CS 262 Lecture 3: User Input, Expressions

GEORGE MASON UNIVERSITY

Overview of Lecture 3

Operators

A familiar topic from other languages

Expressions

What exactly are expressions?

What types of expressions does C have?

Getting User Input

It’s a bit unique in C since C can directly manipulate memory

Reminders

Notices

The videos for lecture 2 content are posted under the Lecture 2 module on Canvas

A step-by-step guide for getting VSCode set up to connect to Zeus is posted under the Lecture 2 module

The practice midterm is updated to include material through last lecture

Operators

Operators are symbols that tell the compiler to perform a specific operation on one or more operands

C has many different types of operators

- Arithmetic operators
- Bitwise operators
- Relational operators
- Assignment operators
- Logical operators
- Increment and decrement operators
- Ternary operators
- Some additional special operators

Operators

Operators are symbols that tell the compiler to perform a specific operation on one or more operands

Arithmetic Operators:

Operator	Description	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`7 - 4`	`3`
`*`	Multiplication	`3 * 20`	`60`
`/`	Division	`80 / 10`	`8`
`%`	Modulus (remainder)	`5 % 2`	`1`

Note: Division between integers truncates the result (no decimals).
We will show how to address this later with **type casting**

Operators

Relational (comparison) operators:

Compare 2 values and return 1 (true) or 0 (false)

Operator	Description	Example	Result
==	Equal to	5 == 3	0
!=	Not equal to	5 != 3	1
<	Less than	5 < 2	0
>	Greater than	5 > 2	1
<=	Less than or equal to	5 <= 2	0
>=	Greater than or equal to	5 >= 2	1

Operators

Assignment operators (refresher from last class)

Assign values to variables (can be combined with other operations)

Operator	Description	Example	Result
`=`	Basic assignment	`x = 5`	`x = 5`
`+=`	Add and assign	`x += 2`	`x = x + 2`
`-=`	Subtract and assign	`x -= 2`	`x = x - 2`
`*=`	Multiply and assign	`x *= 2`	`x = x * 2`
`/=`	Divide and assign	`x /= 2`	`x = x / 2`
`%=`	Modulus and assign	`x %= 2`	`x = x % 2`

Operators

Ternary operators

A short of shorthand for if-else

Format:

```
result = (condition) ? value_if_true : value_if_false
```

Comma operators

Evaluates multiple expressions, returns the last value

```
int x = (y = 2, y + 3) // x is 5
```

Expressions

Expressions are any combination of variables, constants, and operators that the compiler can evaluate to produce a single value

Value: The result of evaluating the expression

Type: The data type of the result

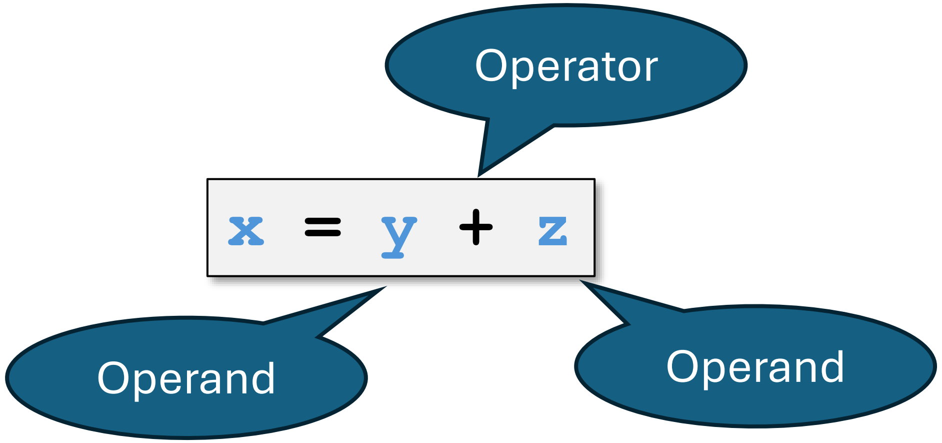
Side effects: The expression can change the state of the program (like using the assignment operator)

Examples:

```
x = 10;  // The assignment operator = assigns 10 to x
3 * (5 + 2);  // Result is = 21
int is_greater = (5 > 3); // Evaluates to 1 (true)
x = y + 2; // x gets the value of y + 2
```

Expressions: Formal Definitions

Binary Expression: An expression involving one operator and two operands



Operator

```
x = y + z
```

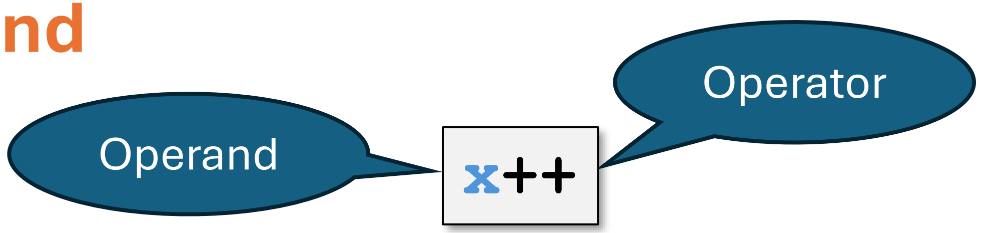
Operand

Operand

```
x = 10 + 30; // Binary expression
y = x / 1; // Another binary expression
z = x - y; // And another
```

Expressions: Formal Definitions

Unary expressions: Expressions involving one operator and one operand



Operator	Operator Name	Description
`++x`	Prefix increment	`Increments x, then evaluated`
`x++`	Postfix increment	`Evaluates x, then incremented`
`--x`	Prefix decrement	`Decrements x, then evaluated`
`x--`	Postfix decrement	`Evaluated, then decrements x`

Be careful with chaining prefix and postfix increments/decrements

If a was 10, then `b = a++;` will set b = 10 and a = 11

If a was 10, then `b = ++a;` will set b = 11 and a = 11

Expressions: Formal Definitions

Ternary expressions: Special conditional expressions of the form:

```
x ? y : z
```

If a is True (non-zero), then this expression evaluates to b’s value.

If a is False (zero), then this expression evaluates to c’s value instead.

```
set_speed = (speed > SPEED_LIMIT) ? SPEED_LIMIT : speed;
// equivalent if-else:
if(speed > SPEED_LIMIT) {
    set_speed = SPEED_LIMIT;
}
else {
    set_speed = speed;
}
```

Operator Precedence

Operator precedence table:

Precedence is which operators are evaluated first.

Associativity is which order operators in the same precedence are evaluated.

Precedence	Operators	Description	Associativity
1 (highest)	(...), [ ]	Function calls, array subscript	Left to Right
2	!, ~, ++, --, type casts	Unary operators, casts	Right to Left
3	*, /, %	Multiplication, Division, Modulus	Left to Right
4	+, -	Arithmetic	Left to Right
7	==, !=	Comparisons	Left to Right
11	&&	Logical AND	Left to Right
12	||	Logical OR	Left to Right
13	? :	Ternary (Conditional)	Right to Left
14	=, +=, -=, *=, /=, %=	Assignments	Right to Left
15	,	Comma operator	Left to Right

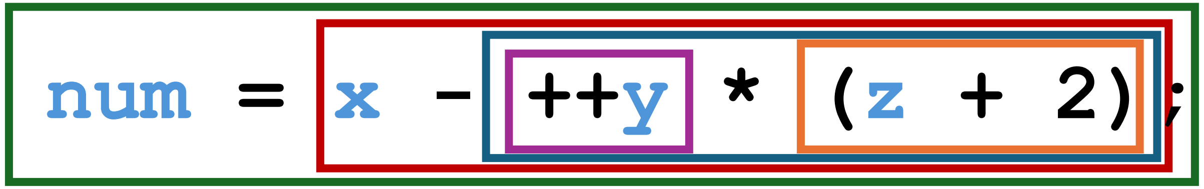
Operator Precedence - Example

What order do we think the following expression will be evaluated in?

```
int num = x - ++y * (z + 2);
```

Operator Precedence - Example

```
int x = 2;
int y = 3;
int z = 5;
```



```
int num = x - ++y * (z + 2);
```

The unary increment is evaluated: `++y`, so now y equals 4

Then, we evaluate `z + 2`: now `z` equals 5

Now, the multiplication between `++y` and `(z + 2)` is evaluated:

`++y * (z + 2) = 28`

Next, x – the above result is computed:

`x - ++y * (z + 2) = -26`

Finally, `num` is set equal to this result

Type Casting

One type can be cast to another for an expression

This results in only a **temporary** change for the variable

Cast operator:

(type) variable

Example:

ch is cast to an int here

But ch is still a char (this prints 1 since char is 1 byte)

```
char ch = ‘A’;
int ascii_value = (int)ch);
printf(“ch is %d bytes\n”, sizeof(ch));
printf(“ascii_value is %d bytes\n”, sizeof(ascii_value));
```

Quick Digression: The sizeof() Function

The `sizeof()` function returns the size (in bytes) of what is passed in

`sizeof(int)` returns the number of bytes of an `int`

`sizeof(char)` returns the number of bytes of a `char`

`sizeof(x)` returns the number of bytes of the variable `x`

`sizeof(ch)` and `sizeof(ascii_value)` would show that `sizeof(ch)` is still 1 byte (the size of a `char`) and `sizeof(ascii_value)` is whatever size an `int` is on your system

```
char ch = ‘A’;
int ascii_value = (int)ch);
printf(“ch is %d bytes\n”, sizeof(ch));
printf(“ascii_value is %d bytes\n”, sizeof(ascii_value));
```

Implicit Conversions and “Promotion”

In C, all operands must have the same type

When the types differ, one is implicitly converted

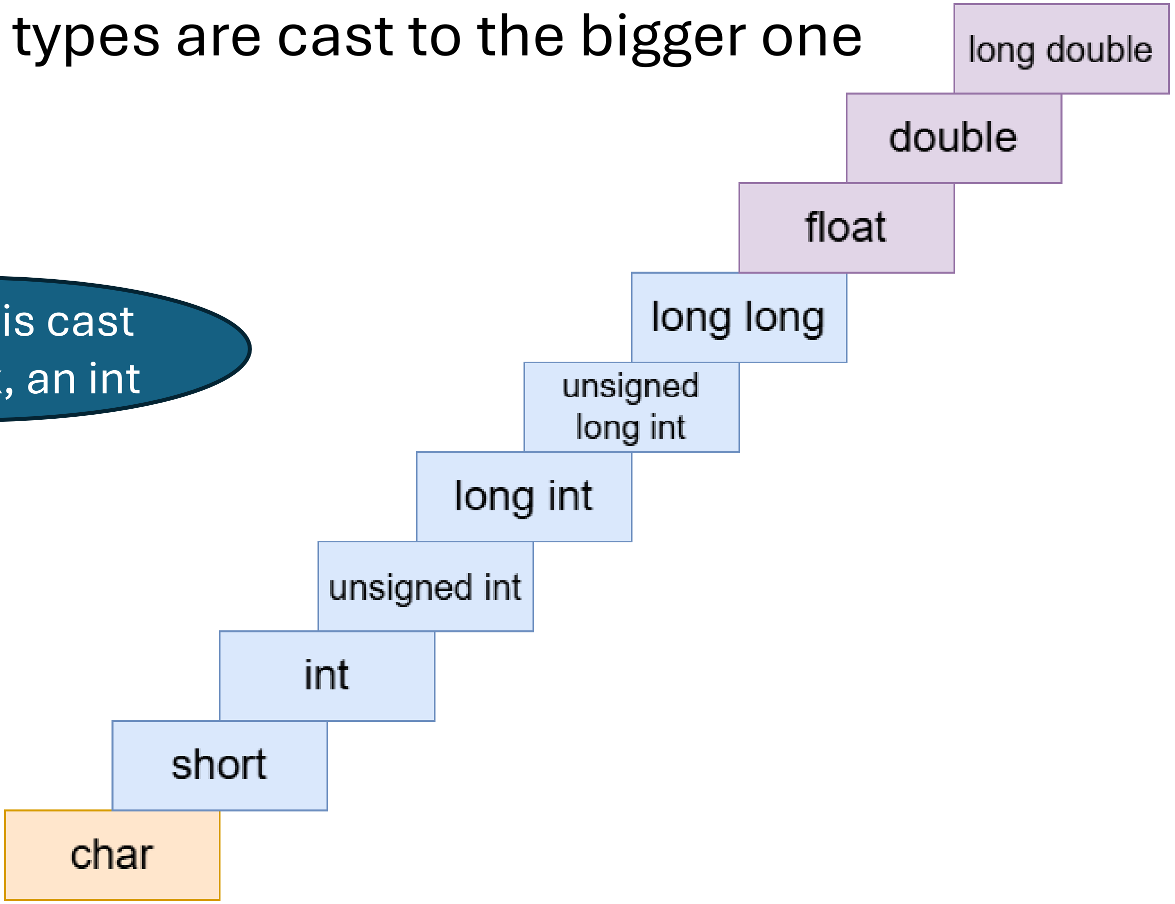
This follows a set hierarchy, where smaller types are cast to the bigger one

Example:

```
int x = 100;
short y = 5;
long z = (x + y);
```

y, a short, is cast to match x, an int

The final result is cast to match z, a long



long double

double

float

long long

unsigned long int

long int

unsigned int

int

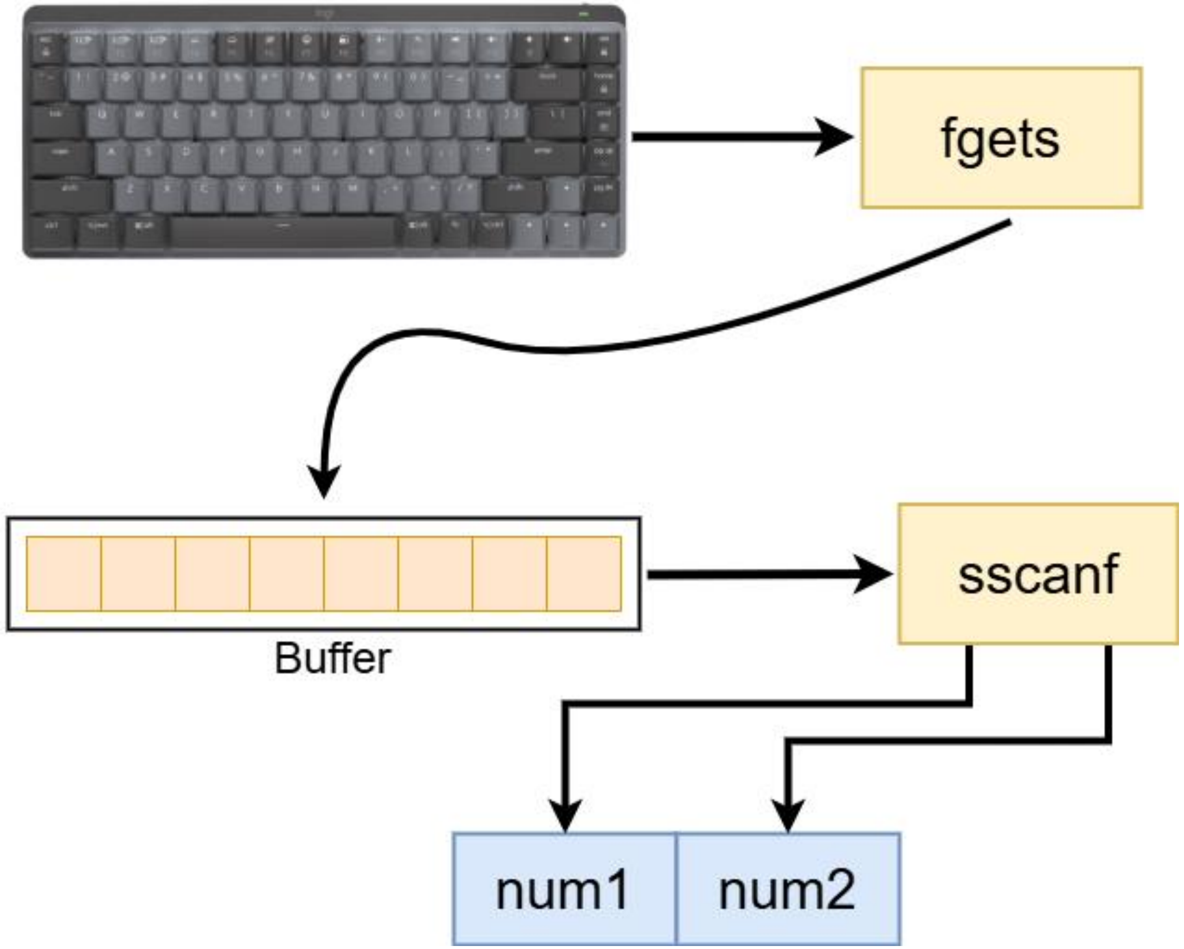
short

char

User Input

A string is read into the buffer from the keyboard (stdin)

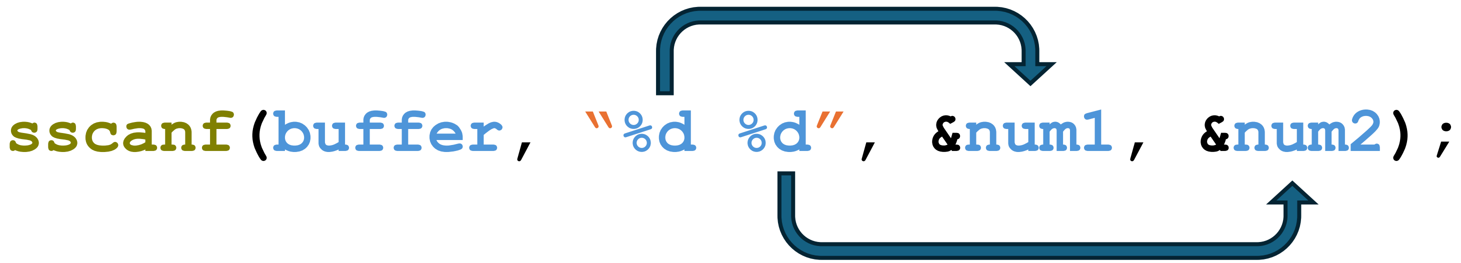
fgets reads the formatted data and stores values in variables



User Input

The sscanf function reads a formatted string into variables

```
int sscanf(buffer, “Format codes”, &variable);
```



```
sscanf(buffer, “%d %d”, &num1, &num2);
```

If `buffer` is a string with 2 numbers, `sscanf` will read the first number and put it into the first variable `val1`, then read the second number and put it into `val2`

User Input

% Conversion codes for printf and sscanf

% Code	Description	Example
c	Character	sscanf(buffer, “%**c**”, &val);
i or d	Integer	sscanf(buffer, “%**d**”, &val);
u	Unsigned Int	sscanf(buffer, “%**u**”, &val);
hd	short int (short)	sscanf(buffer, “%**hd**”, &val);
ld	long int (long)	sscanf(buffer, “%**ld**”, &val);
f	floating-point	sscanf(buffer, “%**f**”, &val);
lf	double	sscanf(buffer, “%**lf**”, &val);
s	String	sscanf(buffer, “%**s**”, val);

User Input

The fgets function lets us read a string from the keyboard

```
fgets(buffer, buffer_len, stdin);
```

`fgets` reads input from the keyboard (`stdin`) into a string, called the **buffer**, containing enough space for **buffer_len** total characters in it

There are other ways of getting input, however these are *unsafe*

`scanf` read input directly into variables, and can result in overflows

`gets`, which reads a full line of input with no limit on character count, is so unsafe it was removed from modern C standards

User Input

The buffer used by `fgets` is an array we create

```
char buffer[buffer_len];
```

The steps for getting user input are

1. We create an array of **`buffer_len`** characters to hold the user input
2. We read the user input
3. We use **`sscanf`** to load the data into the variables