



A Fast Review of C (part 2)

Storage Classes



Outline

- Operators
 - expressions, precedence, associativity
- Control flow
 - if, nested if, switch
 - Looping



Operators


Expressions

- Constant expressions
 - 5
 - $5 + 6 * 13 / 3.0$
- Integral expressions (**int j,k**)
 - j
 - $j / k * 3$
 - $k - 'a'$
 - $3 + (\text{int}) 5.0$
- Float expressions (**double x,y**)
 - $x / y * 5$
 - $3 + (\text{float}) 4$
- Pointer expressions (**int * p**)
 - p
 - $p + 1$
 - "abc"


Precedence & associativity

- All operators have two important properties called **precedence** and **associativity**.
 - Both properties affect how operands are attached to operators
 - Operators with higher precedence have their operands bound, or grouped, to them before operators of lower precedence, regardless of the order in which they appear.
 - In cases where operators have the same precedence, associativity (sometimes called binding) is used to determine the order in which operands grouped with operators.
- $2 + 3 * 4$
 - $3 * 4 + 2$
 - $a + b - c;$
 - $a = b = c;$
 - `/* assign c to b, then
assign b to a */`

Precedence & associativity (page 120 – Table 5.1)

Class of operator	Operators in that class	Associativity	Precedence
primary	() [] -> .	Left-to-Right	 HIGHEST
unary	cast operator sizeof & (address of) * (dereference) - + ~ ++ -- !	Right-to-Left	
multiplicative	* / %	Left-to-Right	
additive	+ -	Left-to-Right	
shift	<< >>	Left-to-Right	
relational	< <= > >=	Left-to-Right	
equality	== !=	Left-to-Right	

Precedence & associativity (page 120 – Table 5.1)

Class of operator	Operators in that class	Associativity	Precedence
bitwise AND	&	Left-to-Right	
bitwise exclusive OR	^	Left-to-Right	
bitwise inclusive OR		Left-to-Right	
logical AND	&&	Left-to-Right	
logical OR		Left-to-Right	
conditional	? :	Right-to-Left	
assignment	= += -= *= /= %= >>= <<= &= ^=	Right-to-Left	
comma	,	Left-to-Right	LOWEST

Parenthesis

- The compiler groups operands and operators that appear within the parentheses first, so you can use parentheses to specify a particular grouping order.
 - $(2 - 3) * 4$
 - $2 - (3 * 4)$

- The inner most parentheses are evaluated first. The expression $(3+1)$ and $(8-4)$ are at the same depth, so they can be evaluated in either order.

$$1 + ((3+1) / (8 - 4) - 5)$$

$$1 + (4 / (8 - 4) - 5)$$

$$1 + (4 / 4 - 5)$$

$$1 + (1 - 5)$$

$$1 + -4$$

$$-3$$

Binary arithmetic operators

Operator	Symbol	Form	Operation
multiplication	*	$x * y$	x times y
division	/	x / y	x divided by y
remainder	%	$x \% y$	remainder of x divided by y
addition	+	$x + y$	x plus y
subtraction	-	$x - y$	x minus y



The remainder operator

- Unlike other arithmetic operators, which accept both integer and floating point operands, the remainder operator accepts only integer operands!
- If either operand is negative, the remainder can be negative or positive, depending on the implementation
- The ANSI standard requires the following relationship to exist between the remainder and division operators
 - ***a*** equals ***a*%*b*** + (***a*/*b***) * ***b*** for any integral values of ***a*** and ***b***

Arithmetic assignment operators

Operator	Symbol	Form	Operation
assign	=	a = b	put the value of b into a
add-assign	+=	a += b	put the value of a+b into a
subtract-assign	-=	a -= b	put the value of a-b into a
multiply-assign	*=	a *= b	put the value of a*b into a
divide-assign	/=	a /= b	put the value of a/b into a
remainder-assign	%=	a %= b	put the value of a%b into a

Arithmetic assignment operators

```
int m = 3, n = 4;
```

```
float x = 2.5, y = 1.0;
```

```
m += n + x - y
```

```
m /= x * n + y
```

```
n %= y + m
```

```
x += y -= m
```

Equivalent expressions:

```
m = (m + ((n+x) -y ))
```

```
m = (m / ((x*n) + y ))
```

```
n = (n % (y + m) )
```

```
x = ( x + ( y = (y - m )))
```

Increment & decrement operators

Operator	Symbol	Form	Operation
postfix increment	++	a++	get value of a, then increment a
postfix decrement	--	a--	get value of a, then decrement a
prefix increment	++	++a	increment a, then get value of a
prefix decrement	--	--a	decrement a, then get value of a

Increment & decrement operators

```
main () {  
    int j=5, k=5;  
    printf("j: %d\t k : %d\n", j++, k--);  
    printf("j: %d\t k : %d\n", j, k);  
    return 0;  
}
```

j: 5	k: 5
j: 6	k: 4

```
main () {  
    int j=5, k=5;  
    printf("j: %d\t k : %d\n", ++j, --k);  
    printf("j: %d\t k : %d\n", j, k);  
    return 0;  
}
```

j: 6	k: 4
j: 6	k: 4

Increment & decrement operators

`int j = 0, m = 1, n = -1;`

Equivalent expression:

`m++ - --j`

$(m++) - (--j)$
(2)

`m += ++j * 2`

$m = (m + ((++j) * 2))$
(3)

`m++ * m++`

$(m++) * (m++)$
(implementation-dependent, in cc: '2')

Comma operator

- Comma operator allows you to evaluate two or more distinct expressions wherever a single expression allowed!
 - The result is the value of the rightmost operand
 - Usage: a, b; Operation: evaluate a, evaluate b, return b.
 - Though using the “,” is legal, it may lead to confusing code, it's mostly used in the first and last expressions in a **for** loop:
 - for (j = 0, k = 100; k - j > 0; j++, k--) which is same with:
 - j=0; k=100;
 - while (k-j <0){
 - ...
 - j++; k--;
 - }
 - How many times it will loop?

Relational operators

Operator	Symbol	Form	Result
greater than	>	a > b	1 if a is greater than b; else 0
less than	<	a < b	1 if a is less than b; else 0
greater than or equal to	>=	a >= b	1 if a is greater than or equal to b; else 0
less than or equal to	<=	a <= b	1 if a is less than or equal to b; else 0
equal to	==	a == b	1 if a is equal to b; else 0
not equal to	!=	a != b	1 if a is NOT equal to b; else 0

Relational operators

```
int j=0, m=1, n=-1;  
float x=2.5, y=0.0;
```

$j > m$

$m/n < x$

$j \leq m \geq n$

$++j == m != y * 2$

$j > m$

(0)

$(m / n) < x$

(1)

$((j \leq m) \geq n)$

(1)

$((++j) == m) != (y * 2)$

(1)

Logical operators

Operator	Symbol	Form	Result
logical AND	&&	a && b	1 if a and b are non zero; else 0
logical OR	 	a b	1 if a or b is non zero; else 0
logical negation	!	!a	1 if a is zero; else 0

Logical operators

```
int j=0, m=1, n=-1;
```

```
float x=2.5, y=0.0;
```

```
j && m
```

```
j < m && n < m
```

```
x * 5 && 5 || m / n
```

```
!x || !n || m + n
```

```
(j) && (m)
```

```
(0)
```

```
(j < m) && (n < m)
```

```
(1)
```

```
((x * 5) && 5) || (m / n)
```

```
(1)
```

```
((!x) || (!n) || (m + n))
```

```
(0)
```

Bit manipulation operators

Operator	Symbol	Form	Result
right shift	<code>>></code>	<code>x >> y</code>	x shifted right by y bits
left shift	<code><<</code>	<code>x << y</code>	x shifted left by y bits
bitwise AND	<code>&</code>	<code>x & y</code>	x bitwise ANDed with y
bitwise inclusive OR	<code> </code>	<code>x y</code>	x bitwise ORed with y
bitwise exclusive OR (XOR)	<code>^</code>	<code>x ^ y</code>	x bitwise XORed with y
bitwise complement	<code>~</code>	<code>~x</code>	bitwise complement of x

Bit manipulation operators

Expression	Binary model of Left Operand	Binary model of the result	Result value
$5 \ll 1$	00000000 00000101	00000000 00001010	10
$255 \gg 3$	00000000 11111111	00000000 00011111	31
$8 \ll 10$	00000000 00001000	00100000 00000000	2^{13}
$1 \ll 15$	00000000 00000001	10000000 00000000	-2^{15}

Expression	Binary model of Left Operand	Binary model of the result	Result value
$-5 \gg 2$	11111111 11111011	00111111 11111110	$2^{13} - 1$
$-5 \gg 2$	11111111 11111011	11111111 11111110	-2

Shifting negative numbers: some implementations fill the vacant bits with 0s, while others fill them with 1s.

Bit manipulation operators

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 & 5722	0x0452	00000100 01010010

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 5722	0x36DE	00110110 11011110

Bit manipulation operators

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
5722	0x165A	00010110 01011010
9430 ^ 5722	0x328C	00110010 10001100

Expression	Hexadecimal Value	Binary representation
9430	0x24D6	00100100 11010110
~9430	0xDB29	11011011 00101001

Bitwise assignment operators

Operator	Symbol	Form	Result
right-shift-assign	>>=	a >>= b	Assign $a \gg b$ to a.
left-shift-assign	<<=	a <<= b	Assign $a \ll b$ to a.
AND-assign	&=	a &= b	Assign $a \& b$ to a.
OR-assign	 =	a = b	Assign $a b$ to a.
XOR-assign	^=	a ^= b	Assign $a \wedge b$ to a.

Cast & sizeof operators

- Cast operator enables you to convert a value to a different type
- One of the use cases of cast is to promote an integer to a floating point number of ensure that the result of a division operation is not truncated.
 - $3 / 2$
 - `(float) 3 / 2`
- The **sizeof** operator accepts two types of operands: an expression or a data type
- **sizeof** returns the number of bytes that operand occupies in memory
 - `sizeof (3+5)` returns size of int
 - `sizeof(short)`
- The result type of sizeof can be: *int, unsigned int, unsigned long=>* depending on your compiler (on my compiler its *unsigned long*).

Conditional operator (? :)

Operator	Symbol	Form	Operation
conditional	? :	a ? b : c	if a is nonzero result is b; otherwise result is c

- The conditional operator is the only ternary operator.
- It is really just a shorthand for a common type of **if...else** branch

z = ((x<y) ? x : y);

```
if (x<y)
    z = x;
else
    z = y;
```

Memory operators

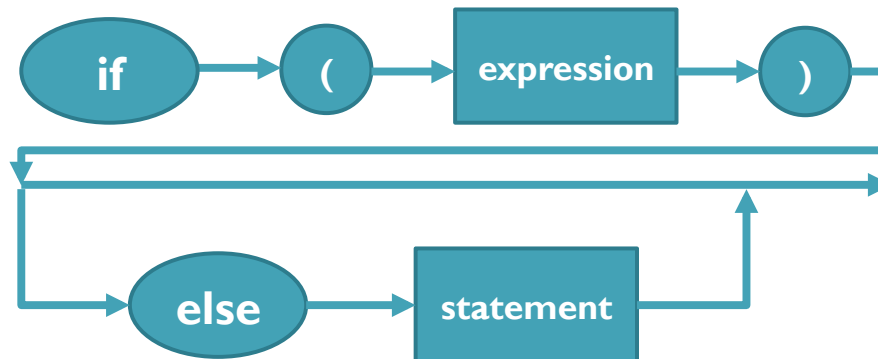
Operator	Symbol	Form	Operation
address of	&	&x	Get the address of x.
dereference	*	*a	Get the value of the object stored at address a.
array elements	[]	x[5]	Get the value of array element 5.
dot	.	x.y	Get the value of member y in structure x.
right-arrow	->	p -> y	Get the value of member y in the structure pointed to by p



Control flow

- Conditional branching
 - if, nested IF
 - switch
- Looping
 - for
 - while
 - do...while

IF...else statement



Ex1 :

```
if (x)
    statement1; // executed only if x is non-0
statement2;    //always executed
```

Ex2:

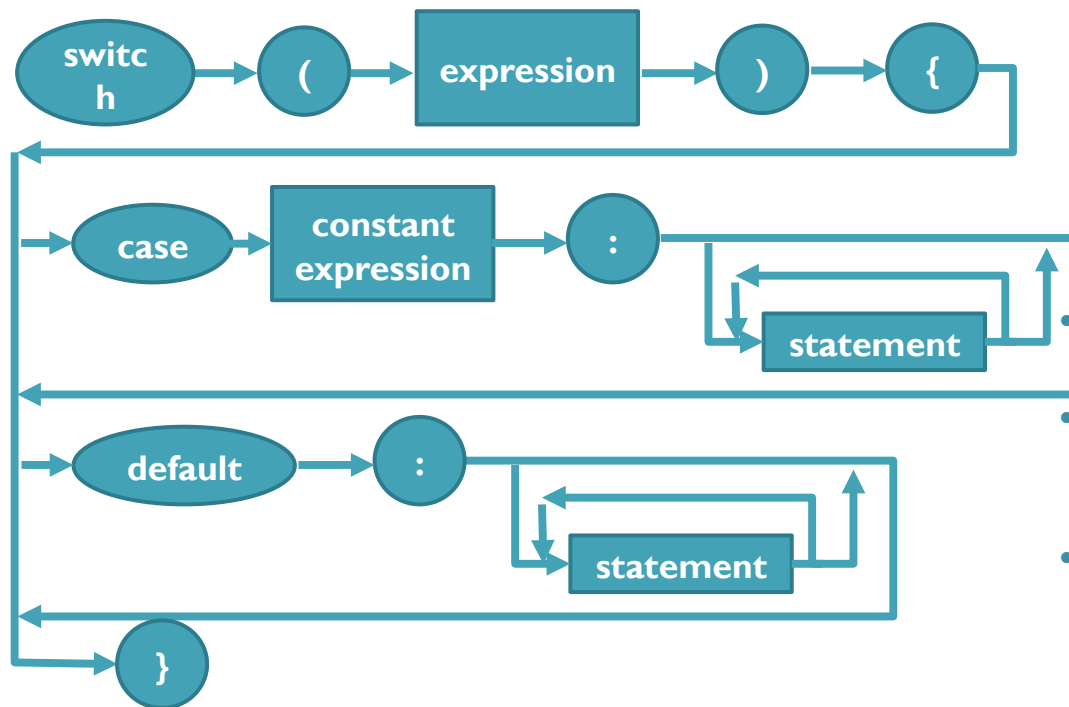
```
if (x)
    statement1; // executed only if x is non-0
else
    statement2; // executed only if x is 0
statement3;    //always executed
```

Nested if statements

- Note that when an **else** is immediately followed by an **if**, they are usually placed on the same line.
 - this is commonly called an **else if** statement.
- Nested if statements create the problem of matching each else phrase to the right if statement.
 - This is often called the **dangling else** problem !
 - An else is always associated with the nearest previous if.

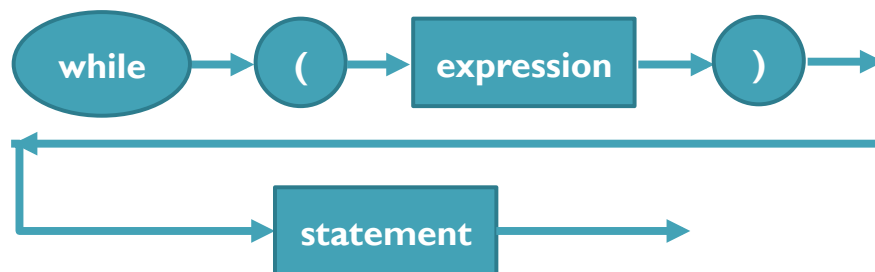
```
if(a<b)
    if(a<c)
        return a;
    else
        return c;
else if (b<c)
    return b;
else
    return c;
```

Switch statement



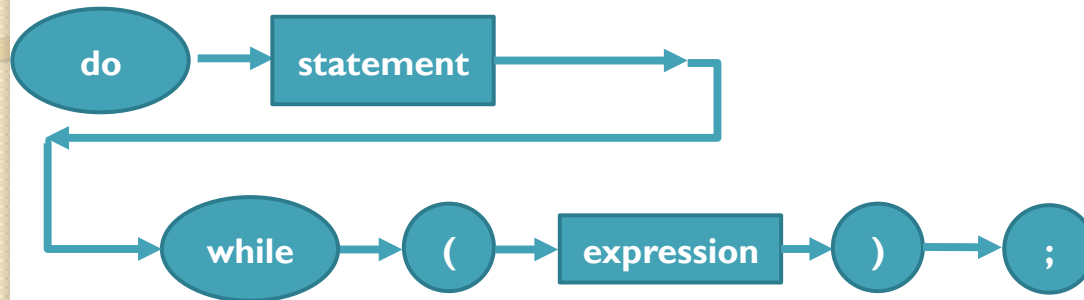
- The semantics of the **switch** statement are straight forward. The **switch** expression is evaluated, and if it matches one of the case labels, program flow continues with the statement that follows the matching case label.
 - If none of the case labels match the switch expression, program flow continues at the default label, **if exists!**
- No two case labels may have the same value!
- The default label need not be the last label, though it is good style to put it last
- You should almost always out a **break** statement at the end of each case label's statements, otherwise, once it enters a case, it'll go on to the next ones, until the end of switch "}". 32

The while statement



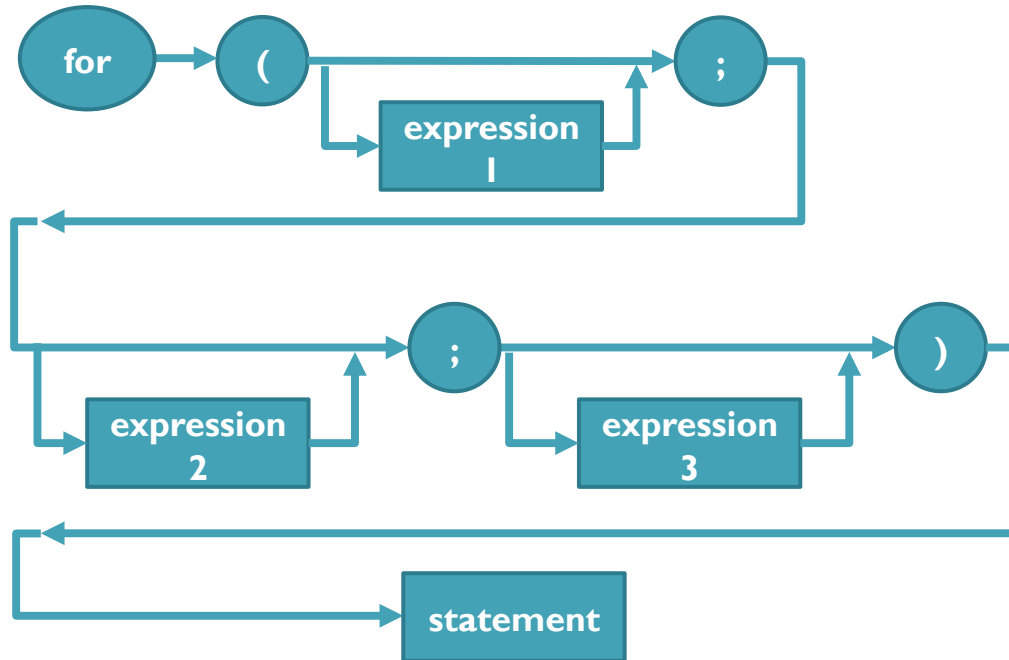
- First the expression is evaluated. If it is a **nonzero** value, statement is executed.
- After statement is executed, program control returns to the top of the while statement, and the process is repeated.
- This continues indefinitely until the expression evaluated to zero.

The do...while statement



- The only difference between a do..while and a regular while loop is that the test condition is at the bottom of the loop.
 - This means that the program always executes statement **at least one**.

The for statement



- First, **expression 1** is evaluated.
- Then **expression 2** is evaluated. This is the conditional part of the statement.
- If **expression 2** is **false**, program control exits the for statement. If **expression 2** is **true**, the **statement** is executed.
- After **statement** is executed, **expression 3** is evaluated. Then the statement loops back to test **expression 2** again.

NULL STATEMENTS

- It is possible to omit one of the expressions in a for loop, it is also possible to omit the body of the for loop.

```
for(c = getchar(); isspace(c); c = getchar());
```

- **ATTENTION**
- Placing a semicolon after the test condition causes compiler to execute a null statement whenever the if expression is **true**

```
if ( j == 1);
```

```
    j = 0;
```

```
/* j get assigned 0 regardless of whether j equals 1.*/
```

Nested loops

- It is possible to nest looping statements to any depth
- However, keep that in mind inner loops must finish before the outer loops can resume iterating
- It is also possible to nest control and loop statements together.

```
for( j = 1; j <= 10; j++) {  
    // outer loop  
    printf("%5d|", j);  
    for( k=1; k <= 10; k++) {  
        printf("%5d", j*k);  
        // inner loop  
    }  
    printf("\n");  
}
```

Break & continue & goto

- **break**

- We have already talked about it in **switch statement**
- When used in a loop, it causes program control jump to the statement following the loop

- **continue**

- continue statement provides a means for returning to the top of a loop earlier than normal.
- it is useful, when you want to bypass the reminder of the loop for some reason.
- Please do NOT use it in any of your C programs.

- **goto**

- goto statement is necessary in more rudimentary languages!
- Please do NOT use it in any of your C programs.



STORAGE CLASSES



Outline

- Fixed vs. Automatic duration
- Scope
- Global variables
- The ***register*** specifier
- Storage classes
- Dynamic memory allocation



Fixed vs. Automatic duration

- **Scope** is the technical term that denotes the region of the C source text in which a name's declaration is active.
- **Duration** describes the **lifetime** of a variable's memory storage.
 - Variables with fixed duration are guaranteed to retain their value even after their scope is exited.
 - There is no such guarantee for variables with automatic duration.
- A fixed variable is one that is stationary, whereas an automatic variable is one whose memory storage is automatically allocated during program execution.
- Local variables (whose scope limited to a block) are automatic by default. However, you can make them fixed by using keyword static in the declaration.
- The auto keyword explicitly makes a variable automatic, but it is rarely used since it is redundant.

Fixed vs. Automatic duration cont'd

```
void increment ( void ) {  
    int j = 1;  
    static int k = 1;  
    j++;  
    k++;  
    printf("j : %d\t k:%d\n", j, k);  
}  
  
main ( void ) {  
  
    increment();  
        j:2      k:2  
    increment();  
        j:2      k:3  
    increment();  
        j:2      k:4  
}
```

- Fixed variables initialized **only once**, whereas automatic variables are initialized **each time their block is reentered**.
- The ***increment()*** function increments two variables, ***j*** and ***k***, both initialized to 1.
 - ***j*** has automatic duration by default
 - ***k*** has fixed duration because of **static** keyword
- When ***increment()*** is called the 2nd time,
 - memory for ***j*** is reallocated & ***j*** is reinitialized to 1
 - ***k*** has still maintained its memory address and is **NOT** reinitialized.
- Fixed variables get a default initial value of **0**.

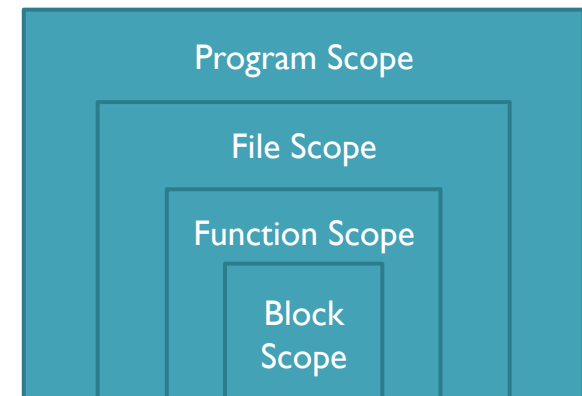


Scope

- The scope of a variable determines the region over which you can access the variable by name.
- There are four types of scope;
 - **Program scope** signifies that the variable is active among different source files that make up the entire executable program. Variables with program scope are often referred as global variables.
 - **File scope** signifies that the variable is active from its declaration point to the end of the source file.
 - **Function scope** signifies that the name is active from the beginning to the end of the function.
 - **Block scope** that the variable is active from its declaration point to the end of the block which it is declared.
 - **A block is any series of statements enclosed in braces.**
 - This includes compound statements as well as function bodies.

Scope cont'D

```
int i ;           // Program scope
static int j;     // File scope
func ( int k) {   // Block scope
    int m;        // Block scope
start :          // Function scope
}
```



scope cont'D

```
foo ( void ) {  
    int j=30, ar[20];  
    ...  
    {  
        // Begin debug code  
        int j; // This j does not  
               // conflict with other j's.  
               // it hides the upper-level scope j  
        for(j=0; j <= 10; ++j)  
            printf("%d\t", ar[j]);  
        // End debug code...  
    }  
    ...  
}
```

- A variable with a block scope can NOT be accessed outside its block.
- It is also possible to declare a variable within a nested block.
 - can be used for debugging purposes.
see the code on the left side of the slide!
- Although variable hiding is useful in situations such as these, it can also lead to errors that are difficult to detect!

Scope cont'D

- Function scope

- The only names that have function scope are **goto** labels.
- Labels are active from the beginning to the end of a function.
 - This means that labels must be unique within a function
- Different functions may use the same label names without creating conflicts

- File & Program scope

- Giving a variable file scope makes the variable active through out the rest of the file.
 - if a file contains more than one function, all of the functions following the declaration are able to use the variable.
 - To give a variable file scope, declare it outside a function with the **static** keyword.
- Variable with program scope, i.e. global variables, are visible to routines in other files as well as their own file.
 - To create a global variable, declare it outside a function **without** **static** keyword



Global variables

- In general, you should avoid using global variables as much as possible!
 - they make a program harder to maintain, because they increase complexity
 - create potential for conflicts between modules
 - the only advantage of global variables is that they produce faster code
- There are two types of declarations, namely, definition and allusion.
- An **allusion** looks just like a definition, but instead of allocating memory for a variable, it informs the compiler that a variable of the specified type exists but is defined elsewhere.
 - `extern int j;`
 - The `extern` keyword tells the compiler that the variables are defined elsewhere.
- Whenever you want to use global variables defined in another file you need to declare them with allusions.

The *register* specifier

- The **register** keyword enables you to help the compiler by giving it suggestions about which variables should be kept in registers (so, on CPU).
 - it is only a hint, not a directive, so **compiler is free to ignore it!**
 - The behavior is implementation dependent.
- Since a variable declared with register might never be assigned a memory address, **it is illegal to take address of a register variable.**
- A typical case to use register is when you use a counter in a loop.

```
int strlen ( register char *p) {  
  
    register int len=0;  
    while(*p++) {  
        len++;  
    }  
    return len;  
}
```

Remember that: registers are not addressable! They are not on the memory, but on the CPU

Storage classes summary

- **There are 4 storage-class specifiers**
- **auto**
 - Redundant and rarely used.
- **static**
 - In declarations within a function, static causes variables to have fixed duration. For variables declared outside a function, the static keyword gives the variable file scope.
- **extern**
 - For variables declared within a function, it signifies a global allusion. For declarations outside of a function, extern denotes a global definition.
- **register**
 - It makes the variable automatic but also passes a hint to the compiler to store the variable in a register whenever possible.
- **There are 2 storage-class modifiers**
- **const**
 - The const specifier guarantees that you can NOT change the value of the variable.
- **volatile**
 - The volatile specifier causes the compiler to turn off certain optimizations. Useful for device registers and other data segments that can change without the compiler's knowledge.
 - A variable should be declared volatile whenever its value could change unexpectedly. In practice, only 3 types of variables could change:
 - 1. Memory-mapped peripheral registers
 - 2. Global variables modified by an interrupt service routine
 - 3. Global variables accessed by multiple tasks within a multi-threaded application
 - Volatile tells the compiler not to optimize anything that has to do with the volatile variable.
 - There is only one reason to use it: When you interface with hardware. (Memory-mapped peripheral registers)
 - Another use for volatile is signal handlers.