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 November).
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...

```c
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...

```c
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

```c
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;
}
```

- Prints "The answer is 42."
- Followed by an ASCII newline character to the display.
- Terminates the program: returns 0 (success, by convention) to the operating system.
Comments Help Human Readers (Including the Author!)

Good programs have many comments...

```c
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning on Lab Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8-bit 2's complement / ASCII</td>
</tr>
<tr>
<td>int</td>
<td>32-bit 2's complement</td>
</tr>
<tr>
<td>float</td>
<td>IEEE 754 single-precision floating-point (32 bits)</td>
</tr>
<tr>
<td>double</td>
<td>IEEE 754 double-precision floating-point (64 bits)</td>
</tr>
</tbody>
</table>

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:

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

Here are a few examples:

```c
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:

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

For example,

```c
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.