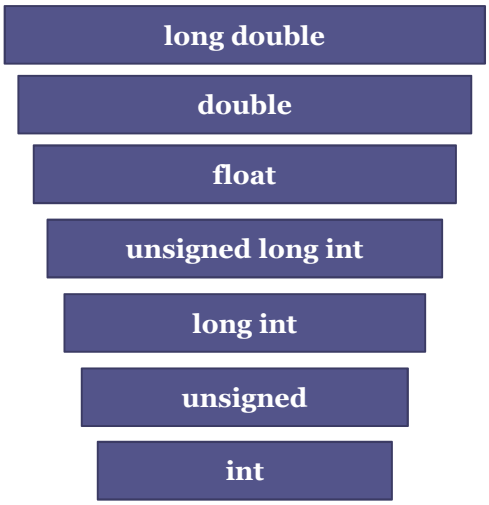
Structured Programming

H. İrem Türkmen

Mixing types

- C enables using variables of different types together



Mixing types
Integral widening conversion

- Integral widening conversion

```
float sum;
sum=2.0 + 1/2 ;          evaluted as;
sum=2.0 + 0;             sum=2.0
```

```
float sum;
sum=2.0 + 1.0/2 ;        evaluted as;
sum=2.0 + 0.5;           sum=2.5
```


Mixing types
Assignment conversion

- Assignment conversion

```
int sum;
sum=2.2  * 2;            evaluted as;
                         sum=4;
```

```
float sum;
sum=2.2  * 2 ;           evaluted as;
                         sum=4.4
```

Mixing types
Explicit Conversions - Casts

- **Explicit conversion is called casting and is performed with a construct called *a cast***

```
int j=2, k=3;
float f;
f = k / j ;                    // f=1
```

- **To cast an expression, enter the target data type enclosed in parenthesis directly before expression**

```
f = (float) k / j;             // f=1.5
```

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

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***
substract-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***

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	--	**--b**	decrement a, then get value of a

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

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

bit manipulation operators

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

bit manipulation operators

Expression	Binary representation	Result
9430	0010 0100 1101 0110	
5722	0001 0110 0101 1010	
9430 & 5722	0000 0100 0101 0010	1106

Expression	Binary representation	Result
9430	0010 0100 1101 0110	
5722	0001 0110 0101 1010	
9430 | 5722	0011 0110 1101 1110	14046

bit manipulation operators

Expression	5 << 1	
Binary model of Left Operand	00000000 00000101	
Binary model of the result	00000000 00001010	10 (decimal system)

Expression	255 >> 3	
Binary model of Left Operand	00000000 11111111	
Binary model of the result	00000000 00011111	31 (decimal system)

bit manipulation operators

Expression	Binary representation	Result
9430	0010 0100 1101 0110	
5722	0001 0110 0101 1010	
9430 ^ 5722	0011 0010 1000 1100	12290

Expression	Binary representation	Result
9430	0010 0100 1101 0110	
~9430	1101 1011 0010 1001	56105

bitwise assignment operators

Operator	Symbol	Form	Result
right-shift-assign	>>=	**a >>= b**	Assign a>>b to a.
left-shift-assign	<<=	**a <<= b**	Assign a<<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^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
  - **the expression may not have type function or void or be a bit field !**
- ***sizeof*** returns the number of bytes that operand occupies in memory
  - sizeof (3+5) returns the size of int
  - sizeof(short)

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

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.
  - 2 + 3 * 4
  - 3 * 4 + 2
- In cases where operators have the same precedence, associativity (sometimes called binding) is used to determine the order in which operands grouped with operators.
  - a = b = c;

Precedence & associativity

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

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 specifiy a particular grouping order.
  - (2 – 3) * 4
  - 2 – (3 * 4)

arithmetic assignment operators

int m = 3, n = 4, j=0;
float x = 2.5, y = 1.0;

m += n + x – y	m = (m + ((n+x) –y))	(8)
m /= x * n + y	m = (m / ((x*n) + y))	(0)
m += ++j * 2	m = (m + ((++j) * 2)	(3+1*2=5)

increment & decrement operators

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

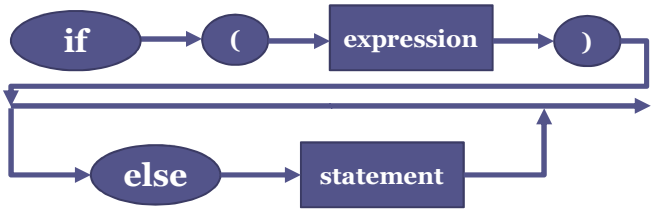
j: 6 k:4
j: 6 k:4

Increment & Decrement Operators

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

s = m++ - --j	(m++) – (--j)	(s=2)
s = m += ++j * 2	m = (m + ((++j) * 2)	(s=3)

if...else statement



Ex1:

```
if (x)
{
        statement1;
        statement2;
}
else
{
        statement3;
        statement4;
}
```

Ex2 :

```
if (x)
   statement1;          // executed only if x is nonzero
statement2;             //always executed
```

Ex3:

```
if (x)
   statement1;          // executed only if x is nonzero
else
   statement2;          // executed only if x is zero
statement3;             //always executed
```

switch statement

```
switch (expression)
{
   case constant1:
         // statements
         break;
   case constant2:
         // statements
          break;
         .
         .
         .
   default:
         // default statements
}
```

- The expression is evaluated once and compared with the values of each case label.
- If there is a match, the corresponding statements after the matching label are executed.
- If there is no match, the default statements are executed.
- If we do not use break, all statements after the matching label are executed.
- The default clause is optional.

switch statement

```
char x;
int a=3,b=5,c;
scanf("%c",&x);
switch (x)
{
         case 'A':
                  c=a+b;
                  printf("Addition      :%d",c);
                  break;
         case 'S':
                  c=a-b;
                  printf("Substraction  :%d",c);
                  break;
         case 'M':
                  c=a*b;
                  printf("Multiplication:%d",c);
                  break;
         default : printf("Invalid operand");
}
```

// statements inside the body of the loop
}

while statement

```
while (testExpression)
{
   // statements
}
```

- 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.

do...while statement

```
do
{
   // statements
}
while (testExpression);
```

- 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.***

for statement

```
for (expression1; expression2; expression3)
{
   // statements
}
```

- First, ***expression1*** is evaluated.
- Then ***expression2*** is evaluated. This is the conditional part of the statement.
- If ***expression2*** is **false**, program control exists the for statement. If ***expression2*** is **true**, the ***statements*** are executed.
- After ***statements*** are executed, ***expression3*** is evaluated. Then the statements loops back to test ***expression2*** again.

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