

## 2 Computer Organization and Architecture

This is the physical structure of the computer.

A) Central Processing Unit (CPU = Arithmetic-Logic Unit (ALU) + Control Unit (CU))

B) Primary storage — memory

C) Secondary storage — disk, CD, DVD, tape in gigabytes

D) Buses, interconnect, network connections

Storage is measured in bytes, words, megabytes, gigabytes

All data is encoded; one byte equals one character, usually ASCII

“Word” usually means 32 bits = 4 bytes, but sometimes means 8 bytes.

Performance often depends on **memory bandwidth** since processors are much faster than cache memory,

cache is faster than ordinary memory, and ordinary memory is faster than secondary storage.

(You will need to get some familiarity with binary numbers in order to do well in this course.)

## Program Translation

Machines don't understand C; they understand **machine code**

To compute  $c = a + b$  you need to translate the symbols  $a, b, c$  into memory locations, recognize the  $+$  as requiring an ADD instruction, and then execute LOAD (value of  $a$ ), LOAD (value of  $b$ ), ADD, STORE (value of  $c$ )

Programming in machine code would be tedious; practically since the beginning of computing there have been higher level programming languages to make the job easier.

Start with **source code** in C. This has symbols, higher level math functions, input and output routines, etc.

Compile down to **object code**. This isn't quite machine code yet. If you call library functions (like sine, square root, etc.) that aren't basic C language functions, then the object code will still have a reference to the library function.

We won't deal much with object code, but you will need to know a little bit about it.

Eventually you get an **executable module**

## Safe and Prudent Programming Practice

Programming is somewhat tedious and is very error-prone because of the tedium.

The primary thing standing between you and a correct program is **you**.

Part of learning to program properly is simply learning a discipline that lets you avoid making the errors that  
you know you will make.

Sometimes this discipline seems excessive.

I guarantee that if you shortcut the discipline you will make mistakes.

## Safe and Prudent Programming Practice

“Failure is the norm. You’re going to fail. So fail in such a way that you can recover quickly and get it right the second time.” (John Mashey)

A working program that does most of what it’s supposed to do, but not everything, has some value.

A program that fails to execute at all has no value, even if the error is almost totally trivial.

“Make it right before you make it better.”

There are some of us who believe that the only way to learn to program properly is through the school of hard knocks. They (and to some extent I) argue that it is only by making the mistakes that you truly understand why there are standards and practices that are adopted as the norm.

Remember what you have learned about the scientific method. One of the fundamental rules of scientific experiment is to control the number of variables, so that a change in the results can be causally linked to a change in one of the variables in the experiment. Programming is much the same. If you make incremental changes, then when a program stops working correctly you will know what change has caused the error. If you make many changes all at once, then you have to worry that it could be any one of the changes or some combination of the changes that has caused the problem.

# The Software Development Process

1. Specify the problem requirements
2. Analyze the problem
3. Design the algorithm to solve the problem
4. Implement the algorithm, that is, write the program
5. Test and verify the program
6. Maintain and update the program

## A Sample Problem

Calculate and display the volume of a cylinder given its base radius and its height, both given in centimeters.

**Everything** can be boiled down to

**Input:** radius in centimeters, height in centimeters

**Computation:** volume  $V = \pi r^2 h$

**Output:** volume  $V$  in cubic centimeters

Pseudocode version of the program:

```
read value of radius r
read value of height h
obtain a value for pi
compute volume  $V = \pi * r * r * h$ 
output V
```

C version of the implemented program

```
#include <stdio.h>
#define PI 3.5678
int main(void)
{
    double h,r,v;

    scanf("%lf %lf",&r,&h);
    v = PI * r * r * h;
    printf("%lf\n",v);

    return(0);
}
```

Where could we make errors?

Where could we make errors?

```
#include <stdio.h>
#define PI 3.5678
int main(void)
{
    double h,r,v;

    scanf("%lf %lf",&r,&h);
    v = PI * r * r * h;
    printf("%lf\n",v);

    return(0);
}
```

We could have screwed up the value of  $\pi$ .

We could have mis-read the value for  $r$  or for  $h$ .

We could have screwed up the implementation of the formula.

We could have mis-written the value for  $v$ .

A better program.

```
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    double h,r,v;

    scanf("%lf %lf",&r,&h);
    printf("radius is %lf and height is %lf\n",r,h);
    printf("we use the value %lf for pi\n",PI);
    v = PI * r * r * h;
    printf("volume is %lf\n",v);

    return(0);
}
```

**Always** “echo” the input values to make sure you have them right.

**Always** check the defined constants to make sure you have them right.

**Always** label the output so you know what the numbers are supposed to mean.

## Buell's Laws of Debugging

- There is nothing in any program that can be taken for granted.
- Debugging a program is an empirical process, not a metaphysical process. Do not try to figure out what the program is *supposed* to be doing. Find out what it is *actually* doing. Do not use the entrails of a chicken or tarot cards. Find out what actual values of variables are changing and you will be well on the way to finding out why the values you are getting are the wrong ones.
- Much of the time, if you simply explain in detail to someone else what your program is supposed to be doing, you will find the error yourself, because you will be forcing yourself to really look carefully at the code.
- It all comes down to discipline.

## A simple program

```
// Convert distances from miles to kilometers
#include <stdio.h> // include the standard input/output libraries
#include <stdlib.h> // include the other standard libraries
#include <math.h> // include the math function libraries
#define KMS_PER_MILE 1.609
int main(int argc, char *argv[])
{
    double kms, miles; // distances in kms and in miles, respectively

    // get the distance in miles
    printf("enter the distance in miles\n");
    scanf("%lf", &miles);

    // convert to kilometers and print
    kms = KMS_PER_MILE * miles;
    printf("%lf miles is %lf kilometers\n", miles, kms);

    return(0);
}
```

Much of this is boilerplate. Just copy this from one program to the next as needed and ignore for the moment what it means.

## Some of the Rules

The `#include` line tells the compiler to include source code from elsewhere. The “dot h” is the convention for a “header” file.

The `#define` is used to define a constant that cannot be changed by the program. The convention is that defined constants are all capitals, and nothing else is done as all caps.

The variables in this program are `kms` and `miles`. There are rules for what constitutes legal variable names.

C has many **reserved words** like `main`, `printf`, `scanf`, and `double`.

Comments can be done in either of two ways.

All C programs have a filename `something.c` that ends with “dot c”.

This program would be compiled with the command `gcc something.c -lm` that would produce an executable file named `a.out`. (The `-lm` says that you want to include the math libraries.)

By simply typing `a.out` and hitting “return” you cause the file to be executed.

“Equals” isn’t really “equals.” Rather, it’s an assignment of the right hand side to the left hand side, and really ought to be written `kms <-- KMS_PER_MILE * miles;` but it isn’t in the C language.

All executable C statements end in a semicolon.

Input and output are done with `scanf` and `printf`, respectively.

The open and close braces `{ }` have the same logical function with regard to source code that parentheses do in mathematical expressions. In this case, they identify everything between the open and close brace as the source code of the `main` program.

All these rules form what is called the **syntax** of the language.

Syntax is tedious, detailed, and annoying.

I will not insist that you memorize all the syntax.

I will insist that you know enough syntax to be able to write correct programs. You should know how to get from point A to point B, but you need not memorize all possible ways of doing so, provided that your decision is a matter of style. You can always look up the syntax in the book and test it on the computer.

## Some Syntax Details—Data Types

**char** – 8 bits, one byte, holding a single ASCII character

**short** – 16 bits, two bytes, holding an integer in the range  $-32768 = 2^{15}$  to  $+32767 = 2^{15} - 1$

**int** – USUALLY 32 bits, four bytes, for integers in the range  $-2147483648 = 2^{31}$  to  $+2147483647 = 2^{31} - 1$

**long** – USUALLY 32 bits, four bytes, for integers in the range  $-2147483648 = 2^{31}$  to  $+2147483647 = 2^{31} - 1$

**long long** – USUALLY 64 bits, eight bytes, for integers in the range  $-2^{63}$  to  $+2^{63} - 1$

**float** – four bytes, for numbers with decimal points, in a variation on scientific notation

**double** – eight bytes, for numbers with decimal points, in a variation on scientific notation

I will (almost?) never use **float** variables. The **double** data type provides more decimals of precision, and there is little reason any more to use the lower precision variables.

## Some Syntax Details—Data Types continued

I will similarly almost always use `long` variables.

The variations in `int`, `long`, and `long long` are machine- and compiler-dependent. Early microprocessor-based computers had 16-bit hardware, and on those machines an `int` was 16 bits. Almost all current microprocessors are now 32-bit machines, so the “default” size for these variables is 32 bits. In the other direction, there are some high-end computers on which all three variables are 64 bits, because the hardware is a native 64-bit processor.

You can always find out what the data type means. If you declare a variable with

```
long n
```

then in a program the C function

```
sizeof(n)
```

will return the number of bytes used to store that variable.

## Some Syntax Details—Data Types continued

Declare variables as

```
long n;
```

```
double x,y;
```

and then assign them values with an executable statement such as

```
n = 13567;
```

```
x = 19.3579;
```

```
y = 19.3579e5;
```

Variables that have been declared but for which no value has yet been assigned are officially UNDEFINED although some compilers will initialize the value to zero.

DO NOT RELY ON THIS INITIALIZATION. Declaration of a variable really means that the program is pointing to some location in memory where that variable will be stored. If you have not yourself explicitly stored something there, you may well find that the “value” of the variable is whatever the bit pattern is that is left over from the last time that memory location got used.

## Some Syntax Details—Variable Names

C variable names can be of any length (although some compilers will have limits) and may contain any alphanumeric characters and the underscore and must start with either an alphabetic character or the underscore.

C is CASE SENSITIVE. The variable

`this_variable`

is not the same as the variable

`This_Variable`

A “convention” exists that defined constants are all caps but that variables are not all caps. It is not a good idea to make a variable be all capital letters.

Style rule number one: **Variable names should be meaningful.** Don’t make the names so long they clutter up the readability of the program, but don’t use one-letter names unless one-letter names are meaningful.

## Safe Practices With Constants

It is legal to put a constant in an assignment:

```
x = 32.2 * y;
```

However, when you come back to that line of code, how will you know what the 32.2 really means?

And even if you remember, how easy will it be to change from English to metric?

```
#define GRAVITY_ENGLISH 32.2
#define GRAVITY_METRIC 9.81
...
x = GRAVITY_ENGLISH * y;
```

Remember the convention that defined constants are written all caps.

## An Exception to Every Rule

```
double PI;
```

```
PI = 4.0 * atan(1.0);
```

In most cases, there is no way to get an exact value of a constant into a program.

In this case, however, since  $\arctan(1.0) = \pi/4.0$ , and since 1.0 and 4.0 are exact going decimal to binary, this code gets into the program the best possible value for  $\pi$ , because we assume the writers of the compiler and the math library did it better than we could do.

## Safe Practices With Constants

It is legal to declare a variable and define a value at the same time:

```
double one = 1.0;
```

However, if you really mean to use a `CONSTANT` of 1.0, then declare a defined constant. If you declare a variable and assign a value as above, then you need to be very careful about assigning a new value to this variable. It is common to see things such as

```
execute chunk of code  
one = 2.0;  
execute chunk of code
```

and then the second time the chunk of code get executed, the variable named `one` has the value 2.0.

This will probably be a bad thing.

## Some Syntax Details—Assignment Statements

In programming, EQUALS IS NOT "EQUALS"

Assignment statements look like

```
r1 = (-b + sqrt(b*b - 4.0*a*c))/(2.0*a);
```

when in fact the meaning is more like

```
r1 <-- (-b + sqrt(b*b - 4.0*a*c))/(2.0*a);
```

or "r1 gets the value yada yada"

(Some programming languages, but not C, use a left-arrow to indicate assignment.)

## Some Syntax Details—Assignment Statements

Order: (mult and div), (add and sub), left to right

MUST have the `*` for multiplication

Parentheses mean the same as in algebra, and are good practice

Constants are `4.0` and `2.0`, not just `4` and `2`

There is no exponentiation inherent in the C language

The `sqrt`, `exp`, `log`, `sin`, `cos`, `tan`, `atan`, etc., functions all come from the `math.h` library and are included by using the `-lm` at the end of the `gcc` line in the `makefile`

## Some Syntax Details—Assignment Statements

Safe programming: don't be cute or clever; use spaces and parentheses to make the expression as readable as possible. If necessary, break it up over multiple lines:

```
disc = b*b - 4.0*a*b;  
r1 = (-b + sqrt(disc))/(2.0*a);}
```

or even

```
num = -b + sqrt(b*b - 4.0*a*b);  
den = 2.0*a;  
r1 = num/den;
```

Lots of errors come from misplaced parentheses or no parentheses

## Syntax Details

All programs look like

```
include files
global definitions, declarations, and header stuff
main()
{
    local declarations
    executable statements in the main program
}
function1()
{
    local declarations
    executable statements in this function
}
function2()
{
    local declarations
    executable statements in this function
}
etc.
```

The main program and functions can be arranged in any order, with some syntactical fuss

## Syntax of Symbol Names

Must start with an alphabetic character or an underscore

Don't use a leading underscore (except *exactly* where you need to)

Must be alphanumeric with underscore

## Syntax of Arithmetic Expressions

Detailed rules exist for arithmetic expressions

Ignore the rules—parenthesize for clarity instead

## Statements

C ignores white space, including newlines

So use white space for clarity

## Logical Expressions

C evaluates all logical expressions to produce a logical “yes” (1) or logical “no” (0)

Do not rely on the rules—parenthesize for clarity

Do not think about

```
x >= y && z == w
```

Instead, write what you really would mean

```
(x >= y) && (z == w)
```

```
== // equal to
```

```
!= // not equal to
```

```
<= // less than or equal to
```

```
>= // greater than or equal to
```

```
|| // logical or
```

```
&& // logical and
```

## A Way to Prevent A Very Common Dumb Mistake

If you code something like

```
if(x == 1.0)
{
    code to be executed
}
```

then you *will* (I guarantee it!) goof and write

```
if(x = 1.0)
{
    code to be executed
}
```

which is very much different. The second version *assigns* 1.0 to the variable **x** and returns a “true” because the assignment succeeds. Avoid this error by writing, if one half of an equals is a constant,

```
if(1.0 == x)
{
    code to be executed
}
```

In this case, if you write = instead of ==, the compiler will flag this as an error.

## If-then-else Statements

```
disc = b*b - 4.0*a*c;
if(0.0 == disc)
{
    code to be executed if the test succeeds
}
else if(disc > 0.0)
{
    code to be executed for positive values of {\tt disc}
}
else
{
    code to be executed for negative values of {\tt disc}
}
```

You can string together as many such **else if** parts as you want

However, the above code has stylistic flaws.

NEVER test for exactly zero. Floating point values are NEVER exactly zero.

And always test for *exactly* the options you want.

## If-then-else Statements

```
#define EPSILON 0.000001
disc = b*b - 4.0*a*c;
if(abs(disc) < EPSILON)
{
    code to be executed for an effectively zero disc
}
else if(disc > 0.0)
{
    code to be executed for positive values of disc
}
else if(disc < 0.0)
{
    code to be executed for negative values of disc
}
else
{
    printf("ERROR: disc not positive, not negative, not zero\n");
    exit(0);
}
```