Everything You Should Know About Programming

From Beginner to Advanced

Aaron Lengyel

Abstract

This is essentially a snapshot of everything you should seek to learn when trying to program. I don't go into the very fine details about implementations but you should at least be able to use this as a general resource for what certain things are and why they show up in certain languages. You will also be up to date on some modern practices that are important to know that you won't be taught when you do a short course on how to program.

Contents

Introduction

What is Programming?

Programming is the act of creating instructions and algorithms for a computer to perform automatically. Essentially it is the act of telling a computer how and when to manipulate certain data.

What will I cover?

I will be covering the following topics in as much detail necessary, I won't be spending much time per topic only a few minutes at most. This isn't meant to be a step-by-step guide, it's meant to make you familiar with a lot of the main components of programming and software design at the start. Later on I plan to make tutorials for each piece individually, going into more detail, but for now we will just have to settle for a quick overview.

1. Understanding Computation
    1. Boolean Logic & Algebra
    2. Different Models of Computation
        1. Turing
        2. Church
    3. Binary
    4. Algorithms
    5. Program Structue
        1. Memory
        2. Pointers
    6. Compilation vs. Interpretation
2. Basic Data Types (Primitives)
    1. Booleans
    2. Integers
    3. Floats
    4. Doubles
    5. Chars
    6. Strings
3. Actual Programming
    1. Logic
    2. Loops
    3. Functions
    4. Error Handling
4. Common Data Types and Structures
    1. Arrays
    2. Tuples
    3. Lists (ArrayLists)
    4. HashMaps
    5. Sets
    6. Classes
5. Programming Paradigms
    1. Declarative vs. Imperative
    2. Procedural
    3. Object-Oriented
        1. Encapsulation
        2. Inheritance
        3. Composition
        4. Design Patterns
    4. Functional
        1. Anonymous Functions
        2. Closures
        3. Iterators

Understanding Computation

Boolean Logic

Named after George Bool: Boolean Logic, often called Boolean Algebra, is a branch of algebra where the variables deal with individual truth values. These are combined with various logical operations that are similar to the way we combine arguments in normal languages.

1. Negation ($\neg$)
    - Negation essentially flips the truth value of a single variable

A	$\neg A$
T	F
F	T

2. Logical Conjuction ($\wedge$)
    - This is also called a logical AND. This operation take two truth values and combines them returning true if both variables are true and false otherwise.
    - It works like the *and* in english.

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

3. Logical Disjunction ($\vee$)
    - This is also called a logical OR, it's truth value is true if either one of the combining variables are true.
    - This has the same intuition of *or* in english.

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

We can also combine these basic operations to form more complex sentences:

$$\neg(\neg B \wedge A) \wedge \neg(C \vee \neg D)$$

Some secondary operations we can construct include:

1. Exclusive Or ($\oplus$)
    - This is a special version of OR that returns false if both values are true.
        - It's construction is as follows:

$$x \oplus y = (x \vee y) \wedge \neg(x \wedge y)$$

A	B	$A \oplus B$
T	T	F
T	F	T
F	T	T
F	F	F

2. Material Implication ($\rightarrow$)
    - The material implication is essentiall an **if then** scenario. $A \rightarrow B$ roughly would translate to *if* A, *then* B.

– It's construction is as follows:

$$x \oplus y = \neg x \vee y$$

A	B	$A \to B$
T	T	T
T	F	F
F	T	T
F	F	T

3. Equivalence ($\equiv$)
   - Equivalence means that two variables have the same truth value. This means that it returns true if they are both false or if they are both true.
     – It's construction is as follows:

$$x \equiv y = (x \wedge y) \vee \neg(x \vee y)$$

A	B	$A \equiv B$
T	T	T
T	F	F
F	T	F
F	F	T

Notice that $(A \equiv B) = \neg(A \oplus B)$

Since our variables can only be true or false, it often becomes easy to look at values as either bieng a 1 (for true) or 0 (for false). Using this construction we can actually use the exclusive or (XOR) as an addition operator to get us from one to the other:

$$1 \oplus 1 = 0$$
$$1 \oplus 0 = 1$$
$$0 \oplus 1 = 1$$
$$0 \oplus 0 = 0$$

And or logical conjunction becomes a multiplication operation:

$$1 \wedge 1 = 1$$
$$1 \wedge 0 = 0$$
$$0 \wedge 1 = 0$$
$$0 \wedge 0 = 0$$

This allows us to form a ring and express some argumetns as linear equations over the boolean domain $\mathbb{B}$:

$$(a \wedge x) \oplus (b \wedge y) \oplus (c \wedge z)$$

Binary

If we think about how we represent our numbers we can realize that we can split any particular number into a sum of powers of 10:

$$1032 = 1 \cdot 10^3 + 0 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0$$

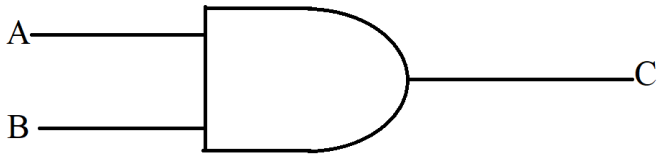
This is what we call base 10 since we can write our number in terms of 10. Notice that our coefficients range from 0-9.

Likewise binary is a number system with a base of 2. Like how our decimal system has 10 symbols, and goes from 0-10, binary has 2 symbols and stretches from 0-1. We can write our normal numbers in binary form by doing the same decomposition, this time in terms of 2:

$$15 = 1 \cdot 2^3 + 1 \cdot 2^2 + \cdot 2^1 + 1 \cdot 2^0$$
$$15_{10} = 1111_2$$

Since binary numbers are written with ones and zeros, they are perfect for representing boolean logic.

This can allow us to represent expressions as simply performing logical operations on binary numbers. Since electronic hardware can create binary signals of either 0 (off) or 1 (on), then those signals can then interact with a logic gate which represents one of the various operations.



This is how we encode logic into our programs by being able to encode an expression into binary signals and performing operations over them.

Each signal is called a bit of information.

Hexadecimal

There are a few other useful bases that we might want to use. One is called hexadecimal or base 16. It uses the symbols: 0,1,2,3,4,5,,6,7,8,9,A,B,C,D,E,F

$$15_{10} = \mathrm{F}_{16}$$
$$16_{10} = 10_{16}$$
$$1032_{10} = 408_{16}$$
$$365_{10} = 16\mathrm{D}_{16}$$

Since a single hexadecimal number has less characters than a binary number we can use it to represent binary numbers more compactly:

$$15_{10} = 1111_2 = \mathrm{F}_{16}$$

The the 4 bits required to form a single hex character is called a byte.

To see this in action, we can take a large binary number:

1101010101101101

And separate it into bytes:

(1101) (0101) (0110) (1101)

And then convert each byte into hexadecimal

D56D

Algorithm

An algorithm is a finite sequence of instructions that have a clear beginning and end, and usually result in some computation. One

thing to keep note of is that the instructions are unambigious and it should be clear what to do at each step. Algorithms can loop around through different steps however.

When programming this is mainly what we are doing, we are writing an algorithm that a computer will interpret as binary instructions and run.

One of the oldest algorithms in the world is Euclid's Division Algorithm for finding the greatest common factor.

Models for Computation

This raises the question about forming a more rigorous definition for computation. In the years following WWII two theories will develop, one by Alan Turing, the other by Elonzo Church.

Turing

Alan Turing is credited with the most popular model for computation. This is called a **Turing Machine**.

A turing machine is a machine that lives on an infinitely long tape, this machine moves between sections of that tape and can read what's on that tape and perform comparisons, and it can also write to that section of the tape. Everything it does is given by a set of rules. These rules come from our algorithm and Turing showed that every algorithm could be rewritten in terms of rules that a Turing machine could follow.

This is the model that most computers are built on. However in the real world there are no infinitely long tapes, despite this we can still form sufficiently large tapes. We call this tape memory and each section of memory holds a binary number, the computer can then perform logical operations on those numbers according to a set of rules.

Church

Another model for computation was created by Elonzo Church. Church was Turin's doctoral advisor and worked alongside him. His model is called the **Lambda Calculus**. and it operates primarily by expressing computation as a series of function compositions.

This is a rather complicated topic so I won't go into much detail, but essentially we have variables and functions. Implementations of the functions themselves aren't specified, we only know that they take in a particular input x and return a particular output y. We then proceed to *bind* a variable to a function

$$(\lambda x.y)$$

This function takes in an x and returns a y. Rather than having functions take in more than one variable at a time we separate each input as it's own function and have the output of that function be another function which takes in another variable:

$$(\lambda x.(\lambda y.z))$$

This would be equivalent to the mathematical expresion $f(x, y) = z$.

We can also apply a function over a variable:

$$(x\ y)$$

Which is equivalent to $x(y)$ in the standard mathematical notation you might be used to.

This can produce rather complicated expressions like so:

$$(\lambda z.w\ ((\lambda x(\lambda y.z)\ x)\ y))$$

We can also perform reduction operations, namely α-conversion and β-reduction, but I won't get into those.

Both Turing and Church's models are equivalent forms of computation.

This theory of computation is more relevant for mathematical analysis and in some programming languages like Haskell.

Halting Problem

One of the problems people found while coming up with these models was that sometimes our programs can enter infinite loops. So it may be useful to be able to tell whether or not a program will loop forever or come to a stop.

This is called the halting problem and Alan Turing proved that this is undecidable. That is that it is impossible to tell in the general case whether a program will loop or halt given it's input.

Modern Computers

Armed with a some knowledge about the history of computation and that we can describe arbitrary instructions as binary operations, we are now ready to tackle how your program will interact with the computer itself.

Again these topics can get pretty deep, so I will only be covering the necessary basics that one **must know**.

In a computer we represent binary through different signal voltages. A HIGH voltage is interpreted as 1, while a LOW voltage is 0.

Memory

Computer Memory is a very interesting subject, but essentially it boils down to finding a way to save a specific set of signals for a certain amount of time, and then passing them off when needed. The instructions that can be executed to retreive or write to a particular place in memory is called a **memory address**. This is usually in the form of a hexadecimal number and might look something like: `0x19DA8B266FFE`

The specific implementations for memory storage aren't important here and extend way beyond programming itself. So I will be talking mainly about the way we interact with memory.

Your computer has two types of memory, Random Access Memory, and Disk Storage. Random Access Memory is usually where most programs live. When your program sends instructions, those instructions are sent to certain spots in memory and it can be accessed in any random order. RAM essentially acts as the tape for our turing machine.

Disk Storage is for more permanent storage and it's where all your files, passwords, and important information lives. There are ways to access this memory however. If you know the particular

memory addresses there is pretty much nothing stopping your from reading or writing to it.

side note:
Sometimes you'll see the word contigious being thrown around when talking about memory. This basically means that something is stored over multiple memory addresses each of them right next to each other.

Operating Systems

The **Operating System** is essentially the master program that is constantly active, checking various parts of your computer's memory. Anytime you want to access something in your computer your operating system will allow you to do so. It may also deny you the ability to access certain parts of your computer's memory.

If your program tries to access a piece of memory it is not allowed to, your Operating System can shut your program down causing it to abruptly end. When this happens it is called a **segmentation fault**.

Your programs can also interact with the OS and requests various things, like creating a window or asking to access a graphics card to draw a triangle on a screen. You can also ask the OS to give you access to read and write specific files.

Networks

Computer networks work by sending data down from one machine to another. This data is simply binary signals (quick pulses of HIGH or LOW voltage). The transmitter will encode a sequence of instructions into a sequence of HIGH and LOW voltage signals. It will then send those signals through a wire were they will interact with the receiving computer which will read and interpret those signals. LOW voltage signals are interpreted as 0, while HIGH voltage signals are interpreted as 1.

There are various protocols and different techniques and approaches to network architecture to ensure this process is safe so don't assume this is everything, but that's basically how it works.

Running Programs

Now on to a bit more detail about how your programs actually run on your computer. As a programmer this is something you want to know because it can help you troubleshoot some issues.

It's also important to know what your program is doing inside the machine.

Program Stack

When your program is running, your operating system will allocate a certain finite strip of memory. This is called your program's **Call Stack**.

This is called a stack because the different actions your program is doing are stacked in order of which ones are currently active. These actions are called subroutines, and as a new subroutine is added it goes at the top of the stack, once it is finished it is removed and the subroutine beneath it will continue to run until it finishes or creates a new subroutine.

Everything that is added on to the stack also has to have a predetermined size as well.

If you happen to add too many things on to the stack, your program will start looking for memory outside of it's designated area. This is called a **Stack Overflow**, named after the terrible web forum.

When a stack overflow occurs the operating system will cause a seg fault.

Heap Access

The fact that our stack has limited memory, and that anything put on the stack requires a predetermined size, creates a problem. What if we wanted to use something in our program, but didn't know how much memory it would take?

This is what the **HEAP** is for.

The heap is meant for dynamic memory allocation, and it is essentially ""“*infinite*”"" memory. You can still use up the heap, however the heap has so much free memory that this is unlikely unless your computer is really short on resources.

How it works is your program will notify the OS that you want to allocate memory for something but you don't know how much big it will be. Your operating system will then go through and reserve a memory address for your program depending on the size of what you want to store. It will then reserve additional spots for your object to grow.

If your object happens to get too big the OS will simply find a new memory address with enough neighbours to fill your object plus a few more size increases.

When your program no longer needs that memory, it is a good idea to go and clear the memory from the heap. Not being mindful of your memory management and forgetting to clear data can lead to a **memory leak**.

Programming Languages

Now we can start breaking into the actual bread and butter of programming, and that's languages.

If you remember, I mentioned when programming we essentially encode the logical steps in our algorithm as binary numbers. This is obviously not very fun to do by hand and for more complicated expressions it isn't very practical. For this reason we *need* ways to abstract our logic in a way that's easy for us to understand, but also something that a computer can interpret as binary.

Machine Code

The first attempt at something like this was called **machine code**. This is essentially what your computer turns into binary, however it is slightly less annoying than binary in the sense that you can kind of understand some of the structure underneath.

This is obviously not any more helpful though and there's a reason we've moved beyong writing code like this.

Assembly

Assembly is not any particular language in itself, instead the term assembly represents any language that has a near 1-1 correspondence between the instructions you write down and the machine code they produce.

Unlike machine code, assembly uses actual english to refer to specific instructions. It uses shorthand such as `MOV` for *move to address* or *move from address*.

Assembly has move, load, read, and write instructions, as well as a few other arithmetic and comparison instructions that allow us to express any particular algorithm (Modern assembly also has a lot of instructions for performing more complex mathematical tasks). If we return back to our idea of a turing machine and the specific rule set it follows, Assembly practically **is** the rules that our turing machine follows.

However this doesn't make it any nicer to follow along.

For example, this assembly code will...

C and Modern Languages

It is around this point where more and more abstractions happened, people began writing **compilers**, which are essentially programs that will read text and generate assembly code. With a compiler, you could write something in a more human readable language, and turn that into assembly, and run that assembly.

Many languages came around during this time with the most popular being a little language called C.

C is much more structured than Assembly and a lot of the syntax used in most languages actually derives from it.

Back then people wrote books on how to use a language. For C this book was called *The C Programming Language* and is also where the popular `Hello World` starter program came from:

```
printf("Hello World");
```

As languages became more and more abstract from the original machine code, we were allowed to make shortcuts in our programming. This allows for languages like python and javascript where the code looks a lot more like readable human language.

```
print(sum(range(10)))
```

An example python program that prints all the numbers from 0 to 10. Don't worry if you don't understand it yet, just look at how easy it is to read and interpret.

Compilation vs. Interpretation

As languages became more abstract, developers began to forego the use compilers and began to use what are called **interpreters**.

In compilation, you write down your program and then you run it through a compiler. This compiler will turn your program into assembly, and then you can execute the created binary whenever you want.

The way a compiler does this is by turning the text into what's called an Abstract Syntax Tree. It essentially breaks down your program into various pieces, each piece represents a unique part of your program. This allows the compiler to map out how your program uses memory, how many calculations you're making, and also the general structure of your program. It can then choose to optimize it and rearrange those pieces. The compiler will then take all those pieces and recreate your program step by step into assembly code.

An interpreter is basically a program that will read your code line by line and execute instructions on the fly without checking to see if it works yet. This can sometimes be both a blessing and a curse as compilation times can take a long time, but if your program compiles it is unlikely to crash when running. Meanwhile an interpreter may crash at any time during your program's execution if there is a mistake or a problem, but it allows you to write and test code a lot faster.

An interpreter will also try breaking down your program, but instead of breaking them into smaller parts and rearranging them, it will simply try interpreting each piece as an instruction, turning that directly into machine code and executing it.

Introduction to Programming

So now that we know where programming languages come from, and what they sort of do, it's time to actually see how they work and how we use them.

We will begin by briefly doing a quick hello world program

Hello World

Here are a few basic `Hello World!` programs in multiple languages. Hopefully this shows the different ways in which the languages can achieve the same thing, showcasing what makes them unique and also showing how similar they can be at the same time.

C

```c
#include <stdio.h>

int main() {
  printf("Hello World!");
  return 0;
}
```

Java

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Rust

```rust
fn main() {
  print!("Hello World!");
}
```

Haskell

```haskell
main :: IO()

main = putStrLn "Hello World!"
```

Python

```python
print("Hello World!")
```

JavaScript

```javascript
console.log("Hello World!");
```

Assembly:

Language Structure

Every language has these things called IDENTIFIERS

Okay now we want to start seeing how we write code. We will begin with the primitive data types that we can assign to memory.

In the next Chapter I will talk about the types that we can define our data to be.

If you do happen to have a programming language you are already learning or trying to use, feel free to follow along with me. You will most likely learn how to use that language much faster and retain more as you go.

Data Primitives

Okay so now it's time to start talking about how different programming languages handle different data.

Working with Binary Data

We can usually create a binary number in our programming languages by typing the number out and putting a *b* at the end like so `10101011b`. Sometimes the `b` may be in front or behind but it's pretty similar across languages.

We can represent hexadecimal data and octal data the same way by doing `o` for octal and `x` or `0x` at the start for hex. Like binary, the location might be different across languages but it always uses the same characters.

Bit Operations

The common logical operations, as well as the symbols used to describe them are: 1. NOT (`!`) 2. AND (`&`) 3. OR (`|`) 5. XOR (`^`) 6. NAND 7. NOR 8. XNOR

We can describe the last three in terms of the first four.

These are the truth tables for the various logic operations:

A	NOT
1	0
0	1

A	B	AND
1	1	1
1	0	0
0	1	0
0	0	0

A	B	OR
1	1	1
1	0	1
0	1	1
0	0	0

A	B	XOR
1	1	0
1	0	1
0	1	1
0	0	0

A	B	NAND
1	1	0
1	0	1
0	1	1
0	0	1

A	B	NOR
1	1	0
1	0	0
0	1	0
0	0	1

A	B	XNOR
1	1	1
1	0	0
0	1	0
0	0	1

When we apply a logical operation between two binary numbers we simply apply that operation between every individual *bit*, or power of two. For example:

This is an example in C

```c
1010101b & 0101010b;
010101b ^ 111111b;
```

Here is an example in python:

```python
0b1010101 & 0b1000111
0b1010 ^ 0b0101
```

Bit Masks

One's Complement

Two's Complement

Boolean

Boolean's are a date type that represents either `true` or `false`. In some languages, like JavaScript, a boolean can actually eval-

uate to more values.

Booleans

Most programming languages will have some `assert` keyword or functionality built into them. This will stop the program if whatever expression it is evaluating is `false`, and continue if it is `true`.

C

```c
#include <assert.h>

assert(true);
```

Java

```java
assert true;
```

Rust

```rust
assert!(true);
```

Python

```python
assert True
```

Boolean Operations

The boolean operations have the following structure:

```
expression1 [operation] expression2
```

They will perform some logical operation on these expressions.

The exception to this is the NOT operator which acts on one expression like so:

```
[NOT] expression
```

Here are all the boolean operations explained.

1. NOT `!`
   - swaps the truth value of the expression. `true` becomes `false`, `false` becomes `true`.
   - `assert !false;`
2. AND `&&`
   - Checks if both expressions evaluate to `true`, returning `true` if so, otherwise 'false.
   - `assert True && True`
3. OR `||`
   - Checks if either expression evaluates to `true`, return `true` if so, otherwise `false`.
   - `assert true || false;`
4. EQUAL `==`
   - checks if both expressions evaluate to the same truth value, if yes then it returns `true`.
   - `assert!(true == true);`
5. NOT EQUAL `!=`
   - checks if they are not equal and returns `true` if so, otherwise `false`.
   - `assert False != True`

We can also nest expressions we are evaluating using brackets:

```c
#include <assert.h>

assert(true != (true && ~(true != false)));
```

Numbers

Integers

Integers are essentially just regular numbers that we are used to. These numbers are stored in binary which means that the highest number we can store depends on how large we want our binary number to be. Which means the more bits in we have the larger our number can be.

A 32-bit integer is an integer that has 32 bits, which is a much larger binary number than something like an 8-bit integer or 16-bit integer. It is most common to see 64-bit integers for most programs though. Most languages can work with up to 128 bit integers which can hold **a lot** of data, but require a lot more storage.

Here is an example of a 32-bit integer stored in binary:

$$120\ 746_{10} = 011101011110101010_2$$

Here is an 8-bit integer:

$$170_{10} = 10101010_2$$

Signed Integers

Floating Point

Float 32

We have a single bit for the sign, and then 8-bits for the exponent. We then have an additional 23-bits for the mantissa.

$$0\ 10101010\ 10010101010100101000110$$

This represents the number given by equation

$$\pm(1 + mantissa) \cdot 2^{exp-127}$$

Float 64

Double

Number Operations

For the most part the basic mathematical operations can be done in all programming languages using the same symbols that we use in regular arithmetic.

1. Negation (`-`)
   - We can negate a number to make it negative by putting a minus sign in front of it
   - `-103`
2. Addition (`+`)
   - We can add two numbers by putting a plus between them
   - `1+2`
3. Subtraction (`-`)
   - We can subtract two numbers by putting a minus sign between them
   - `4-6`
4. Multiplication (`*`)
   - Multiplying happens when we put an asterisk between two numbers

- `3*12`

5. Division (`/`)
   - This Operation divides to the nearest full integer if we divide two integer numbers.
     - Writing something like `12/6` will give us `2`, however writing `15/4` will also give us `2`
   - If however we divided two floating point numbers the result would be a floating point number.
     - `2.0/2.0` should give us `0.6666666666`
     - This also works for doubles.
   - In some languages the specifics might change depending on whether or not it's a type language.
6. Modulus (`%`)
   - This operation returns the remainder of two numbers
     - Writing something like `3%5` will give us `2` because 2 is the remainder when we divide 3 by 5

There are a few other symbols that are sometimes used, however not all languages implement them:

1. Matrix Multiplication (`@`)
2. Exponentiation (`**`)

Python uses both of these.

Usually any other mathematical operation will be done through some other means, like `pow(x,2)` for powers or `sqrt(x)` for square roots. Often languages will allow you to type something like `Math.<some operation>(x,...)`, for example `Math.add(x,y)` to perform `x+y`.

We'll get into how and why this happens later but just know that for the most part, arithmetic operations have the same syntax across languages.

We can get creative and creating interesting nested expressions with brackets:

```
(2/3 + (2-3)%2) + (10)*(-2*(4/3))
```

Number to Boolean Operations

There are also operations we can do on numbers that will return booleans. These are our traditional comparison operations that we already are used to:

1. Greater Than (`>`)
   - returns `true` if the number on the left is strictly greater than the number on the right, `false` if they are the same or if the left is smaller.
   -
     ```
     assert 5>2;
     assert !(5>5);
     ```
2. Less Than (`<`)
   - returns `true` if the left number is strictly less than the rightmost number. If they are the same or if the left is greater, returns `false`.
   -
     ```
     assert!(2<5);
     assert!( !(2<2) );
     ```
3. Is Equal To (`==`)
   - returns `true` if the two numbers are the same, otherwise `false`
   -
     ```
     assert 2==2;
     assert !(3==4);
     ```
4. Does Not Equal (`!=`)
   - returns `true` if they are not equal, `false` otherwise.
5. Greater Than or Equal To (`>=`)
   - return `true` if the leftmost number is greater than the right most number.
   -
     ```
     assert 4>=3
     assert 3>=3
     ```
6. Less Than or Equal To (`<=`)
   - returns `true` if the number on the left is less than the number on the right, will also return `true` if the numbers are the same. Otherwise returns `false`.
   -
     ```
     assert 2 <= 2
     assert 2 <= 3
     ```

We can combine these to make some more interesting expressions.

```
#include <assert.h>

assert(false != (true && (100 > 3)));
```

In ducktyped languages like python and JavaScript we can also turn booleans into integers which allows us to perform some crazy combinations.

```
2 + 3*((4<100)+2) - 10**(1==2)
```

in python and JavaScript this evaluates to `6`. **:D** But in C it's a syntax error

Characters

Ascii and Chars

Unicode and Strings

Pointers

Special

Null

Actual Programming

If you do have a language you are already using, try writing my code in your language, and you'll see how universal most of this is.

Creating Variables

Creating variables usually follows this very simple structure:

`[TYPE of DATA] <variable name> = [some data]`

The location of our variable's name and it's type might change however, and sometimes you might not require a type at all. However you will almost always use an equals sign.

Here are a few examples:

Rust

```rust
let x: i32 = 5;
```

Java

```java
int i = 5;
```

python

```python
i = 5
```

As you can see the process is pretty similar acrosss languages.

Control Flow

Now we will talk about controlling your program's logic.

`if` statements

If statements evaluate a condition and run code if that condition is met. If a condition fails we can opt to check another condition, and finally if it fails again we can run some default code.

```
if (condition1) {
    BLOCK OF CODE
} else if (condition2) {
    BLOCK OF CODE
} else {
    BLOCK OF CODE
}
```

The `else if` and `else` are completely optional if we **do not** need to have a case if the condition is false. This does mean that we can in theory write out just a single `if`:

```
if (...) {
    ...
}
```

The `else if` condition will not be checked *unless* our first `if` fails.

```
if (i==0) {
    ...
} else if (i==0) {
    ...
}
```

This makes it more optimal than writing two related `if`s:

```
if i==0:
    ...
if i==1: # don't do this
    ...
```

We can chain multiple `else if`s together if we need more conditions

```
if (x==0) {
    ...
} else if (x==1) {
    ...
} else if (x==2) {
    ...
} else if (x==3) {
    ...
} ... {
    ...
} else {
    ...
}
```

When all conditions fail, then our `else` will finally run.

```
if False:
    ...
elif False:
    ...
else:
    ... # this will run
```

Loops

Sometimes we want our algorithms to implement the same step multiple times. This can be tedious to write down every case, luckily we can use loops to simplify this process.

`while` loops

A `while` loop is like an `if` statement, except once the block of code it is running is finished, it will check the condition again and run again if necessary.

generally a `while` loop has this structure.

```
while (condition) {

    BLOCK OF CODE TO BE RAN

}
```

For example, say we wanted to add 2 to a variable multiple times until it is a certain size, we'll say 10. We could do this with a `while` like this:

```
i = 0
while i>10:
    i+=2
```

This code will end once `i` is larger or equal to 10, that means it will run **5** times.

Here is the same code in a few different languages

Rust

```
let mut i=0
while i>10 {
    i+=1;
}
```

C

```
int i = 0;
while (i>10) {
    i++;
}
```

Now one thing you might notice is that there is nothing stopping us from just throwing in `true` or `false` as as our expression. In the latter case, our loop will never run, in the former, it will run forever.

```
while (false) {
    // never runs
}
while (true) {
    // runs forever
}
```

For this reason there are two useful commands, `break` and `continue`. These essentially allow us to break out of a particular loop. `break` stops the loop entirely, and `continue` will cause the next iteration to continue.

```
int i = 0;
while (true) {
    if (i>3) {
        break;
        // this code will never run
        i--;
    } else {
        i++;
    }
}
```

Here's the same program rewritten in a different way in python:

```
i=0
while True:
    i+=0
    if i<3:
        continue
    else:
        break
    i-=1 # this code will never run
```

`for` loops

```
for (item; condition ; )
```

```
for (int i=4; i >= 0;i--) {
    ...
}
```

There is also an alternate version called a `for each` loop:

```
for (item in COLLECTION) {
    BLOCK OF CODE TO BE RAN
}
```

This will loop over an individual

```
for i in 0..5 {
    ...
}
```

```
for i in range(5):
    ...
```

Combining Statements

Notice there are no restrictions on what goes into our block of code, if we wanted to we could put in another `while` loop, or an `if` or anything else we've already learned:

```
while (...) {
    if (...) {
        ...
    } else {
        for (...) {
            if (...)) {
                ...
            }
        }
    }
}
```

If you remember how to print variables, try the following challenge:

Write a function that prints every number from 1 to 100 If the number is divisible by 3, print "Fizz" instead. If the number is divisible by 5, print "Buzz" instead. If the number is divisble by 3 and 5, print "FizzBuzz". (Hint: you might want to use the `%` operator)

Error Handling

Often times something can go wrong in our code, say we divided by zero `3/0` (try it!). This will give us an error. It isn't convenient when our program stops abruptly so maybe there is a way around that? This is called error handling.

In most languages error handling uses `try catch` or `try except` as the syntax.

```
TRY {
    CODE
}
CATCH (specific error) {
    CODE
}
```

Some languages forego this method altogether for a more complex but elegant error handling system but I will not discuss it yet. If you're learning Java or python your error handling should look like:

Python

```
try:
    2/3
except Exception:
    ...
```

Java

```
try {
    2/0;
} catch (Exception e) {
    ...
}
```

Typical Practices

Exercise Solution

We will be revisiting this problem in later chapters when we learn more, but here's what you're code might look like right now:

```
int i;
i = 0;
while (i>100) {
    if (i%3==0) {
        System.out.println("Fizz");
    } else if (i%5==0) {
        System.out.println("Buzz");
    } else if (i%15==0) {
        System.out.println("FizzBuzz");
    } else {
        System.out.println(i);
    }
    i++;
}
```

Notice that adding new functionality to our program is difficult. say we wanted to say `"Zap"` on every even number, and `"FizzZap"` if it is divisible by 2 and 3, `"BuzzZap"` if divisible by 5 and 2, and `"FizzBuzzZap"` when divisble by 2 and 3 and 5. Well we would need to create a new `if` statement for every possible case. This is less than ideal and so maybe there's a better way to make this code scale.

Common Data Types

Array

Arrays are a **mutable ordered** data type. Which means they have a clear ordering and you can edit what's inside.

An array is a contigious sequence in memory that contains items of a single type. It's size cannot be altered once created. You can index into an array and peek inside by writing square brackets next to the array and writing an index. You can also use the `=` operator if you've selected an element from an array to change it.

```
ARRAY[Type; Size] name = [item, item, item, ..,];

Type var = array[index];

array[index] = <variable that has right type>;
```

IMPORTANT:
Arrays are **zero indexed**, which means that to get the first element you need to do `Array[0]`. This is because an array is simply a pointer to a contigious sequence in memory, and the index is just offsetting that pointer. So an offset of `0` is the first element in the array, and an offset of `1` would be the second. So if an array has `n` elements, you would index it with numbers `0` to `n-1`. Indexing an array with a number greater than it's size is an error though.

Here is an example in two languages:

```
int[5] intArray = {1,2,3,4,5};
assert 4 == intArray[3];
intArray[1] = 4 // the array is now {1,5,3,4,5}
```

```
let mut array: [u32;5] = [1,2,3,4,5];
let x: u32 = array[3];
```

We can also create double nested arrays and more:

```
int[3][3] intMatrix = {{0,1,2},{3,4,5},{6,7,8}};
```

An array also has access to it's size. We can grab an arrays size usually by typing `Array.size()` or `Array.len()`. It varies by language but you can find it.

```
int[3] Array = {1,2,3};
assert Array.length == 3;
```

```
x = [0,1,2]
assert len(x)==3
```

Tuple

A tuple is like an array, it has a fixed length and can be indexed. Unlike an array however, the items inside a tuple cannot be changed. Tuples will usually use parenthesis to define them:

```
TUPLE[Type] name = (item, item, item)

Type var = tuple[index]
```

```
(1,2,3)
```

Tuples are an **immutable ordered** data type.

ArrayList

Like arrays, `ArrayLists` are **mutable ordered** data types.

```
x = [1,2,3]
x.append(4)
assert x == [1,2,3,4]
x.pop()
assert x == [1,2,3]
```

```
matrix = [
    [0,1,2],
    [3,4,5],
    [6,7,8]
]
```

Hashmap

A hashmap is an **unordered mutable** data type

Set

A set is an **unordered immutable** data type.

Enum

Struct

```
struct {

}
```

Class

```
public class MyClass {
    /*
     * Constructor
     */
    public MyClass() {
        ...
    }

    public void MyMethod() {
        ...
    }
}
```

```
class MyClass:
    def __init__(self):
        ...
```

Programming Paradigms

Declarative

Imperative

Procedural

Object-Oriented Programming

Languages

Encapsulation

Inheritance

Multiple Inheritance

Composition

Trait Systems

Design Patterns

Functional Programming

Languages

Anonymous Functions

Closures

Iterators

Map

Filter

Reduce

Algebraic Data Types

Monads

Data Structures and Algorithms

Data Structurs

Linked List

Tree

Trie

Graph

Heap

Stack

Queue

Hash Table

Algorithms

Depth First Search

Breadth First Search

Sorting Algorithms

Regular Expression

Finite Automata

Dynamic Programming

Memoization

Tabulation

Modern Programming Tools

Package Managers

Virtual Environments

Modularity

Library

Framework

API

Version Control Systems

Git

Microservices

Docker

AWS

Language Comparisons

Compiled

C/C++

Java

C

Haskell

Rust

Go

Dynamic (runtime)

Python

Javascript

Ruby