University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Introduction to the C Programming Language

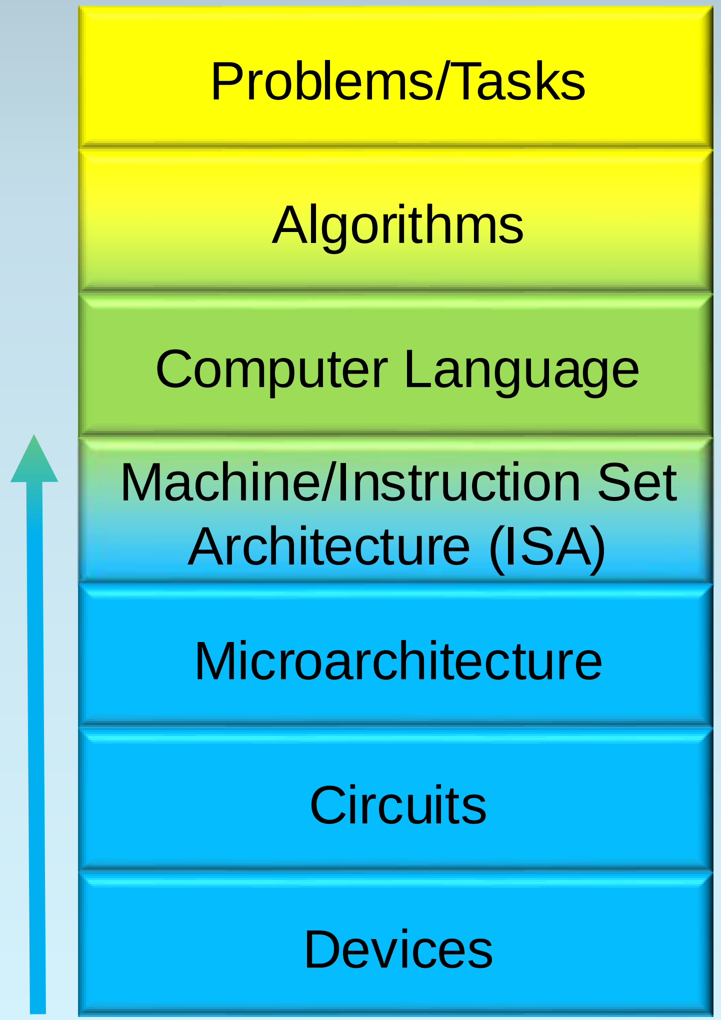
Few Programmers Write Instructions (Assembly Code)

So far, you learned to **use bits to represent information**.

Our class will teach you **how to design a computer**.

But computer **instructions are quite simple** (add two numbers, copy some bits).

Not many programmers use them directly.

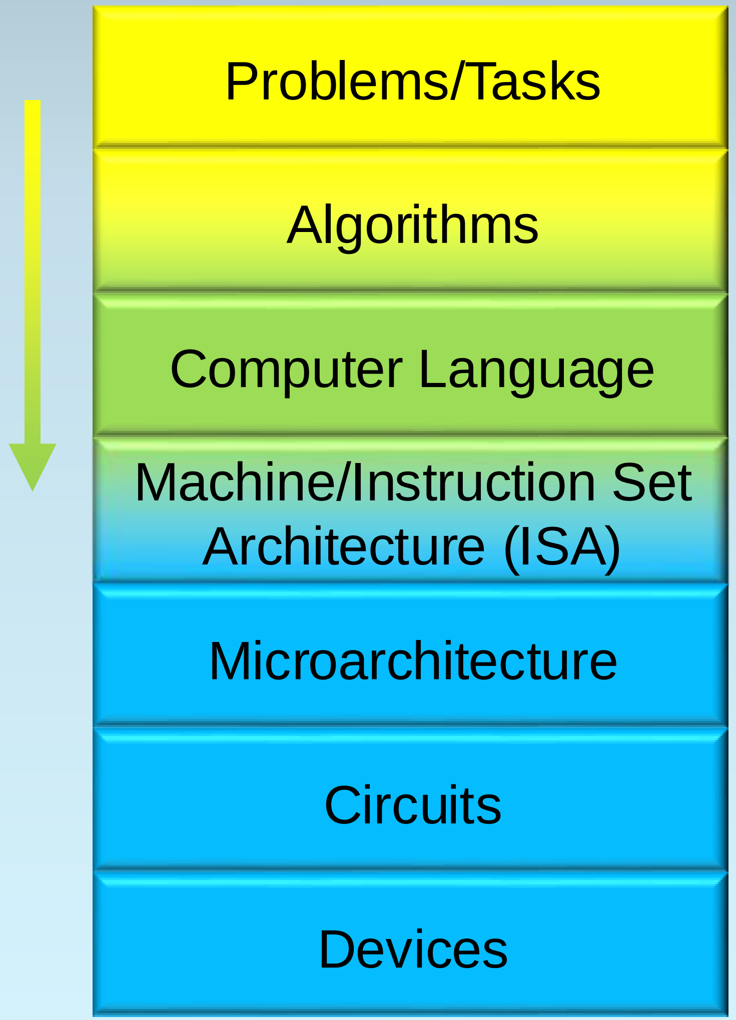


Most Programs Are Written in High-Level Languages

Since 1954 (FORTRAN), people have been trying to bridge the **semantic gap** between human problems/tasks and ISAs.

The result is 1000s of computer languages.

Most programs are written in these languages.



Spend a Week Learning the C Programming Language

Before we move upwards from bits into gates, we will spend a week on the language **C**.

Why?

- Allow more time to **become familiar with mechanical aspects** of computer languages (2 semesters instead of 2/3 of a semester in ECE classes a few years ago).
- **Start simple**: make small modifications.
- **Read examples** before writing your own.

We Will Not Teach You How to Program (Yet)

To be clear:

> Programming means translating a human task into an algorithm expressed in a computer language (or an ISA).

We are **NOT teaching you how to program** yet.

So What ARE We Teaching You Now?

Three skills:

- how to **express certain types of tasks formally** enough for a computer to understand them,
- how to **read and interpret (simple) formal expressions** of computation in **C**, and
- how to **use a compiler** to translate a **C** program into instructions.

Computers (Programs) Help with Digital Design

Remember: the world is digital.

So we will

- **connect these skills** (expressing tasks and reading **C** programs) **to the material** (how to build a computer)
- to **help you learn the skills**
- and to realize that **computers can help** with much of what you are learning.

What about Programming?

So far, **computers don't know how to program**.

In our class,

- you will start learning that skill (art)
- in part 4 of the class
  (week 12 / early April in Spring,
  or early November in Fall).

A Brief History of C

The **C programming language** was
- developed by Dennis Ritchie in 1972
- to simplify the task of writing Unix.

C has a transparent mapping to typical ISAs:
- easy to understand the mapping (ECE220)
- easy to teach a computer:
  C compiler (a program) converts a
  C program into instructions

C was first standardized in 1989 by ANSI.

Starting a Program Executes its **main** Function

Let's take a look at a C program…

```
int
main ()
{
    int answer = 42;  /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here! */
    return 0;
}
```

The function **main** executes when the program starts.

After **main** has finished, the program terminates.

The Function `main` Divides into Two Parts

`main` consists of two parts…

```
int
main ()
{
    int answer = 42;    /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here! */
    return 0;
}
```

Declarations for variables used by **`main`**.

A sequence of statements.

What Does the Program Do? Execute Statements in Order

Prints “The answer is 42.” followed by an ASCII newline character to the display.

```
int
main ()
{
    int answer = 42;  /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here! */
    return 0;
}
```

Terminates the program; returns 0 (success, by convention) to the operating system.

Comments Help Human Readers (Including the Author!)

Good programs have many comments…

Comments start with /* and end with */ .

```
int
main ()
{
    int answer = 42; /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here! */
    return 0;
}
```

Comments can span more than one line.

So Far, We Have Four Pieces of C Syntax

a few elements of **C syntax***:

- **`main`**: the function executed when a program starts
- **variable declarations** specify symbolic names and data types
- **statements** tell the computer what to do
- **comments** help humans to understand the program

* A computer language's **syntax** specifies the rules that one must follow to write a valid program in that language.

Pitfall: “Functions” in Programs are not Functions in Math

Be careful about terminology:

- **`main` is a “function”**
- **in the syntactic sense of the C language** (a set of variable declarations and a sequence of statements ending with a **`return`** statement)
- **but not necessarily in the mathematical sense**.

A "Function" is a Block of Code that Returns a Value

For example,

- although **`main`** does return an integer,
- we can **write a program that returns a random integer from 0 to 255**.

Given the same inputs,

- the value returned is **not unique**, and
- the value returned is **not reproducible** (running the program two times can give different answers).
- **Both properties are required for a mathematical function**.

Pitfall #2: "Functions" are Not Algorithms

The `main` function is **not necessarily an algorithm**.

For example, we can **write a program that runs forever** (never terminates, and never returns a value).

Algorithms must be finite
(see Patt & Patel).

Variable Declarations Allocate and Name Sets of Bits

Variable declarations

- allow the programmer to **name sets of bits**
- and to **associate a data type**

The declaration `int answer = 42;`

tells the compiler…

- to make space for a **32-bit 2's complement** number (an **`int`**),
- to initialize the bits to the bit pattern for 42,
- and to make use of those bits whenever a statement uses the **symbolic name answer**.

Pitfall #3: Variables in C are Not Variables in Algebra

In algebra, a variable is a name for a value.

A variable's value does not change.

For example:
- If we write **A=42** in algebra,
- the variable **A** continues to be equal to **42**
- for the duration of that problem or calculation.

In C, any statement can change the value of a variable.

Variables in C are Sets of Bits (0s and 1s)

In C, a variable is a name for a set of bits.

The bits will (of course!) **always be 0s and 1s**.

But **variables in C can change value as the program executes**.

Other properties of a variable must be inferred from the program (in the example program, **answer** is always 42, because no statement changes **answer**).

Each Variable Has a Specific Data Type

Many languages (such as **C**) require that the programmer **specify a data type for each variable**.

A **C** compiler uses a variable's data type to interpret statements using that variable.

For example, a "+" operation in **C** might mean to add two sets of bits

- as **unsigned** bit patterns,
- as **2's complement** bit patterns, or
- as **IEEE single-precision floating-point** bit patterns.

The compiler generates the appropriate instructions.

Primitive Data Types are Always Available

Primitive data types

- part of the **C** language
- include **unsigned**, **2's complement**, and **IEEE floating-point**
- 8-bit primitive data types can also be used to store **ASCII** characters

Pitfall #4: Primitive Data Types Depend on the System

Since the **C** language was designed to be efficient, **primitive data types are tuned to the system**.

Unfortunately, that means the actual data type can vary from one compiler to another.

For example, `long int` may be a **32-bit 2's complement** value, or it may be a **64-bit 2's complement** value.

Use `int32_t` or `int64_t` to be specific.

Code Examples in Slides Use Only a Few Types

We use these data types in examples.

name	meaning on lab machines
`char`	8-bit 2's complement / ASCII
`int`	32-bit 2's complement

(Add "`unsigned`" before types above for unsigned.)

name	meaning on lab machines
`float`	IEEE 754 single-precision floating-point (32 bits)
`double`	IEEE 754 double-precision floating-point (64 bits)

See the notes for a more complete listing.

Each Variable Also Has a Name (an Identifier)

Rules for **identifiers** in **C**

- composed of **letters and digits** (start with a letter)
- any length
  - **use words** to make the meaning clear
  - avoid using single letters in most cases
- **case-sensitive**
  - The following are distinct identifiers: variable, Variable, VARIABLE, VaRiAbLe.
  - **Do NOT use more than one**!

Examples of Variable Declarations

Putting the pieces together, a variable declaration is

```
<data type> <identifier> = <value>;
```

Here are a few examples:

```
int anIntegerIn2sComplement = 42;
unsigned int andOneUnsigned = 100;
float IEEE_754_is_Cool = 6.023E23;
```

Variables Always Contain Bits

The initialization for a variable is optional.

So the following is acceptable:

```
<data type> <identifier>;
```

For example,

```
int i;
```

What is the initial value of `i`?

You guessed it! **BITS**!
(They may be 0 bits, but they may not be.)

Statements Tell the Computer What to Do

In **C**, a statement specifies a complete operation.

In other words, **a statement tells the computer to do something**.

The function **`main`** includes a sequence of statements.

When program is **started** (or **runs**, or **executes**),

- **the computer executes the statements in `main`**
- in the order that they appear in the program.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Expressions and Operators in C

Expressions are Used to Perform Calculations

Let's talk in more detail starting with a fifth element of **C** syntax: expressions.

An **expression** is a calculation consisting of variables and operators.* For example,

A + 42

A / B

Deposits – Withdrawals

* And function calls, but that topic we leave for ECE220.

Our Class Focuses on Four Types of Operator in C

The **C** language supports many operators.

In our class, we consider four types:

- **arithmetic** operators
- **bitwise** Boolean operators
- **relational** / **comparison** operators
- the **assignment** operator

We also introduce logical operators, but leave their full meaning for ECE220.

Five Arithmetic Operators on Numeric Types

Arithmetic operators in **C** include

- addition: **+**
- subtraction: **–**
- multiplication: *****
- division: **/**
- modulus: **%** (integers only)

The **C** library includes many other functions, such as exponentiation, logarithms, square roots, and so forth. We leave these for ECE220.

Arithmetic Mostly Does What You Expect

Declare: `int A = 120;  int B = 42;`

Then…

`A + B`	**evaluates to**	`162`
`A – B`	evaluates to	`78`
`A * B`	evaluates to	`5040`
`A % B`	evaluates to	`36`
`A / B`	evaluates to…	`2`

What's going on with division?

A Few Pitfalls of C Arithmetic

No checks for overflow, so be careful.

- `unsigned int A = 0 - 1;`
- `A` is a large number!

Integer division

- Trying to **divide by 0** ends the program (floating-point produces **infinity** or **NaN**).
- Integer division **evaluates to an integer**, so `(100 / 8) * 8` **is not 100**.

C Behavior Sometimes Depends on the Processor

Integer division is rounded to an integer.

Rounding **depends on the processor**.

Most modern processors **round towards 0**, so…

`11 / 3` evaluates to `3`

`-11 / 3` evaluates to `-3`

Modulus `A % B` is defined such that

`(A / B) * B + (A % B)` is equal to `A`

So `(-11 % 3)` evaluates to `-2`.

Modulus is not always positive.

Six Bitwise Operators on Integer Types

Bitwise operators in C include

- AND: **&**
- OR: |
- NOT: ~
- XOR: ^
- left shift: <<
- right shift: >>

In some languages, ^ means exponentation, but not in the **C** language.

Bitwise Operators Treat Numbers as Bits

Declare: `int A = 120;  int B = 42;`

```
/* A = 0x00000078, B = 0x0000002A
using C's notation for hexadecimal. */
```

Then…

`A & B` evaluates to **40** **0x00000028**

```
    0000 0000 0000 0000 0000 0000 0111 1000
AND 0000 0000 0000 0000 0000 0000 0010 1010
-------------------------------------------
    0000 0000 0000 0000 0000 0000 0010 1000
```

Apply AND to pairs of bits.

Bitwise Operators Treat Numbers as Bits

Declare: `int A = 120;  int B = 42;`

```
/* A = 0x00000078, B = 0x0000002A
using C's notation for hexadecimal. */
```

Then…

`A & B`	evaluates to	**40**	**0x00000028**
`A	B`	evaluates to	**122**	**0x0000007A**
`~A`	evaluates to	**-121**	**0xFFFFFF87**
`A ^ B`	evaluates to	**82**	**0x00000052**

Left Shift by N Multiplies by 2^N

Shifting left by N bits adds **N** 0s on right.

- It's like **multiplying by 2^N**.
- **N** bits lost on left! (**Shifts can overflow.**)

Declare: `int A = 120;/* 0x00000078 */`

```
unsigned int B = 0xFFFFFF00;
```

Then…

`A << 2`	evaluates to	`480`	`0x000001E0`
`B << 4`	evaluates to	`(<B!)`	`0xFFFFF000`

Right Shift by N Divides by 2^N

A question for you: **What bits appear on the left when shifting right?**

Declare: `int A = 120;/* 0x00000078 */`

`A >> 2` evaluates to `30` `0x0000001E`

What about `0xFFFFFF00 >> 4`?

Is `0xFFFFFF00` equal to

`-256` (/16 = `-16`, so insert 1s)? or equal to

`4,294,967,040` (/16 = `268,435,440`, insert 0s)?

Right Shifts Depend on the Data Type

A **C** compiler **uses the type of the variable** to decide which type of right shift to produce

For an `int`

- **2's complement** representation
- produces **arithmetic right shift**
- (copies the sign bit)

For an `unsigned int`

- **unsigned** representation
- produces **logical right shift**
- (inserts 0s on left)

Right Shift by N Divides by 2^N

Declare: `int A = -120;/* 0xFFFFFF88 */`

`unsigned int B = 0xFFFFFF00;`

Then…

`A >> 2`	evaluates to	`-30`	`0xFFFFFFE2`
`A >> 10`	evaluates to	`-1`	`0xFFFFFFFF`
`B >> 2`	evaluates to		`0x3FFFFFC0`
`B >> 10`	evaluates to		`0x003FFFFF`

Notice that **right shifts round down**.

Six Relational Operators

Relational operators in **C** include

- less than: **<**
- less or equal to: **<=**
- equal: **==** (TWO equal signs)
- not equal: **!=**
- greater or equal to: **>=**
- greater than: **>**

C operators cannot include spaces, nor can they be reordered (so no "**< =**" nor "**=<**").

Relational Operators Evaluate to 0 or 1

In C,
- **0 is false**, and
- **all other values are true**.

Relational operators always
- **evaluate to 0 when false**, and
- **evaluate to 1 when true**.

Relational Operators Also Depend on Data Type

Declare:
```
int A = -120;/* 0xFFFFFF88 */
int B =  256;/* 0x00000100 */
```

Is `A < B`?

- Yes, -120 < 256.
- But if the same bit patterns were interpreted using the **unsigned** representation,

```
0xFFFFFF88 > 0x00000100
```

As with shifts, a **C** compiler **uses the data type to perform the correct comparison**.

The Assignment Operator Can Change a Variable's Value

The **C** language uses **=** as the **assignment operator**. For example,

```
A = 42
```

changes the bits of variable **A** to represent the number **42**.

One can write **any expression on the right-hand side of assignment**. So

```
A = A + 1
```

increments the value of variable **A** by **1**.

Only Assign Values to Variables

A C compiler can not solve equations.

For example,

```
A + B = 42
```

results in a compilation error (the compiler cannot produce instructions for you).

The left-hand side of an assignment must be a variable.*

* For ECE120. ECE220 teaches other ways to use the assignment operator.

Pitfall of the Assignment Operator

Programmers sometimes

- write "`=`" (assignment)
- instead of "`==`" (comparison for equality).

For example, to compare variable **A** to **42**,

- one might want to write "`A == 42`"
- but instead write "`A = 42`" by accident.

A **C** compiler can **sometimes** warn you (in which case, fix the mistake!).

Good Programming Habits Reduce Bugs

To avoid these mistakes, get in the habit of writing comparisons with the variable on the right.

For example, instead of "`A == 42`", write

```
42 == A
```

If you make a mistake and write "`42 = A`",

- the **compiler will always tell you**,
- and you can fix the mistake.

* *

Three Logical Operators

Logical operators in **C** include

- AND: **&&**
- OR: **||**
- NOT: **!**

Logical operators operate on truth values (again, **0 is false**, and **non-zero is true**).

Logical operators

- **evaluate to 0 (false)**, or
- **evaluate to 1 (true)**.

* *

Logical Operators Depend only on True/False in Operands

Declare: `int A = 120;` `int B = 42;`

Then…

`(0 > A || 100 < A)`	evaluates to `1`
`(120 == A && 3 == B)`	evaluates to `0`
`!(A == B)`	evaluates to `1`
`!(0 < A && 0 < B)`	evaluates to `0`
`(B + 78 == A)`	evaluates to `1`

(So no bitwise calculations, just true/false.)

Operator Precedence in C is Sometimes Obvious

A task for you:

Evaluate the C expression: `1 + 2 * 3`

Did you get 7?

Why not 9? (1 + 2) * 3

Multiplication comes before addition

- in elementary school
- and in **C**!

The order of operations is called operator **precedence**.

Never Look Up Precedence Rules!

Another task for you:

Evaluate the C expression: `10 / 2 / 3`

Did you get 1.67?

Is it a friend's birthday?

Perhaps it causes a divide-by-0 error?

Or maybe it's … 1?   (10 / 2) / 3, as **`int`**

If the order is not obvious,

- Do NOT look it up.
- **Add parentheses**!