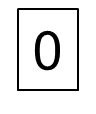
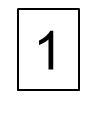
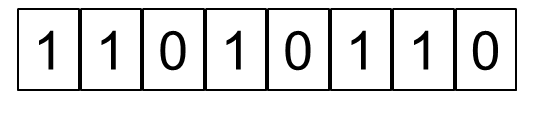
1. C Basics
   1. Computer problems give the processor instructions about how to manipulate data.
   2. How data are stored
      1. **Variables**
   3. Manipulations of data
      1. **Operators**
2. Computer Memory
   1. Very long strings of **bits** – binary elements represented by 0s and 1s.

or ****

* 1. **Byte**: 8 consecutive bits.
     1. Smallest addressable size of memory.
     2. Each has an **address** (way to locate it within that long string).
        1. **Addressable memory**: we can pinpoint a particular location in the long string and store/retrieve values in them
     3. **Low bit**: bit on the rightmost side.
     4. **High bit**: bit on the leftmost side.



* 1. **Word**: 16 consecutive bits.
  2. C allows us to give its address a name with some rules.
     1. Allows users to use **pointers**: way to access memory directly through variables.

1. **Binary Arithmetic**: base 2 represents a string of 0s and 1s.
   1. Can be converted to **decimal** (base-10).
      * 1. Low bit:
        2. High bit:
   2. Converting a byte to decimals
      1. Range = **[0, 255]**
   3. Converting a word to decimals
      1. Same process as a byte, but up to
      2. Range = **[0, 65,535]**
   4. Converting a decimal to binary
      1. Divide by 2
      2. Use carry as the binary value
2. Variables
   1. Example
      1. x + 10
      2. On the surface, x has no intrinsic value.
      3. If we give it a numeric value, the equation can be computed.
         1. x = 5
         2. 5 + 10 = 15
   2. In algebra…
      1. We use symbols to stand in for numeric values that are unknown – variables.
      2. Computers cannot deal with these concepts.
   3. In computers…
      1. Variables are locations in data used to store and retrieve values of data that a program can manipulate.
         1. Computer memory is **allocated** to a variable.
         2. **Mapping** between the **address** (memory location) of the variable and its given symbol is created when a variable is declared.
      2. Setting and changing variable values is fundamental.
   4. Issues
      1. **Legal names/variable identifiers**: certain rules one must follow when choosing names for variables (identifiers).
         1. **Variables**: labels that map to specific locations in memory where a particular value is to be stored.
            1. Programmer does not have to keep track of cryptic memory addresses.
         2. Rules
            1. Only letters, numbers, and \_

**Case-sensitive**

Customary

Lower-case: variables

Upper-case: constants

Compiler will not character

Length of the variable is usually unlimited.

Best to keep it short and meaningful, however.

* + - * 1. First character must either be a letter or \_
        2. Must not be a **keyword**.

C has 37 reserved keywords, which if used as variable names, will confuse the compiler.

Compilation error

Runtime error

Examples

auto

double

int

struct

break

else

long

switch

case

enum

register

typedef

char

extern

return

union

const

float

short

unsigned

continue

for

signed

void

default

goto

sizeof

volatile

do

if

while

statistic

\_Boolinline

\_Complex

restrict

\_Imaginary

* + 1. **Type**
       1. Some variables require more memory space than others.
          1. Rather than allocating an equal block of memory to accommodate the largest of variables, C makes the programmer declare the type of variable.
          2. C therefore…

Memory-efficient

**Heavily-typed**: you must decide the type of variable something is.

* + - 1. It is important for variables to be **declared** before their use in the code.
         1. Different types require different amounts of memory, and determine how much is allocated.
      2. Common variable types
         1. **Integers (int)**: natural numbers.

32 bits (4 bytes).

Typically used to count whole things.

Iterations in loops

Elements of an array

Some special functions

Not good for math in most applications.

As long as you do some of the calculations with integers, not all.

Traditionally represented by…

1 byte: 0-255

1 word: 0-65,535.

Cheap availability of memory now; most are at least 16 or 32 bits.

There are also 64-bit integer values that can be declared.

32 bits: can store values from -231 to 231-1.

**Signed integers**: high bit is used to indicate the sign of the number.

C allows bit-level manipulation

Used for reversing the sign of a signed integer.

*int* defaults to signed integers.

0: +

1: -

Range

Byte: [-128, 127]

Word: [-32,768, 32,767]

**Unsigned integers**: declared by *unsigned int*.

No standard limit to the size.

*short int*: 16 bits, signed

*unsigned short int*: 16 bits, unsigned

*unsigned int*: 32 bits, unsigned

*int*: 32 bits, signed

*long int*: 64 bits, signed

*long long int*: 128 bits

* + - * 1. **Characters (char)**: non-numeric characters.

Mapped onto characters through the ASCII.

In the ASCII table, each character takes up 1 byte.

Treated as **8-bit signed integers**.

Only some represent printable characters.

32/126 are normal characters

There is an extended ASCII table for more symbols.

Store non-computable integer values.

When defined, the processor will not interpret as integer, but will look up equivalent in ASCII table.

Will not store multiple characters, just one.

Multiple are a *string*.

Characters enclosed in ‘’.

* + - * 1. **Floating point (float)**: real numbers.

Store real numbers used in arithmetic calculations.

Represent the decimal point as separator between wholes and fractions in a real numbers.

2 numbers separated by a decimal point.

What we traditionally think of as number.

Accurate to 6 or 7 digits.

* + - * 1. **Double-precision floating point (double)**: high-precision real numbers.

Accurate to 13 digits

Can store larger numbers.

* + - 1. **Complex data structures**: permit more sophisticated ways to store and manipulate data.
         1. **Arrays**: collection of similar variables.
         2. **Strings**: sequence of characters.
         3. **Structures**: collection of dissimilar variables.
         4. **Unions**: collection of dissimilar variables sharing the same addresses.
    1. **Declaring variable**: to use a variable, it must be declared ahead of time.
       1. **Declaration**: statement that tells the compiler that the programmer wants to have a variable of a particular type, with a particular name.
          1. Requests allocation of memory of the appropriate size.
          2. Requests an **index** (map) of its address to the name of the variable.
       2. A number can be **written** (assigned) and **read** (retrieved), as well as updated later.
          1. Its value can be accessed by referring to it by its name.
       3. **Main function**
          1. main()
          2. {
          3. Source code;
          4. }
          5. return(0)

Functions return values which is why main() is declared as an integer.

Meaning, the main() function will return an integer value.

* + - 1. Declaring variables
         1. char var\_name;
         2. float var\_name;
         3. int var\_name;
         4. Or, you can declare multiple variables of the same type

int i, j, n;

* + - * 1. Or, variables can be **initialized** (set to a starting value).

int var\_name = 30, n = 10, x, y;

var\_name is initialized to 30

n is initialized to 10

x and y are not initialized

Optional, but highly recommended.

Sometimes not initializing can cause errors.

If you don’t know what value to set them to, set them to 0.

* + - 1. Using variables
         1. All variables must first be declared before being used.
         2. Usually at the top of the function.

Usually main(), but later user-defined functions.

**Compilation error**: occurs only if a variable is used before it is declared.

* + - * 1. Different from different languages

LISP

Python

* + 1. **Scope**: extent of validity of a variable with regards to parts of the program.
       1. **Local variables**: only recognized within the block of code (i.e., inside a function) in which they are defined.
          1. The system later forgets once the block of code exits.
          2. Most common type
          3. Memory that had been allocated automatically returns to the “free memory stack” for use by the program later when needed.
          4. Strongly preferred over global variables.

Less dangerous because of reduced risk of another part of the program changing it unintentionally.

Always use them in this course.

* + - 1. **Global variables**: valid everywhere in a single-file program**.**
         1. Should be defined outside of any specific block of the code.

e.g., the beginning of the source code.

* + - * 1. There are situations in which they are useful.
  1. Last word on variables
     1. Many other variable types exist.
     2. C allows the programmer…
        1. To define a variable *type*.
        2. **Casting**: temporarily changing a variable type.

1. **Constants**: values that have a special meaning but will not change throughout the program, but can be changed easily through the source code.
   1. Definition
      1. *#define* pre-processor directive
         1. #define PI 3.1415926535
         2. = is not used.
      2. Variables
         1. Set to a fixed value that will not be purposefully changed throughout the program.
         2. Higher computational cost
            1. Takes more memory.
            2. Overhead involved in indexing the variable and retrieving it from memory.
2. **Operators**: perform basic functions on data.
   1. Runtime error if a variable has no assigned value.
   2. Spaces not required, but best for ease of reading.
   3. C has several primitive ones defined in the language.
      1. **Assignment = operator**: assigns a value to a variable.
         1. Example
            1. sum = 17 + 2 + 6 + 12;
         2. Most important and extensively used operator.
         3. Functions can return values later.
            1. main()
            2. You can use the assignment operator in a function to return a variable defined by a value.
      2. **Equality == operator**: Boolean operator used to test for equality.
         1. Checks if x is equal to b by being written as x == b
            1. If yes 🡪 True
            2. If no 🡪 False
      3. **Mathematical operators**
         1. Examples
            1. +
            2. –
            3. \*
            4. /
         2. All of them (except /) apply identically to…
            1. float
            2. double
            3. int
         3. Division
            1. Floating Point Division

For floating points a and b…

a / b

Result: floating point value assignable to a variable of the corresponding type.

float

double

NOT an int

* + - * 1. Integer Division

2 types of division

/: returns whole numbers by truncating everything after the decimal place, and omitting the remainder

11/4 = 2

%: mod operator; returns only the remainder in division.

11%4 = 3

Result < Denominator

* + - 1. Arithmetic Operator Precedence
         1. Order of operations by default

Left 🡪 Right

Anything within () evaluated first

\* and / have equal precedence

+ and – have equal precedence, but less than \* and /

* + - * 1. Example

3 – 4 \* 5 + 6 = 3 – 20 + 6 = -11

* + - 1. **Boolean Operators**: check if a Boolean value is true or false.
         1. Developed by George Bool
         2. In C…

0 = false

Any non-0 = true

* + - * 1. 3 Boolean operators that take Boolean values as operands.

AND(&&): disjunction; true if and only if all arguments are true.

OR (||): true if at least one argument is true.

NOT (!): inverts the truth value of its only argument.

* + - * 1. Frequently expressed with truth tables

|  |  |
| --- | --- |
| **X** | **! x** |
| **True** | **False** |
| **False** | **True** |

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **x && y** |
| **True** | **True** | **True** |
| **True** | **False** | **False** |
| **False** | **True** | **False** |
| **False** | **False** | **False** |

|  |  |  |
| --- | --- | --- |
| **X** | **Y** | **x && y** |
| **True** | **True** | **True** |
| **True** | **False** | **False** |
| **False** | **True** | **False** |
| **False** | **False** | **False** |

* + - * 1. Others

==

=>

<=

>

<

**Inequality operator** (!)

* + - 1. **I/O Operators**
         1. printf(): prints to screen.

Argument

“String”,

%: indicates what you want to print out.

**Format string**: specifies variable type.

int: %d

char: %c

float: %f

double: %lf

That which follows “”,

Label of variable to be printed.

You can print multiple variables on the same line.

**Literals**: just as stated.

Hello world!

Escaping

\: **escape character**

Tells compiler to not print the subsequent letter, but that it has a special meaning.

\n: to go to the next line.

Can be used to print value of variables and even compute an equation.

Not the only one.

* + - * 1. scanf(): provides basic operator to read from keyboard entries by the user.

Similar to printf(), but…

Does not accept literals.

Does not print out anything

Pauses execution to await keyboard entry.

Value inputted is stored in a variable or constant.

scanf(“%d”, &x);

%d: format string.

Same specifications as printf().

&: telling the location of x.

x: tells the value of x.

* + - 1. Shorthand assignment
         1. **Increment/Decrement unary operators**: change values by 1.

Types

++

Pre

Post

–

Pre

Post

Examples

main()

{

int c=5;

printf(“%d ”, c);

printf(“%d ”, c++);

printf(“%d ”, c);

}

5 5 6

main()

{

int c=5;

printf(“%d ”, c);

printf(“%d ”, ++c);

printf(“%d ”, c);

}

5 6 6

|  |  |
| --- | --- |
| x += 5; | x = x + 5;   1. Equation is evaluated at the right.    1. 5 is added to the original value of x. 2. x is then assigned the sum of 5 and the original value of x. |
| y \*= 3; | y = y \* 3; |
| z -= 2; | z = z – 2 |
| w++  **Post-increment operator** | w = w + 1 |
| v—  **Post-decrement operator** | v = v – 1 |
| ++a  **Pre-increment** | Increments a by 1 before it is used in the expression |
| --a  **Pre-decrement** | Decrements a by 1 before it is used in the expression. |

1. Elementary Strings
   1. **Strings**: arrays of characters.
      1. **String literals**: string enclosed in quotation marks.
      2. Can be read from user using scan(f)
      3. Can be printed using printf()
   2. Declaration: char str [128]
      1. str: name of string variable selected by programmer.
      2. 128: maximum size of string
         1. Includes **null terminator**: special character which indicates end of a string.
   3. Reading and printing
      1. scanf(“%s”, str);
         1. No & before str
            1. Because strings are arrays.
      2. printf(“%s”, str);