

Printf and Scanf

- Both formatted I/O
- Both sent to “standard I/O” location
- Printf
 - 🖱 Converts values to character form according to the format string
- Scanf
 - 🖱 Converts characters according to the format string, and followed by pointer arguments indicating where the resulting values are stored

Scanf (cont)

- Scanf requires two inputs:
 - 🔗 String argument - with format specifiers
 - 🔗 Set of additional arguments (pointers to variables)
- Consists of % at the beginning and a type indicator at the end
- Skips over all leading white space (spaces, tabs, and newlines) prior to finding first input value
- In between options:
 - 🔗 * = used to suppress input
 - 🔗 maximum field-width indicator
 - 🔗 type indicator modifier
- Input stops when:
 - 🔗 End of format string
 - 🔗 Input read does not match what the format string specifies i.e. pointer arguments MUST BE the right type
 - 🔗 The next call to scanf resumes searching immediately after the last character already converted.
- Return value = # of values converted

FORMAT	MEANING	VARIABLE TYPE
%d	read an integer value	int
%ld	read a long integer value	long
%f	read a real value	float
%lf	read a double precision real value	double
%c	read a character	char
%s	read a character string from the input	array of char

Scanf examples

```
int day, month, year;  
scanf("%d/%d/%d", &month, &day, &year);  
Input:  
01/29/64
```

```
int anInt;  
scanf("%*s %i", &anInt);  
Input:  
Age: 29  
anInt==29 result
```

```
int anInt;  
scanf("%i%%", &anInt);  
Input:  
23%  
anInt==23
```

```
double d;  
scanf("%lf", &d);  
Input:  
3.14  
d==3.14
```

```
int anInt, anInt2;  
scanf("%2i", &anInt);  
scanf("%2i", &anInt2);  
Input:  
2345  
anInt==23  
anInt2==45
```

```
int anInt; long l;  
scanf("%d %ld", &anInt, &l);  
Input:  
-23 200  
anInt==-23  
l==200
```


```
string s;  
scanf("%9s", s);  
Input:  
VeryLongString  
s=="VeryLongS"
```


Arithmetic type issues

- Type combination and promotion
 - 🖱 ('a' - 32) = 97 - 32 = 65 = 'A'
 - 🖱 Smaller type (char) is “promoted” to be the same size as the larger type (int)
 - 🖱 Determined at compile time - based purely on the types of the values in the expressions
 - 🖱 Does not lose information – convert from type to compatible large type


Arithmetic operators

Mathematical Symbols

 $+ - * / \%$

 addition, subtraction, multiplication, division, modulus

Works for both int and float

 $+ - * /$


➤ $/$ operator performs integer division if both operands are integer
i.e. truncates; otherwise, float

$\%$ operator divides two integer operands with an integer result of the remainder

Precedence – left to right

 $()$ always first

 $* / \%$

 $+ -$

Arithmetic type conversions

- Usual Arithmetic Conversions → Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

If either operand is long double, the other is converted to long double.

If either operand is double, the other is converted to double.

If either operand is float, the other is converted to float.

Otherwise, the integral promotions are performed on both operands;

If either operand is unsigned long int, the other is converted to unsigned long int.

If one operand is long int and the other is unsigned int, the effect depends on whether a long int can represent all values of an unsigned int; if so, the unsigned int operand is converted to long int; if not, both are converted to unsigned long int.

If one operand is long int, the other is converted to long int.

If either operand is unsigned int, the other is converted to unsigned int.

Otherwise, both operands have type int.

NOTE: There are two changes here. First, arithmetic on float operands may be done in single precision, rather than double; the first edition specified that all floating arithmetic was double precision. Second, shorter unsigned types, when combined with a larger signed type, do not propagate the unsigned property to the result type; in the first edition, the unsigned always dominated. The new rules are slightly more complicated, but reduce somewhat the surprises that may occur when an unsigned quantity meets signed. Unexpected results may still occur when an unsigned expression is compared to a signed expression of the same size.

Arithmetic Expressions – A bug's life

Pitfall -- int Overflow

"I once had a piece of code which tried to compute the number of bytes in a buffer with the expression $(k * 1024)$ where k was an int representing the number of kilobytes I wanted. Unfortunately this was on a machine where int happened to be 16 bits. Since k and 1024 were both int, there was no promotion. For values of $k \geq 32$, the product was too big to fit in the 16 bit int resulting in an overflow. The compiler can do whatever it wants in overflow situations -- typically the high order bits just vanish. One way to fix the code was to rewrite it as $(k * 1024L)$ -- the long constant forced the promotion of the int. This was not a fun bug to track down -- the expression sure looked reasonable in the source code. Only stepping past the key line in the debugger showed the overflow problem. "Professional Programmer's Language." This example also demonstrates the way that C only promotes based on the **types** in an expression. The compiler does not consider the values 32 or 1024 to realize that the operation will overflow (in general, the values don't exist until run time anyway). The compiler just looks at the compile time types, int and int in this case, and thinks everything is fine."

Arithmetic expressions - Truncation

Pitfall -- int vs. float Arithmetic

Here's an example of the sort of code where int vs. float arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```
{  
int score;  
...// suppose score gets set in the range 0..20 somehow  
7  
score = (score / 20) * 100;    // NO -- score/20 truncates to 0  
...
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression (score/20) will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number...

```
score = ((double)score / 20) * 100; // OK -- floating point division from cast  
score = (score / 20.0) * 100; // OK -- floating point division from 20.0  
score = (int)(score / 20.0) * 100; // NO -- the (int) truncates the floating  
  
// quotient back to 0
```


Example

```
#include <stdio.h>

int main()
{
    int first, second, add;
    float divide;

    printf("Enter two integers\n");
    scanf("%d %d", &first, &second);

    add = first + second;
    divide = first / (float)second;

    printf("Sum = %d\n", add);
    printf("Division = %.2f\n", divide);

    return 0;
}
```



Variables



Function calls



Input



Output



Operators



Typecasting

Relational Operators

- Used to compare two values
- < <= > >=
- == !=
- Precedence order given above; then left to right
- “else” equivalences (respectively)
 - >= > <= <
 - != ==
- Arithmetic operators have higher precedence than relational operators
- A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false.
 - For example, the check `0 == 2` evaluates to 0. The check `2 == 2` evaluates to a 1.

TRY: `printf(“%d”,2==1);`

Example

```
#include <stdio.h>
/* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300
   where the conversion factor is  $C = (5/9) \times (F-32)$  */
main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0;                /* lower limit of temperature scale */
    upper = 300;              /* upper limit */
    step = 20;                /* step size */
    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;    // problem? 9.0? Typecast?
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step; }
    return 0;
}
```

Example

```
#include <stdio.h>
#define MAGIC 10
int main(void)
{
    int i, fact, quotient;
    while (i++ < 3)           // value of i? need to initialize
    {
        printf("Guess a factor of MAGIC larger than 1: ");
        scanf("%d", &fact);
        quotient = MAGIC % fact;
        if (0 == quotient)
            printf("You got it!\n");
        else
            printf("Sorry, You missed it!\n");
    }
    return 0;
}
```

i++ is the same as:

i = i + 1

How evaluate?

i = i + 1 < 3

3 1 2

Problem, but...

(i = i + 1) < 3