# Fundamentals

> *"Anyone can write code which a machine can understand - the trick is to write code which another human can understand." -- Martin Fowler*

## Introduction

- What is computer programming?
- What is a computer program?
- What is a programming language?
- Differences between languages (Currently popular languages Don't take it too seriously!)
- Compilers vs. Interpreters
- Creating a program
  1. Editing source files
  2. Compiling source files into object files
  3. Linking object files into executable files
- Executing the program
- Central Processing Unit (CPU)
  - Arithmetic and Logic Units
  - Registers
  - Cache memory
- Main memory

## First C Programs (The Chicken and The Egg)

- A C program is essentially a collection of one or more functions.
- There must be a function named `main`; it must be in all lowercase.
- `main` is the program's starting point; it can call other functions by name
- There must be *exactly* one function named `main` in every program.

The general form of a C program looks like this: (The parts in **bold** are required)

```
include files

function declarations (prototypes)

data declarations (global)

main function header
{
  data declarations (local)
  statements
}

other functions
```

Therefore, the simplest C program you can write:

```c
int main(void)
{
}
```

Technically, you should have a `return` statement:

```c
int main(void)
{
  return 0;
}
```

It looks simple because it is. It does nothing of interest. But, nevertheless, it produces a "functional" program. This simple program demonstrates many characteristics of a C program.

Students that think that `int main(void)` can be changed to `int main()` are required to read this, as I'm not going to spend any more time on it. (Yes, I know that our textbook interchanges them, but you shouldn't.)

A second program in C that actually does something:

```
#include <stdio.h>

/* Say hi to the world */
int main(void)
{
  printf("Hello, World!\n");
  return 0;
}
```

**Output:**

```
Hello, World!
```

Adding line numbers for clarity. They are not (and cannot be) present in the actual code.

```
#include <stdio.h>

/* Say hi to the world */
int main(void)
{
  printf("Hello, World!\n");
  return 0;
}
```

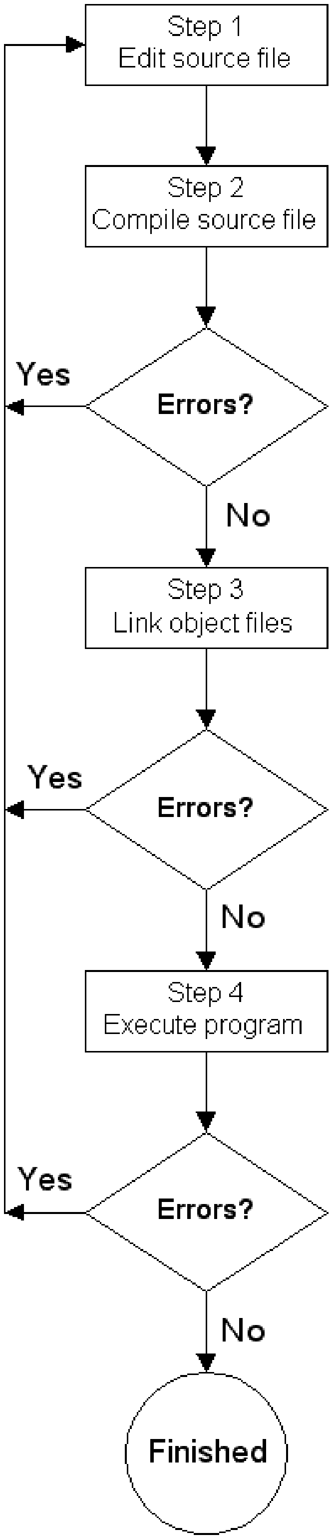
Each non-blank line above has significant meaning to the C compiler. Look at stdio.h to see what the pre-processor adds.

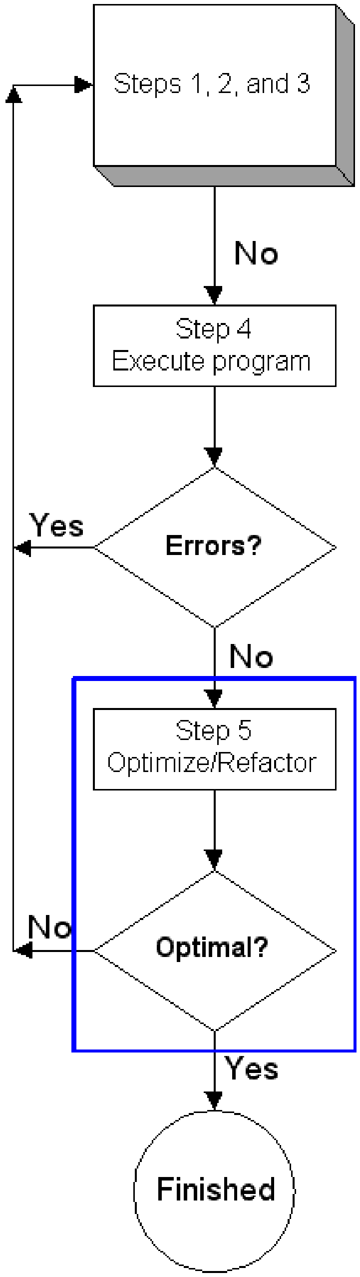
---

## Editing, Compiling, Linking, and Executing

**Edit/Compile/Execute Loop**

**Edit/Compile/Execute Loop Extended**




## Contrasting Languages at 3 Levels

Let's say we have 4 variables (just like in algebra), where *a = 1, b = 2, c = 3, d = 4*. We want to set a fifth one, *e*, to *ab(c + d)*. We can rewrite the multiplication explicitly as

$a \times b \times (c + d)$

This will result in *e* having the value *14*.

### C code (partial program)

```
int a = 1;
int b = 2;
int c = 3;
int d = 4;
int e = a * b * (c + d);
```

---

## Assembly language (Intel x86)

<table>
<tr><th>Assembly language (compiler-generated)</th><th>Assembly language (with comments)</th></tr>
<tr><td>

```
_main proc  far
   mov word ptr [bp-2],1
   mov word ptr [bp-4],2
   mov word ptr [bp-6],3
   mov word ptr [bp-8],4
   mov ax,word ptr [bp-2]
   imul  word ptr [bp-4]
   mov dx,word ptr [bp-6]
   add dx,word ptr [bp-8]
   imul  dx
   mov word ptr [bp-10],ax
_main endp
```

</td><td>

```
_main proc  far
   mov word ptr [bp-2],1  ;the address of a is bp-2
                          ;the value is 1

   mov word ptr [bp-4],2  ;the address of b is bp-4
                          ;the value is 2

   mov word ptr [bp-6],3  ;the address of c is bp-6
                          ;the value is 3

   mov word ptr [bp-8],4  ;the address of d is bp-8
                          ;the value is 4

   mov ax,word ptr [bp-2] ;put a's value in ax reg

   imul  word ptr [bp-4]  ;multiply ax reg by b,
                          ;put result back in ax

   mov dx,word ptr [bp-6] ;put c's value in dx reg

   add dx,word ptr [bp-8] ;add d to the dx reg
                          ;put result back in dx

   imul  dx               ;multiply ax reg by dx
                          ;put result back in ax

   mov word ptr [bp-10],ax ;the address of e is bp-10
                           ;the value of e is 14
_main endp
```

</td></tr>
</table>

Hand-coded assembler (80386 using GNU's assembler, comments start with # and are in **bold**)

```
.section .data
a: .long 1
b: .long 2
c: .long 3
d: .long 4
e: .long

.section .text
.globl _start

_start:
  movl a, %eax      #     a -> %eax
  movl b, %ebx      #     b -> %ebx
  imull %eax, %ebx  # a * b -> %ebx

  movl c, %eax      #     c -> %eax
  movl d, %ecx      #     d -> %ecx
  addl %eax, %ecx   # c + d -> %ecx

  imull %ebx, %ecx  # (a * b) * (c + d) -> %ecx
  movl %ecx, e      # put result in e
```

Here are two other assembler samples output from Microsoft's compiler and GNU's compiler.

---

## Machine language (Intel 80386)

| Hexadecimal dump | Octal dump |
|---|---|
| 014c 0003 0000 0000 0110 0000 000c 0000 | 000514 000003 000000 000000 000420 000000 000014 000000 |

```
0000 0104 742e 7865 0074 0000 0000 0000
0000 0000 0070 0000 008c 0000 00fc 0000
0000 0000 0002 0000 0020 6000 642e 7461
0061 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0040 c000 622e 7373 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0080 c000 8955 83e5
18ec e483 b8f0 0000 0000 c083 830f 0fc0
e8c1 c104 04e0 4589 8be8 e845 00e8 0000
e800 0000 0000 45c7 01fc 0000 c700 f845
0002 0000 45c7 03f4 0000 c700 f045 0004
0000 458b 89fc 0fc2 55af 8bf8 f045 4503
0ff4 c2af 4589 b8ec 0000 0000 c3c9 9090
9090 9090 9090 9090 9090 9090 0021 0000
000b 0000 0014 0026 0000 0009 0000 0014
662e 6c69 0065 0000 0000 0000 fffe 0000
0167 6973 706d 656c 632e 0000 0000 0000
0000 0000 6d5f 6961 006e 0000 0000 0000
0001 0020 0002 742e 7865 0074 0000 0000
0000 0001 0000 0103 0062 0000 0002 0000
0000 0000 0000 0000 0000 642e 7461 0061
0000 0000 0000 0002 0000 0103 0000 0000
0000 0000 0000 0000 0000 0000 0000 622e
7373 0000 0000 0000 0000 0003 0000 0103
0000 0000 0000 0000 0000 0000 0000 0000
0000 5f5f 6d5f 6961 006e 0000 0000 0000
0020 0102 0000 0000 0000 0000 0000 0000
0000 0000 0000 5f5f 6c61 6f6c 6163 0000
0000 0000 0000 0002 0004 0000
```

```
000000 000404 072056 074145 000164 000000 000000 000000
000000 000000 000160 000000 000214 000000 000374 000000
000000 000000 000002 000000 000040 060000 062056 072141
000141 000000 000000 000000 000000 000000 000000 000000
000000 000000 000000 000000 000000 000000 000000 000000
000100 140000 061056 071563 000000 000000 000000 000000
000000 000000 000000 000000 000000 000000 000000 000000
000000 000000 000000 000000 000200 140000 104525 101745
014354 162203 134360 000000 000000 140203 101417 007700
164301 140404 002340 042611 105750 164105 000350 000000
164000 000000 000000 042707 000774 000000 143400 174105
000002 000000 042707 001764 000000 143400 170105 000004
000000 042613 104774 007702 052657 105770 170105 042403
007764 141257 042611 134354 000000 000000 141711 110220
110220 110220 110220 110220 110220 110220 000041 000000
000013 000000 000024 000046 000000 000011 000000 000024
063056 066151 000145 000000 000000 000000 177776 000000
000547 064563 070155 062554 061456 000000 000000 000000
000000 000000 066537 064541 000156 000000 000000 000000
000001 000040 000002 072056 074145 000164 000000 000000
000000 000001 000000 000403 000142 000000 000002 000000
000000 000000 000000 000000 000000 062056 072141 000141
000000 000000 000000 000002 000000 000403 000000 000000
000000 000000 000000 000000 000000 000000 000000 061056
071563 000000 000000 000000 000000 000003 000000 000403
000000 000000 000000 000000 000000 000000 000000 000000
000000 057537 066537 064541 000156 000000 000000 000000
000040 000402 000000 000000 000000 000000 000000 000000
000000 000000 000000 057537 066141 067554 060543 000000
000000 000000 000000 000002 000004 000000
```

## Binary dump

```
0100110000000001000000110000000000000000000000000000000000000000
0001000000000001000000000000000000000011000000000000000000000000
0000000000000000000000100000000010010111001110100011001010111000
0111010000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000011100000000000000000000000000000
1000110000000000000000000000000011111000000000000000000000000000
0000000000000000000000000000000000010000000000000000000000000000
0010000000000000000000001100000001011100110010001100001011101000
0110000100000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0100000000000000000000001100000000101110011000100111001101110011
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1000000000000000000000001100000001010101100010011110010110000011
1110110000011000100000111110010011110000101110000000000000000000
0000000000000000100000111100000000000111110000011110000000001111
1100000111101000000000100110000011110000000000010010001001010001
0111101000100010110100010111101000111010000000000000000000000000
0000000011101000000000000000000000000000000000001100011101000101
1111110000000000100000000000000000000000110001110100010111111000
0000001000000000000000000000000011000111010001011111010000000011
0000000000000000000000001100011101000101111110000000001000000000
0000000000000000100010110100010111111100100010011100001000001111
1010111101010101111110001000101101000101111110000000000110100010
1111101000000111110101111110000101000100101000101111011001011100
0000000000000000000000000000000011001001110000111001000010010000
1001000010010000100100001001000010010000100100001001000010010000
1001000010010000100100001001000000100001000000000000000000000000
0000101100000000000000000000000000010100000000000001001100000000
0000000000000000000100100000000000000000000000000000101000000000
0010111001100110011010010110110001100101000000000000000000000000
0000000000000000000000000000000011111110111111110000000000000000
0110011100000001011100110110100101101101011100000110110001100101
0010111001100011000000000000000000000000000000000000000000000000
0000000000000000000000000000000001011111011011010110000101101001
0110111000000000000000000000000000000000000000000000000000000000
0000000100000000000100000000000000000000100000000000001011100111010
0110010101111000011101000000000000000000000000000000000000000000
0000000000000000000000001000000000000000000000000000001100000001
0110001000000000000000000000000000000000100000000000000000000000
```

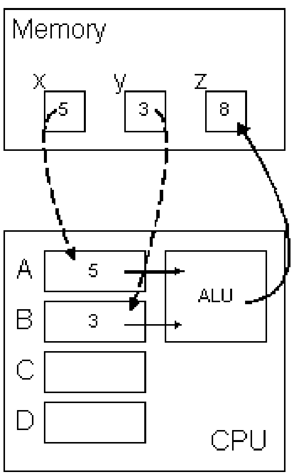
```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000001011100110010001100001011101000110000100000000
0000000000000000000000000000000000000000000000000000001000000000
0000000000000000000000110000000100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000010111001100010
0111001101110011000000000000000000000000000000000000000000000000
0000000000000000000000110000000000000000000000000000001100000001
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000010111110101111101011111011011010110000101101001
0110111000000000000000000000000000000000000000000000000000000000
0010000000000000000000100000000100000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000101111101011111
0110000101101100011011000110111101100011011000010000000000000000
0000000000000000000000000000000000000000000000000000001000000000
0000010000000000000000000000000011111111
```

Now, which language would you rather work with?

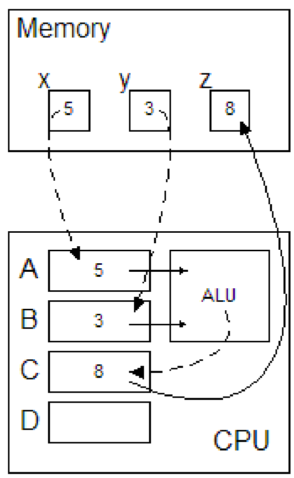
Simple calculation:

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

Simplified view at runtime showing 3 variables in memory, a CPU with 4 registers and an Arithmetic-Logic Unit:




or possibly like this:




## Putting It All Together

**Step 1: Edit**

Create a text file for this C code named **`simple.c`**. The size of this file is 119 bytes. (You can use any text editor like Notepad++ or, preferably, the [Crimson Editor].)

```
int main(void)
{
  int a = 1;
  int b = 2;
  int c = 3;
  int d = 4;
  int e = a * b * (c + d);
  return 0;
}
```

**Step 2: Compile**

The *source* code (text) is compiled into *object* code (binary) and saved in a file named **`simple.o`**. The size of this file is about 450 bytes (depends on the compiler and version).

```
gcc -c simple.c -o simple.o
```

**Step 3: Link**

The object file is linked (combined) with other object code and saved in a file named **`simple.exe`**. The size of this file is about 10,000 bytes. (Again, depends on the compiler.)

```
gcc simple.o -o simple.exe
```

**Step 3: Execute**

Run the executable file by simply typing the name of the executable file (the .exe extension is optional under Windows):

```
simple
```

Of course, nothing *appears* to happen. The program *did* run and it *did* perform the calculations. There just aren't any instructions in the program that display anything for you to see. (Also, the sizes of the object files and executable files will vary with the version of the compiler you are using.)

We can modify it to use the **`printf`** function and display the value of *e* after the calculations:

```c
#include <stdio.h> /* printf */

/* Calculate some values */
int main(void)
{
  int a = 1;
  int b = 2;
  int c = 3;
  int d = 4;

  int e = a * b * (c + d);
  printf("%i\n", e);
  return 0;
}
```

The C program above is a complete program that, when executed, will print the value `14` on the screen.

**Additional Compile Switches**

To enable the compiler to perform a more thorough check of your source code, use the **`-Wall`** *command line switch* like so:

```
gcc -Wall -c simple.c -o simple.o
```

Now, if the compiler detects any potential problems or misuse of the C language, it will alert you with a warning message. For example, if I add the variable *f* like this:

```c
  int a = 1;
  int b = 2;
  int c = 3;
  int d = 4;
  int f = 5; /* Add this variable, but don't use it anywhere */

  int e = a * b * (c + d);
  printf("%i\n", e);
  return 0;
```

I get this warning from the compiler:

```
simple.c: In function 'main':
simple.c:10: warning: unused variable 'f'
```

or, if I add something like this:

```c
a + b - c;   /* Perform some calculation, but discard the value */
```

I get this warning from the compiler:

```
simple.c: In function 'main':
simple.c:11: warning: statement with no effect
```

#### Compile and Link in One Step

If you want to perform both compile and link steps with one command, don't provide the **`-c`** switch. This will compile and then link the program:

```
gcc simple.c -o simple.exe
```

However, if the compile step fails for any reason, the link step is skipped.

#### Other Useful Switches

If you want to generate the assembly output, use **`-S`** switch:

```
gcc -S simple.c
```

This will produce a text file called `simple.s` which you can view with any text editor.

If you want to generate the preprocessor output, use **`-E`** switch:

```
gcc -E simple.c
```

This will produce a ton of information to the screen. To capture the output so you can view it more easily, redirect the output to a file:

```
gcc -E simple.c > simple.out
```

The

```
> simple.out
```

causes the output to be written to a file (named `simple.out`) instead of to the screen. This file is a text file that you can view with any text editor.

## Another Example

This example will show constructs such as identifiers, literal constants, defines, expressions, and several others.

**The C code: `marathon.c`**: (with line numbers for clarity)

```c
#include <stdio.h> /* printf */

#define YARDS_PER_MILE 1760
#define KILOS_PER_MILE 1.609

/* A marathon is 26 miles, 385 yards              */
/* Prints the distance of a marathon in kilometers */
int main(void)
{
  int miles, yards;
  double kilometers;

  miles = 26;
  yards = 385;
  kilometers = (miles + (double)yards / YARDS_PER_MILE) * KILOS_PER_MILE;
  printf("A marathon is %f kilometers\n", kilometers);

  return 0;
}

```

The program will output: `A marathon is 42.185969 kilometers`

The preprocessed file named `marathon.pre` (generated by: **`gcc -E marathon.c > marathon.s`**)

The assembly file named `marathon.s` (generated by: **`gcc -S marathon.c`**)

The code above uses a few arithmetic operators. Many operators correspond with the ones you've seen in algebra. Here are a few *binary* operators:

| Operator | Meaning |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| % | Modulo (Remainder) |

A full list of operators including their *precedence* and *associativity*.

## Computer Data Storage (Refresher)

This section is just a short refresher on the binary, decimal, and hexadecimal number systems.

- Computers represent information as patterns of bits (0's and 1's), known as base 2; e.g. 1011101000010101000111100011
- People use base 10; doesn't translate well into base 2
- Hexadecimal (base 16) works well with binary; computer programmers often work in base 16
- Group the bits into sets of 4 and translate into hexadecimal using the chart below

Relationship between hex and binary numbers:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | 0000 | **4** | 0100 | **8** | 1000 | **C** | 1100 |
| **1** | 0001 | **5** | 0101 | **9** | 1001 | **D** | 1101 |
| **2** | 0010 | **6** | 0110 | **A** | 1010 | **E** | 1110 |
| **3** | 0011 | **7** | 0111 | **B** | 1011 | **F** | 1111 |

Character representations

The number above, 1011101000010101000111100011, translates into hex (BA151E3) and decimal (195,121,635) as:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Binary** | 1011 | 1010 | 0001 | 0101 | 0001 | 1110 | 0011 |
| **Hexadecimal** | B | A | 1 | 5 | 1 | E | 3 |
| **Decimal** | 195121635 | | | | | | |

More information on Binary numbers.

Binary/Decimal converter (BinConverter.exe)

## Lexical Conventions

C programs are typically stored in one or more files on the disk. These files are given a rather fancy name: *translation units*. After the pre-processor has removed the directives and performed the appropriate action, the compiler starts its job. The first thing the compiler needs to do is to *parse* through all of the *tokens* in the files.

There are different classes of tokens (lexical elements). In no particular order they are:

1. keywords
2. identifiers
3. constants
    - integers
    - characters
    - floating point
    - enumerations
    - string literals
4. operators (Lots!)
5. puncuators and separators

The standard actually names these 6:

1. keywords
2. identifiers
3. constants
4. string literals
5. operators
6. puncuators

*White space* includes things like blank spaces, tab, newlines, etc. Comments are a form of whitespace since they are stripped out and replaced by a single space.

We will spend the entire course studying these aspects of the C language.

## Identifiers

- Variables, constants, function names, and other elements of a C program are called *identifiers*; they are used to name things.
- Identifiers must contain only letters (upper- or lower-case), digits (0-9), and underscores ( _ ).
- They must begin with a letter or underscore. (Not a digit)
- There are certain words in C that have special meanings (keywords) and can't be used as identifiers. (e.g. **int, float, const**)
- C is a case sensitive language; upper- and lower-case letters are considered to be different. (e.g. `SUM, Sum, sum` are all different identifiers)
- Most compilers allow identifiers to be 31 characters or more in length. (You probably won't need more than that.)
- Use meaningful names, you will be glad you did (and your grade in this course will reflect it!)

Some examples:

| Valid | Invalid | Invalid Reason |
|---|---|---|
| `foo1` | `1foo` | Doesn't start with a letter or underscore |
| `_foo` | `$foo` | 1. Doesn't start with a letter or underscore<br>2. $ is illegal character |
| `foo` | `foo$` | $ is an illegal character |
| `vaid_identifiier` | `invalid-identifier` | - is an illegal character |
| `a_long_and_valid_name` | `foo bar` | Can't have spaces in identifier names |
| `Int` | `int` | **int** is a keyword |

Examples:

| Good Identifier Names | Bad Identifiier Names |
|---|---|
| `int rate;`<br>`int time;`<br>`int distance;`<br><br>`rate = 60;` | `int x;`<br>`int y;`<br>`int z;`<br><br>`x = 60;` |

| | |
|---|---|
| `time = 20;`<br>`distance = rate * time;` | `y = 20;`<br>`z = x * y;` |
| `#define PI 3.1416F`<br>`float radius = 5.25F;`<br>`float volume;`<br>`volume = 4.0 / 3.0 * PI * radius * radius * radius;` | `#define A 3.1416F`<br>`float id1 = 5.25F;`<br>`float id2;`<br>`id2 = 4.0 / 3.0 * A * id1 * id1 *id1;` |
| `double base = 2.75, height = 4.8;`<br>`double area_of_triangle = 0.5 * base * height;` | `double table = 2.75, chair = 4.8;`<br>`double couch = 0.5 * table * chair;` |

## Keywords

Keywords are identifiers that are reserved for the compiler. You can't use any of these as identifiers:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| auto | const | double | float | int | short | struct | unsigned |
| break | continue | else | for | long | signed | switch | void |
| case | default | enum | goto | register | sizeof | typedef | volatile |
| char | do | extern | if | return | static | union | while |

Remember that C is *case-sensitive* so these keywords must be typed exactly as shown. **`int`** is not the same as **`Int`** or **`INT`**.

## Constants

A literal value is a constant just as you type it in the code:

```
int a = 1;
float f = 3.14F;
double d = 23.245;
```

Examples:

| Constant | Type |
|---|---|
| 5 | int |
| 3.14 | double |
| 3.14F | float |
| 'A' | int |
| "hello" | string |

The `marathon.c` program has 6 literal values. There are 4 integers, 1 double, and 1 string.

## Algorithms

A large part of your programming career will deal with *algorithms*.

- One of the most fundamental concepts in computer programming
- A set a steps that defines how a task is accomplished; a recipe
- A computer program represents an algorithm
- Programs = software
- Algorithms have been known and studied for centuries; they were not invented by computer scientists
- Algorithms *do not* require computers, although computers require algorithms
- There are many famous algorithms; Euclid's Greatest Common Divisor (GCD) is a simple one (e.g. the GCD of 27 and 12 is 3, the GCD of 68 and 12 is 4)

Euclid's algorithm in English (with some algebra thrown in):

| Step | Actions to be Performed |
|---|---|
| 1 | Assign the larger number to M, and the smaller number to N. |
| 2 | Divide M by N (M/N) and assign the remainder to R. |
| 3 | If R is not 0, then assign the value of N to M, assign the value of R to N, and return to step 2.<br>If R = 0, then the GCD is N and the algorithm terminates. |

Notice that in Step 3 there are two different possibilities. This is typically how algorithms work. There usually needs to be some *terminating condition*, otherwise the algorithm (program) runs forever.

Here is how we might write the algorithm in *pseudo-code*:

1. Assign larger value to M
2. Assign smaller value to N
3. Divide M by N and assign remainder to R
4. While remainder, R, is not 0
    1. Assign N to M
    2. Assign R to N
    3. Divide M by N and assign remainder to R
5. End While
6. The algorithm has terminated and the GCD is N

Coding this algorithm in an assembler language might look like the code below. This version is using memory locations that are named `M` and `N` for easier understanding.

```
.section .data
M: .long 45       # put 45 in location named M
N: .long 12       # put 12 in location named N

.section .text
.globl _start

_start:
  movl M, %eax     # put M in %eax
  movl N, %ecx     # put N in %ecx
  movl $0, %edx    # zero out for idivl
  idivl %ecx       # divide M/N, (%edx:%eax/%ecx)
                   # result -> %eax, remainder -> %edx

start_loop:
  cmpl $0, %edx    # is remainder 0?
  je loop_exit     # if 0, we're done

  movl %ecx, %eax  # put N in M
  movl %edx, %ecx  # put R in N
  movl $0, %edx    # zero out for idivl
  idivl %ecx       # divide M/N, (%edx:%eax/%ecx)
                   # result -> %eax, remainder -> %edx
  jmp start_loop   # check again
loop_exit:
```

This second version doesn't use any memory locations (like M and N above). All values are stored directly in the registers on the CPU.

```
.section .data
.section .text
.globl _start

_start:
  movl $45, %eax    # put 45 in eax (M)
  movl $0, %edx     # set to 0 (high word of divisor)
  movl $12, %ecx    # put 12 in ecx (N)
  idivl %ecx        # divide M/N, (%edx:%eax/%ecx)
```

```
                    # result -> %eax, remainder -> %edx

start_loop:
  cmpl $0, %edx     # Is remainder 0?
  je loop_exit      # if 0, we're done

  movl %ecx, %eax   # put N into M
  movl %edx, %ecx   # put R into N
  movl $0, %edx     # set high word
  idivl %ecx        # divide M/N, (%edx:%eax/%ecx)
                    # result -> %eax, remainder -> %edx
  jmp start_loop    # continue algorithm
loop_exit:
```

Euclid's GCD algorithm as high-level computer programs (assuming that M and N already have values and that M > N)

<table>
<tr><th>C</th><th>Pascal</th><th>BASIC</th></tr>
<tr>
<td><pre>r = m % n;
while (r != 0)
{
  m = n;
  n = r;
  r = m % n;
}</pre></td>
<td><pre>r := m Mod n;
While r <> 0 Do
Begin
  m := n;
  n := r;
  r := m Mod n;
End;</pre></td>
<td><pre>r = m MOD n
WHILE r <> 0
  m = n
  m = r
  r = m MOD n
WEND</pre></td>
</tr>
</table>

The algorithm reprinted:

| Step | Actions to be Performed |
|---|---|
| 1 | Assign the larger number to M, and the smaller number to N. |
| 2 | Divide M by N (M/N) and assign the remainder to R. |
| 3 | If R is not 0, then assign the value of N to M, assign the value of R to N, and return to step 2.<br>If R = 0, then the GCD is N and the algorithm terminates. |