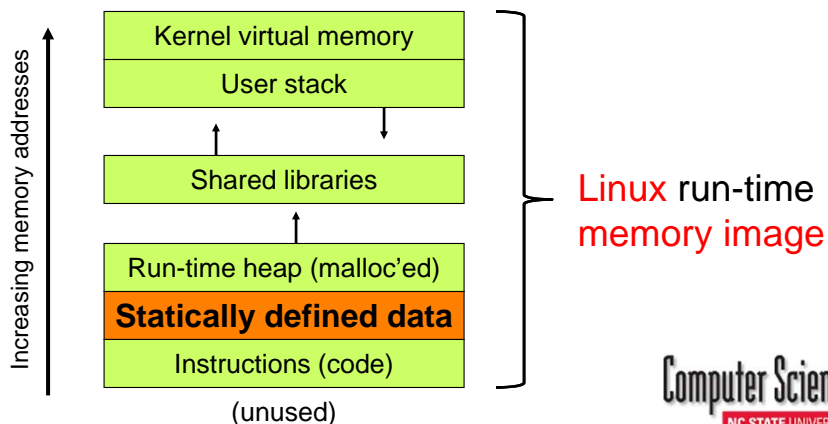
- The scope of **(goto) labels** is just the function they are contained in
 - so: you cannot **goto** a label defined in another function

```
int f1(int b) {  
    if (b < 10)  
        goto labelx;  
    (void) printf("b < 10\n");  
labelx:  
    return 0;  
}  
  
int f2(int a) {  
    if (a > 5)  
        goto labelx;  
    (void) printf("a > 5\n");  
labelx:  
    return 0;  
}
```

CSC230: C and Software Tools © NC State University Computer Science Faculty

Lifetime (Storage Class) of Variables

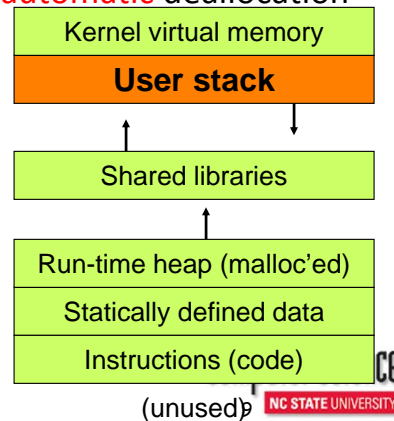
- Memory space for a **global** variable is **statically-allocated** at compile and load time
 - this is called the **static** storage class



CSC230: C and Software Tools © NC State University Computer Science Faculty

Lifetime... (cont'd)

- Memory space for a variable declared inside a block is **automatically-allocated** at run time
 - entry into the block triggers **automatic** memory allocation and exit triggers **automatic** deallocation
 - this is called the **auto storage class**



CSC230: C and Software Tools © NC State University Computer Science Faculty

```
...
char c = 'a';
int i = 15;
int j = 0;

int main (void)
{
    char c = 'b';
    int i = 25;
    int sum = 0;
    for (int k = 1; k < 4; k++)
        sum += k;
    (void) printf("c=%c, i=%d, sum=%d\n", c, i, sum);
    {
        int i = 35;
        j = i + 13;
        (void) printf("c=%c, i=%d, j=%d\n", c, i--, j);
    }
    (void) printf("c=%c, i=%d, j=%d\n", c, i, j);
    ...
}
```

For each i variable, how is the memory allocated (statically or dynamically) and what is the storage class?

Lifetime... (cont'd)

- In C, you can manually force variables declared inside a block to be **static** storage class using the **static** keyword
 - memory space is allocated only **once**, in the Static area of memory

```
...
char c = 'a';
int i = 15;
int j = 0;

int f(void)
{
    char c = 'b';
    int i = 25;
    int sum = 0;
    for (int k = 1; k < 4; k++)
        sum += k;
    (void) printf("c=%c, i=%d, sum=%d\n", c, i, sum);
    {
        static int i = 35;
        j = i + 13;
        (void) printf("c=%c, i=%d, j=%d\n", c, i--, j);
    }
    (void) printf("c=%c, i=%d, j=%d\n", c, i, j);
    ...
}
```

Initialization of Variables

- **static**, **auto**, ... - what does it matter?
- 1. Space on the stack is **limited** (remember problems allocating **bigarray[]** as an **auto** variable?)
- 2. **static** variables with global scope can be initialized only with **constant expressions**

```
char c = 'a';  
int i = 15 + (39 % 3);  
int j = 0;
```

Initialization ... (cont'd)

- **auto** class variables can be initialized using **any valid expression** at the point at which they are declared

```
{...  
    int i = 15 + (39 % 3) + f();  
    int j = getchar() * 6 + i;  
...}
```

- 3. **default** value of static variables (if not explicitly initialized) is **0**
 - There is **no default initial value** for auto variables

```
{ int i; static int j; Output?  
  printf("%d %d\n", i, j); }
```

⚠ common source of bugs ⚠
**failure to explicitly
initialize variables**

Initialization ... (cont'd)

- 4. Static class variables are **initialized only once!**

auto variables are (re-)initialized every time the block is entered

⚠ common source of bugs ⚠
**expecting static
behavior from
auto variables**

Initialization Example

```
void f(void);

main(void) {
    ...
    f();
    f();
    f();
}

void f( void ) {
    int k = 0;
    static int j = 0;
    printf ( " %d %d\n", ++j, ++k );
}
```

Output?

The **register** Storage Class

- A **recommendation** to the compiler to consider storing the variable in a register instead of memory

```
int main (void)
{
    ...
    int i = 25;
    double sum = 0.0;
    for ( register int k = 1; k < 1000000; k++)
        sum += k;
    ...
}
```

What difference does that make?

The **register** ... (cont'd)

- Can only be specified for **auto** variables (i.e., **not** for **global** variables)
 - some restrictions on what types of variables can be specified as **register** class (*see reference manual for details*)
- **Optimizing compilers** may be able to do a better job than you can at identifying candidates for register storage

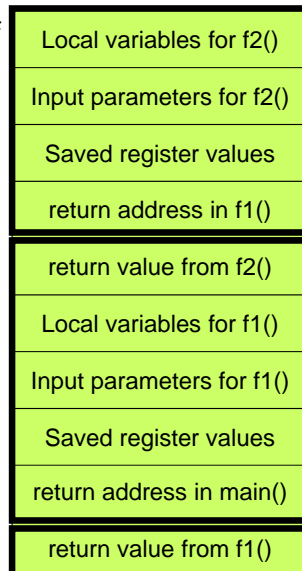
The Stack

- Area of **memory** reserved for dynamic allocation of auto variables, and **function parameters**
 - input parameter values
 - return address of caller
 - auto variables local to the block
 - saved register values
 - result returned by function
- All of this is called the **activation record** or **stack frame** of the called function (i.e., the callee)

Example of Stack Contents

- Top of stack

Increasing memory addresses
↓



Activation record for function f2 ()

Operations on stack are **LIFO** (last-in, first-out): **push** onto stack and **pop** from stack

Activation record for function f1 ()

Another Storage Class: **extern**

- *discussed later along with linking...*

★ you try it ★
Storage

Exercise

What is printed...?

Put a * if it could
be anything.

How is the memory
allocated
(statically or
dynamically) and
what is the
storage class for
the instructor
selected variable?

```
int x = 15, y = 25;
int main (void) {
    printf("%d\n", x); //1
    int x;
    printf("%d\n", x); //2
    x = y;
    printf("%d\n", y); //3
    for (int j = 0; j < 3; j++) {
        int y = 32;
        static int x = 35;
        x = 2 * x;
        printf("%d\n", x); //4
        y = y / 2;
        printf("%d\n", y); //5
    }
    int y = 100;
    printf("%d\n", y); //6
}
```