Chapter III

Getting Started With Programming

3.1 Programming and Programming Languages

There are basically three types of files that are found on computers: data file, libraries and programs. Data files contain information but do nothing with it. Libraries are files are used as a part of one or more programs. By themselves they do nothing but may be used to provide functionality to a program. Programs, also called executable files, contain instructions that the computer can natively understand. Fundamentally all a program does is read and write bits in the computer's memory.

There are three main approaches to writing a computer program.

Machine code This means writing the program in the computer's native language.

Assembly code A slightly more human readable version of machine code. The specific instructions that the computer uses to perform operations are called. These are then translated to machine code.

High level language Here the instruction set of the underlying computer hardware is abstracted away. The code is generally more human readable and less error prone. This is generally translated into assembly and then machine code.

We will use the high level languages C and C++ for this class. There are many other languages we could use, such as FORTRAN and Java. Some of the advantages and disadvantages of C are:

Pros

- **C is a small language.** C has few built-in features compared with most languages and this means fewer features to learn. This is generally extended with libraries of functions, such as the C Standard Library.
- **Efficient.** C was built to be fast.
- **Portable.** A properly coded C program can be compiled and run on almost any modern computer.
- **Flexible.** C is not restricted to scientific computing. You can use C to write any type of program you can think of.
- **Standard Library.** C has a large standard library where much of the basic programming work is already done for you.

Cons

- **Error-prone.** It is easy to make mistakes in C programs and can be difficult to find them when they arise.
- **Difficult to read.** It is very easy to make C programs that are difficult to read. The writer must always work to create readable code.
- **Difficult to modify.** If a program is not specifically written with maintenance in mind it may be as much work to make a small change as to write a new program from scratch.

3.2 History of C and Computing Languages

The history of C really starts with the history of computing languages. Before graphical user interfaces (GUIs) and even keyboards programmers had to manually program computers using dip-switches and punch-cards.

The first major advance was assembly language. Assembly did, and still does, allow the programmer to set individual registers of the central processing unit (CPU). This allowed a completely electronic and far more general method of computing than ever before. However, this method was still time consuming, cumbersome and error prone. It also had the disadvantage of not being portable. That is, a program written for one type of computer could not be run on a different computer without major changes.

The first successful high level language was FORTRAN (FORmula TRANslator) and was completed in 1957. It was a major breakthrough in computing, promising to be as fast as hand written machine code while reducing the necessary manpower from one scientist and ten engineers to one scientist and, maybe, one engineer. FORTRAN 77 and 90/95 are still the most popular scientific computing languages for high performance computing.

The birth of C was almost solely due to the birth of UNIX. C has its roots in several different languages that were developed during the 1960s, after the original FORTRAN. Most of the languages that influenced C were used in the early development of UNIX at Bell Laboratories by Ken Thompson and Dennis Ritchie and a few others. Originally UNIX was written in assembly but this had the problems of being difficult to code, debug and enhance. Thus, a new language was developed, B. B was based on a language called BPCL which was based on another language, ALGOL. However, when the team got a new computer it was clear that B was not good enough. The language was extended and called NB (for 'New B') and finally C. Once C was refined (and the entire UNIX platform rewritten in it) a new, important benefit was realized: portability. By creating C compilers for different computers UNIX could be easily ported to these computers as well.

C continued to change and evolve through the 1970s and outgrew its UNIX roots. It was no longer standard to use C on the UNIX platform only. Begin in 1983 the American National Standards Institute (ANSI) created a standard for C that was completed in 1988. This was further approved by the International Standards Organization (ISO) in 1990. The version of C that we will be using is this one and is commonly referred to as 'ANSI C' or 'ANSI/ISO C'.

3.3 Structure of a C Program

It is possible to write a simpler C program than our first example, helloworld.c, but it would be far to boring to be of any use.

```
/**********************************************************
 *A program that prints "Hello, world!"                      comments
 **********************************************************/

#include <stdio.h>                                           directive

int main(){                                                  main function
7    printf("Hello, world!\n");                    output
8    return 0;                                     return succesful
9  }                                               end of main and program
```

This demonstrates the basic structure of a typical C program. Lines 0-2 are comments. These can actually appear anywhere in our code and are discussed more in Section 3.3.2. Line 4 is a pre-processor directive that, in this case, tells the compiler that we want to use the `stdio` library or standard input/output library. We discuss directives and how the compiler handles them in Sections 3.3.4 and 3.3.1. On line 6 we declare the `main` function. This is where our program really starts. When we run our program `main` is called by the computer and everything between the `{}`s is executed. The entire output of the program is on line 7 and will be discussed in Section 3.3.3. Line 8 tells the computer that our program finished successfully. Finally, line 9 signals the end of our `main` function that started on line 6.

Note: C is case sensitive. If you type in the above program make sure to type the characters exactly as you see them.

3.3.1 Compiling Our Code

To run the example above we need to compile it first. The compiler is just another program on the computer like any other. The source code (the text in the example) is typed into a plain text file. This source file is really just a data file that is used by our compiler. The compiler takes this source file and translates it into machine code. This machine code is then an executable file that can run on our computer. We will see more of the details of how to do this in the lab.

It is useful to know a little bit about the way that the compiler works. It is three step process. The first step is to run the C pre-processor (CPP). This is really just a search-and-replace program that looks for any line that starts with a `#`. The CPP then reads the line and figures out what it needs to place inside our source file (note that our original file is not actually changed on disk). In the case of line 4 of our example it is inserting information about our `stdio` library. The next step changes our modified source code into machine code called an object file. The final steps is to link all our object files together into a single executable. Since our example has only one file the last two steps are combined and only an executable is produced.

3.3.2 Comments

Arguably the most important part of our source code is the comments. These are ignored by the computer but invaluable to the humans who have to read and write the source code. C and most other programming languages are not easy to read. They do complicated things and even the people who write them forget what those things are. Thus, it is important to explain in plain english what is happening at each point of the program. Any program you write should always start with a description of what the program does at the top along with your name. Commenting and coding style in general will be covered later in the course.

Comments in C always start with `/*` and end with `*/`. Anything inbetween them is ignored by the compiler. Comments can go anywhere in the text of the source code as long as they do not fall in the middle of a word or between quotation marks.

hellocomments.c is our original example with comments scattered throughout. Note that the compiled program is exactly the same.

```
0  /**********************************************************
1   *A program that prints "Hello, world!"
2   **********************************************************/
3
4  #include /* 1 */ <stdio.h>
5
6  int /* 2 */ main(/* 3 */) /* 4 */{
7    pr/*comment*/intf("Hello, world!\n"); /* 6 */
8    /* 8 */ return 0/*
```

```
        10
                    */;
  /* /* 7  */ */
}
/* 9 */
```

3.3.3 Basic Output

The `printf` command allows us to send text and information to the screen. In `helloworld.c` we see the command used to print text. The command requires a string (a list of characters surrounded by quotes). `printf` only prints what we tell it to so print a new line we need to use the `\n` character. printf.c prints several strings to the screen.

```
/*****************************************
 *An example of using printf
 *****************************************/
                                          Necessary to use printf
#include <stdio.h>

int main(){
  printf("This is a line of text\n");     There is no new line at the end of this string
  printf("This is also ");
  printf("a line of text\n");
  printf("This is two\nlines of text\n"); There is a new line in the middle of this string
  printf("\n\n\n");                       Three new lines
  return 0;
}
```

3.3.4 Directives

Directives are used by the CPP to do serach-and-replace on our source code. Two important examples are `#include` and `#define`.

`#include` Directive

`#include` is used to include the contents of other files in our own file. For example,

`#include <stdio.h>`

will find the file `stdio.h` in a special directory where standard library files are kept and paste the contents into our file. Similarly,

`#include ‘‘path/file’’`

will find paste the contents of the file `file` with path `path` into our source file.

`#define` Directive

`#define` will excute search and replace using the two items that follow it. For examples,

`#define PI 3.14159`

will search for every instance of `PI` and replace it with `3.14159`.

3.3.5 Variables

Variables allow us to store information. As we have already seen with integers and floating point numbers different types of information must be stored in different ways. There are three general variable types available in C: integers, floating points and characters.

Integers

Integers may be unsigned but are signed by default. There are three different sizes of integers that can represent different ranges of numbers. The number of bits used by each interger type is dependent on the type of cpu that is being used. For a 32-bit machine, like the ones we are using, the different types are as follows

Type	Number of Bits	Smallest Value	Largest Value
`short int`	16	-32,768	32,767
`unsigned short int`	16	0	65,535
`int`	32	-2,147,483,648	2,147,483,647
`unsigned int`	32	0	4,294,967,295
`long int`	32	-2,147,483,648	2,147,483,647
`unsigned long int`	32	0	4,294,967,295

Floating Points

There are three sizes of floating point numbers: `float`, `double` and `long double`. `float` and `double` are also known as single and double precision numbers as `double` has twice as many bits as `float`.

Type	Number of Bits	Smallest Positive Value	Largest Value	Precision
`float`	32	1.17×10^{-38}	3.40×10^{38}	6 digits
`double`	64	2.22×10^{-308}	1.79×10^{308}	6 digits

The size of `long double` is not strictly defined but is typically 80 to 128 bits on most 32-bit machines.

Characters

There is only one type of character variable: `char`. C actually treats this as an 8-bit integer. The numerical values are then mapped to character values that may or may not be printable.

The most common mapping, and the one used on the computers we will be using, is call ASCII (American Standard Code for Information Interchange). Other than the characters that are on the keyboard there are a few more important ones that are encoded.

Name	Escape Sequence
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\t`
Backslash	`\\`
Question mark	`\?`
Single quota	`\'`
Double quota	`\"`

Declaration

To use a variable we must declare it first. To declare a variable we must first state the type of variable we want followed by the name. For example, to define a `int` variable named `myinteger`.

```
int myinteger;
```

Note that this is a statement so it must end in a `;`. More than one variable of the same type may be declared in the same statement by using a comma to separate the names.

```
int myinteger, myotherinteger;
```

Variable names may be composed of letters, numbers and `_`. The name must start with a letter. It is a general convention that variable names are always lower case.

```
int myinteger, myotherinteger;
float float1, float_2;
```

Variable declaration must be done in `main` before anything else. This means all varaibles must be declared as a block and no other commands or statements can interupt.

Assignment

To use a variable we must be able to assign values to it. This is done with the assignment operator `=`.

```
myinteger = 42;
float1 = 6.011e-32;
```

We can also assign the variable when we declare it.

```
int myinteger = 42, myotherinteger;
float float1 = 6.011e-32, float_2 = 16, product;
```

Once a variable has a value assigned we can use it.

```
product = float1 * float_2;
```

Note that the operators `*`, `/`, `+` and `-` correspond to $\times$, $\div$, $+$ and $-$. Standard order of operations rules apply.

Printing Variables

If we couldn't get the value of a variable out of our program then our program is useless. The command to print anything to the screen in C is `printf`. We have already seen `printf` in `helloworld.c`. To print variables we must insert conversion specifiers into the text. For integers we will use `%d` for 'decimal' and `%g` for floating points.

```
printf("product = %g * %g = %g\n", float1, float_2, product);
```

Example

A simple example of using variables can be found in volume.c.

```
/*******************************************************
 * A program to calculate volume and demonstrate variables.
 *******************************************************/

#include <stdio.h>
```

```

int main(){
  /* declare and assign our variables */
  double volume, height=10, width, length=6e-1;

  width=1.5;


  /* caluculate the volume */
  volume = height * width * length;

  /* print the values of the variables */
  printf("The dimensions of the box are %g X %g X %g\n", height, width, length);
  printf("The volume is %g\n", volume);

  return 0;
}
```

all variables must be declared here

3.3.6 Keywords

There are a number of keywords are a part of the C language and have special meaning. As a result you can not use these for variable or constant names.

`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`	`static`	`while`

3.4 Mathematical Expressions

As a general rule if a program is going to be useful it has to do math at some point. Doing math in C involves writing out expressions that are similar to, though not quite the same as, what we would typically write on paper. Mathematical expressions consist of variables connected by operators. We will explain these operators and how they interact with different types. Finally, we will explore C's built-in math library.

3.4.1 Operators

There are many different types of operators in C. For example, mathematical, logical and bit-wise operators. Logical operators are covered in Section 3.5 and bit-wise operators are beyond the scope of this course.

All operators are either binary or unary and may be non-commutative. The terms 'unary' and 'binary' refer to whether or not the operator operates on one or two variables at a time. Commutative means that the order of the variables, or which side of the operator the variable(s) are on, doesn't matter?

Arithmetic Operators

The arithmetic operators are

Operator	Binary/Unary	Function	Commutative
`+`	unary	positive	Left side of variable.
`-`	unary	negative	Left side of variable.
`+`	binary	addition	Yes.
`-`	binary	subtraction	No.
`*`	binary	multiplication	Yes.
`/`	binary	division	No.
`%`	binary	remainder	No.

Non-commutative operators give different results if the variables involved swap sides about the operator. For example, + i commutative,

$$7 + 2 = 2 + 7,$$

but − is not,

$$7 - 2 \neq 2 - 7.$$

When implementing a complex equation we may ask "in what order will our operators be applied?" C uses the same order of operations that applies to ordinary mathematics. This is sometimes called operator precedence. The order of precedence is

highest	`- +`	(unary)
	`/ % *`	
lowest	`- +`	(binary)

If in doubt we can always use parentheses to specify the order. Note that we can only use `()` as other brackets have special meaning. For example,

`a + b * -c`	equals	`a + (b * (-c))`
`a / b - +c`	equals	`(a / b) - (+c)`
`a % b * c`	equals	`(a % b) * c`

This is demonstrated in the file arithmetic.c

```
/*********************************************
 *demonstration of simple arithmetic
 *********************************************/

#include <stdio.h>

int main(){
  int a=-5, b=7, c=22, d=2;
  double x=1.23456789, y=55.6, z=-13.4;

  printf("%d +  %d *  -%d   = %d\n", a, b, c, a + b * -c);
  printf("%d + (%d * (-%d)) = %d\n", a, b, c, a + (b * (-c)));

  printf(" %f / %f  -  +%f  = %f\n", x, y, z, (x / y) - (+z));
  printf("(%f / %f) - (+%f) = %f\n", x, y, z, (x / y) - (+z));

  printf(" %d %% %d  * %d = %d\n", b, c, d, b % c * d);
  printf("(%d %% %d) * %d = %d\n", b, c, d, (b % c) * d);

  return 0;
}
```

Increment and Decrement Operators

C offers shorthand notation for incrementing and decrementing a variable by 1. This is done with the operators `++` and `--`. These are unary operators but they have slightly different meaning depending on what side of the variable they appear on. When `++` is used as a single statement the following are equivalent

```
i = i + 1;
i++;
++i;
```

Similarly for `--` the following statements are equivalent

```
i = i - 1;
i--;
--i;
```

The difference appears when the operator is used as part of a larger statement. `++i` increments `i` *immediately* while `i++` increments `i` at some time in the future.' `--i` and `i--` behaves similarly. This is illustrated in the program incrdecr.c

```
/**************************************************
 *demonstration of increment and decrement operators
 *************************************************/

#include <stdio.h>

int main(){
  int i = 0, j = 1, k = 2;

  1 = i;

  /* prints "i = 0" */
  printf("i = %d\n",i);

  /* prints "i = 1" */
  printf("i = %d\n",++i);
  /* prints "i = 1" */
  printf("i = %d\n",i);

  /* prints "i = 1" */
  printf("i = %d\n",i++);
  /* prints "i = 2" */
  printf("i = %d\n",i);

  /* prints "i = 1" */
  printf("i = %d\n",--i);
  /* prints "i = 1" */
  printf("i = %d\n",i);

  /* prints "i = 1" */
  printf("i = %d\n",i--);
  /* prints "i = 0" */
  printf("i = %d\n",i);

  printf("i = %d, j = %d, k = %d\n", i, j, k);
  printf("i-- + ++j + k-- = %d\n", i-- + ++j + k--);
  printf("i + j + k = %d\n", i + j + k);

  return 0;
}
```

> **Note:** While C specifies that `i++` and `i--` will increment `i` at some point in the future it does not specify exactly when. The only thing we can be sure of is that the action will take place before the next statement. Therefore we must take care when using this version of the operators.

Assignment Operators

Assignment variables allow us to assign new values to our variables. We have already seen `=` (the simple assignment operator) but there are a few more for us to consider.

All assignment operators assign from right to left. For example,

```
i = 1; j = 2;
i = j;
```

gives `i`= 2 and `j`= 2 while

```
i = 1; j = 2;
j = i;
```

gives `i`= 1 and `j`= 1. Furthermore,

```
i = 1;
```

give `i` = 1 while

```
1 = i;
```

makes no sense and will not compile. The left hand side must be a variable.

The remaining assignment operators are called compound assignment operators. These have the general form `v` *op*`=` *expression* and are shorthand for `v = v` *op e*xpression. `v` is a variable while *op* is one of `+`, `-`, `*`, `/` or `%`. *expression* is any mathematical expression. For example,

```
i += 2;
```

is the same as writing

```
i = i + 2;
```

All of the compound assignment operators are used in compound.c

```
/**************************************************
 *demonstration of compound assignment operators
 *************************************************/

#include <stdio.h>

int main(){
  int i = 10, j = 3;

  /* prints "i = 10" */
  printf("i = %d\n",i);

  i += j;
  /* prints "i = 13" */
  printf("i = %d\n",i);

  i -= j;
  /* prints "i = 10" */
  printf("i = %d\n",i);

  i *= j;
  /* prints "i = 30" */
  printf("i = %d\n",i);

  i /= j;
  /* prints "i = 10" */
  printf("i = %d\n",i);

  i %= j;
  /* prints "i = 1" */
  printf("i = %d\n",i);

  return 0;
}
```

3.4.2 Integer vs. Floating Point Mathematics

Arithmetic operators don't operate the same on integer and floating point numbers. There are two operators for which this is important: `%` and `/`.

If we use the `%` operator incorrectly our program simply won't compile. In this case both operators must be integers.

For `/` the effect is more subtle. If both variables are integers then integer division takes place. For example, the value of `2 / 5` is 0 and not 0.2 when we use only integers.

Type Conversion

When we mix integers and floating points the integer variables are automatically promoted to floating point variables. This means that `2. / 5` has a value of 0.2. This is because `2.` is considered a floating point number and `5` is promoted to a floating point number.

We can force integers to be treated as `float`s or `double`s by type casting them. This is done with the general form `(`*type*`)`*variable*. This is especially useful for variables but can also be used for numeric values. For example, `(double)2/5` has a value of 0.2. The same technique can be used to cast floating points to integers.

All of these possibilities are shown in typecast.c.

```
/*********************************************************
 * A program to demonstrate the role of types in arithmetic
 ********************************************************/

#include <stdio.h>

int main(){
  int a = 1, b = 3;
  double x = 1, y = 3;

  printf("\nIntegers\n");
  printf("%d / %d = %d\n", a, b, a/b);

  printf("\nType mixing\n");
  printf("%f / %d = %f\n", x, b, x/b);
  printf("%d / %f = %f\n", a, y, a/y);
  printf("%f / %f = %f\n", x, y, x/y);

  printf("\nType cast to double\n");
  printf("%f / %d = %f\n", (double)a, b, (double)a/b);
  printf("%d / %f = %f\n", a, (double)b, a/(double)b);
  printf("%f / %f = %f\n", (double)a, (double)b, (double)a/(double)b);

  printf("\nType cast to int\n");
  printf("%d / %f = %f\n", (int)x, y, (int)x/y);
  printf("%f / %d = %f\n", x, (int)y, x/(int)y);
  printf("%d / %d = %d\n", (int)x, (int)y, (int)x/(int)y);

  return 0;
}
```

3.4.3 math.h

`#include <math.h>` allows us to use 22 mathematical functions from the C standard library. These are described below.

`double cos(double x)`
Cosine of `x` where `x` is in radians.

`double sin(double x)`
Sine of `x` where `x` is in radians.

`double tan(double x)`
Tangent of `x` where `x` is in radians.

`double acos(double x)`
Arc cosine of `x` where `x` is between -1 and 1.

`double asin(double x)`
Arc sine of `x` where `x` is between -1 and 1.

`double atan(double x)`
Arc tangent of `x`.

`double atan2(double x, double y)`
Arc tangent of `y/x`.

`double cosh(double x)`
Hyperbolic cosine of `x`.

`double sinh(double x)`
Hyperbolic sine of `x`.

`double tanh(double x)`
Hyperbolic tangent of `x`.

`double exp(double x)`
e to the power of `x`.

`double frexp(double value, int *exp)`
Returns f and sets `exp` to n such that `value`$= f \times 2^n$.

`double ldexp(double x, int exp)`
Returns $\mathtt{x}\times 2^{\mathtt{exp}}$

`double log(double x)`
Natural logarithm of `x`.

`double log10(double x)`
Common logarithm of `x`.

`double modf(double value, double *iptr)`
Returns the fractional part of `x`. `iptr` holds the integer part of `value`.

`double pow(double x, double y)`
Returns $\mathtt{x}^{\mathtt{y}}$.

`double sqrt(double x)`
Square root of `x`> 0.

`double ceil(double x)`
Round up `x` to nearest integer.

`double fabs(double x)`
Floating point absolute value of `x`.

`double floor(double x)`
Round down `x` to nearest integer.

`double fmod(double x, double y)`
Floating point modulus of `x` divided by `y`.

3.5 Logical Expressions

As we had a number of mathematical operators we have a number of logical and relational operators. With these we can have the computer execute logical and comparison statements.

3.5.1 Boolean Type

Many high level languages have a special binary type. Not so with C. C simply uses integers with the convention that 0 means false and 1 means true. More generally we will find that 0 means false and everything else means true. This can lead to some interesting expressions but also can be a powerful tool.

3.5.2 Operators

The operators used for logical expressions are

Operator	Binary/Unary	Function	Commutative
`>`	binary	greater than	No.
`<`	binary	less than	No
`>=`	binary	greater than or equal to	No.
`<=`	binary	less than or equal to	No.
`==`	binary	is equal to	Yes.
`!=`	binary	is not equal to	Yes.
`!`	unary	logical negation	Left of variable.
`&&`	binary	logical *and*	Yes.
`||`	binary	logical *or*	No.

When used all of these operators return `1` or `0`.

Relational Operators

Relational operators should be familiar from mathematics. These compare two numbers and return `1` if the statement is true and `0` if the statement is false.

```
/* prints 1 */
printf(``%d\n'', 1 < 2);

/* prints 0 */
printf(``%d\n'', 5.5 >= 16);
```

Note: Be careful combining these in expressions. Take the example

```
x < y < z;
```

This does not test if `y` is between `x` and `z`. Rather, it is evaluated as

```
(x < y) < z;
```

So,

```
x = 7; y = 5;
z = 9;
x < y < z;
```

evaluates to `1` as `x < y` gives `0` and `0` is less than `9`.

Logical Operators

There are three logical operators: `||`, `&&` and `!`. The `||` is an `or` operator, `&&` is an `and` operator and `!` is a logical negation operator.

If one or both of the variables in question are non-zero then `||` returns `1`. This `or` is inclusive which is commonly referred to in everyday language as 'and/or'.

```
int on = 1, off = 0;

/* prints 1 */
printf(``%d\n'',on || off);

/* prints 1 */
printf(``%d\n'',off || on);
```

```
/* prints 1 */
printf(‘‘%d\n’’,on || on);

/* prints 0 */
printf(‘‘%d\n’’,off || off);
```

If both variables are non-zero the `&&` operator returns `1`. Otherwise `0` is returned.

```
int on = 1, off = 0;

/* prints 0 */
printf(‘‘%d\n’’,on && off);

/* prints 0 */
printf(‘‘%d\n’’,off && on);

/* prints 1 */
printf(‘‘%d\n’’,on && on);

/* prints 0 */
printf(‘‘%d\n’’,off && off);
```

The `!` operator changes the value of a integer from `0` to `1` and from non-zero to `0`.

```
int on = 1, off = 0;

/* prints 1 */
printf(‘‘%d\n’’,!off);

/* prints 0 */
printf(‘‘%d\n’’,!on);
```

Note: `&&` and `||` both *short circuit*. That is, when the result of the operator can be determined from the first variable the second one is not looked at. So, if the first variable for `&&` is `0` the result must also be `0`. Similarly, if the first variable for `||` is `1` then the result must be `1`. For example,
`!y || x/y;`
If `y` is equal to `0` then the `||` must be true and `x/y` is not evaluated.

Equality Operators

`==` and `!=` are the only two equality operators.If the values of both variables are equal then `==` returns `1` while `!=` returns `0`. Alternatively, if the two variables are not equal then `==` returns `0` while `!=` returns `1`.

```
int a = 10, b = 1, c = a;

/* prints 1 */
printf(‘‘%d\n’’,a == c);
/* prints 0 */
printf(‘‘%d\n’’,a != c);

/* prints 0 */
printf(‘‘%d\n’’,a == b);
/* prints 1 */
printf(‘‘%d\n’’,a != b);
```

3.5.3 Example

The previous examples are incorporated into logic.c

```
/******************************************
 *Basic use of logic operators
 ******************************************/
#include <stdio.h>

int main(){
  int on = 1, off = 0;
  int a = 10, b = 1, c = a;
  double w=1, x=2, y=5.5, z=16;

  /* prints 1 */
  printf("%f < %f --> %d\n", w, x, w < x);

  /* prints 0 */
  printf("%f >= %f --> %d\n", y, z, y >= z);

  /* prints 1 */
  printf("%d || %d --> %d\n",on, off, on || off);

  /* prints 1 */
  printf("%d || %d --> %d\n",off, on, off || on);

  /* prints 1 */
  printf("%d || %d --> %d\n",on, on, on || on);

  /* prints 0 */
  printf("%d || %d --> %d\n",off, off, off || off);

  /* prints 0 */
  printf("%d && %d --> %d\n",on, off,on && off);

  /* prints 0 */
  printf("%d && %d --> %d\n",off, on, off && on);

  /* prints 1 */
  printf("%d && %d --> %d\n",on, on, on && on);

  /* prints 0 */
  printf("%d && %d --> %d\n",off, off, off && off);

  /* prints 1 */
  printf("!%d --> %d\n",off, !off);

  /* prints 0 */
  printf("!%d --> %d\n",on, !on);

  /* prints 1 */
  printf("%d == %d --> %d\n", a, c, a == c);
```

```
    /* prints 0 */
    printf("%d !& %d --> %d\n", a, c, a != c);

    /* prints 0 */
    printf("%d == %d --> %d\n", a, b, a == b);
    /* prints 1 */
    printf("%d != %d --> %d\n", a, b, a != b);

    return 0;
  }
```

3.6 Branching

Now that we have seen mathematical and logical expressions we can begin to see how we might write programs that do something useful. However, we don't have all the tools that we need yet. One of our missing tools is *branching*. Branching allows our program to do different things depending on the information it is presented with and the conditions in the program.

In this section, as we learn the elements of branching, we will apply what we learn to a simple physical example; calculating the interaction energy of two neutral, spherical particles.

Lennard-Jones Interactions

When modeling and simulating small particles or atoms it is necessary to model their interaction potential energy. A simple and commonly used model is that of Lennard-Jones

$$U = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \qquad \text{(III.1)}$$

where r is the distance between the centers of the two particles. ϵ and σ correspond the well depth and radius of the particle respectively. This potential energy maybe used to model colloids or Nobel gases or it maybe used as part of the interaction potential for more complicated systems, such as protein or metal ions.

This is known as a *pairwise interaction* because it is only applied to pairs of particles. To get the total interaction energy for a system we must add up the energy from all the pairs. This takes a long time if our system contains a large number of particles. In fact, the amount of time needed is proportional to the square of the number of particles, i.e. $\mathcal{O}(N^2)$. Therefore, shortcuts have been produced.

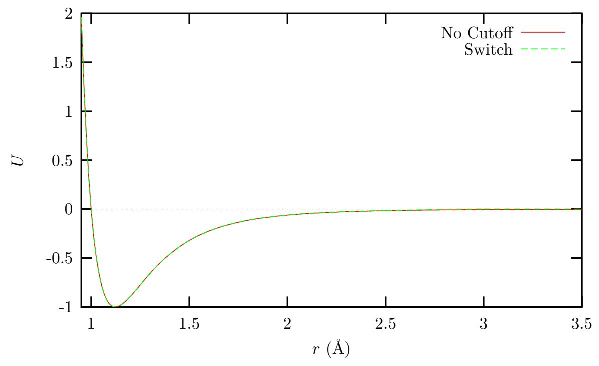
One of the better shortcuts is the switch cutoff. It has the form

$$U = \begin{cases} 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] & r < r_{\text{on}} \\ 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]\left[\frac{(r_{\text{off}}^2-r^2)^2(r_{\text{off}}^2+2r^2-3r_{\text{on}}^2)}{(r_{\text{off}}^2-r_{\text{on}}^2)^3}\right] & r_{\text{on}} < r < r_{\text{off}} \\ 0 & r > r_{\text{off}}. \end{cases} \qquad \text{(III.2)}$$

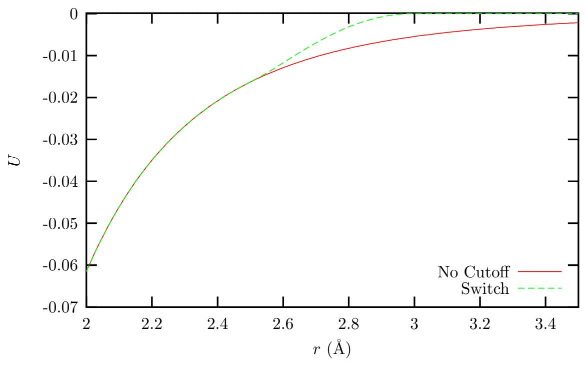
For distances less than r_{on} we use the original potential energy. When r is between r_{on} and r_{off} the original potential energy is multiplied by a switching function. This function continuously drops the potential energy and force to 0 at r_{off}. Beyond r_{off} the potential is now considered 0. See Fig. III.1.

The advantage of this form is that it provides the exact potential and force until the distance r_{on}. Switching also has the benefit that the force is continuous. The trade off for this is that when the potential is switched off a repulsive force results (Figure III.1(c)). The magnitude of this force should be small since r_{off} should be large enough to minimize other truncation effects. It can be further reduced by making the switching region larger.

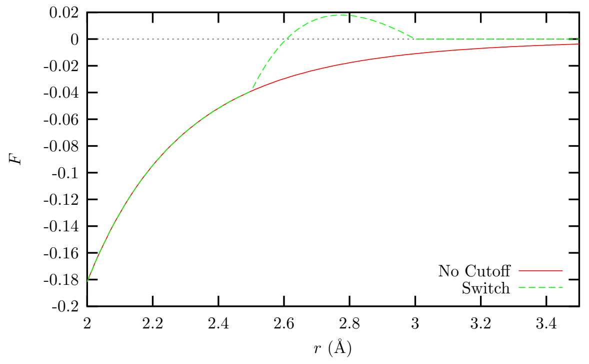
A typical cutoff is $r_{\text{off}} = 2.5\sigma$ which corresponds to approximately 1/60th of the well depth for a standard Lennard-Jones potential.



(a)



(b)



(c)

Figure III.1: Lennard-Jones cutoff schemes $r_{\text{on}} = 1$ and $r_{\text{off}} = 1$. Switching the potential doesn't appear to have much effect on the potential (a), taking a closer look at the cutoff region reveals the differences between the three schemes (b). The resulting force in the cutoff region is shown in (c). We can see that the switch potential creates a repulsive while the shifted potential has a discontinuity in the force.

3.6.1 `if` Statement

`if` statements allow us to selectively run, or not run, certain instructions. It has the form

```
if(expression) statement;
```

This will execute *statement* if *expression* is true. This means that condition is considered to be a logical expression and is false if it evaluates to `0`. To execute an arbitrary number of expressions we use the form

```
if(expression){
   statements;
}
```

This later form is in many cases preferable, even for a single statement, as it is easier to read and is easier to expand. However, you should be aware of the previous style.

A simple example using an if statement is if-test.c

```
/*************************************
 *Demonstrates the use of if.  Prints
 *the sign of the user input number.
 ************************************/

#include <stdio.h>

int main(){
  double x;
  printf("Input a number: \n");
  scanf("%lg",&x);

  if(x < 0){
    printf("x is negative\n");
  }

  if(x > 0){
    printf("x is positve\n");
  }

  if(x == 0){
    printf("x is 0\n");
  }

  return 0;
}
```

Only one of these `if` statements will execute.

Note: Be careful to use `==` and not `=` in your `if` statement. If you use the later an assignment will be performed and the result will be tested as a logical expression. So, following the example above,

```
if(x == 0)...
```

tests if `x` is equal to `0` while

```
if(x = 0)...
```

assigns `0` to `x`. The result, `0`, is evaluated which, in this case, is always zero. Thus the statement will always fail.

We now have enough tools to write a program to evaluate our Lennard-Jones potential. Our program will ask the user for the distance between the two particles and then calculate the appropriate interaction energy. lj-if.c will implement Equation III.2.

```
/*****************************************************************
 *Implements the Lennard-Jones interaction potential with a switch
 *cutoff.  The user is asked for the distance between the centers of
 *mass and the potential energy is returned.
 *****************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *r - the distance between the centers of mass
   *u - the potential energy.
   */
  double r, u;

  /**
   *ron - the distance at which we turn on switch.
   *roff - the distance at which we turn off the potential.
   */
  double ron = 2.5, roff = 3.;

  /**
   *sigma - the radius of the particles
   *epsilon - the depth of the attractive well
   */
  double sigma = 1, epsilon = 1;

  /**
   *r6 - the radius to the 6th power
   */
  double r6;

  printf("Input the distance between the centers of mass: \n");

  scanf("%lg",&r);

  if(r == 0){                             /* complete overlap */
    printf("Infinity energy!!!\n");
  }

  if(fabs(r) < ron){                      /* regular Lennard-Jones */
    r6 = pow(sigma/r, 6);
    u = 4*epsilon*(r6*r6 - r6);
    printf("U = %g\n",u);
  }

  if(fabs(r) >= ron && fabs(r) <= roff){  /* switched Lennard-Jones */
    r6 = pow(sigma/r, 6);
    u = 4*epsilon*(r6*r6 - r6);
    u *= ( pow(roff*roff - r*r, 2) * (roff*roff + 2*r*r -3*ron*ron))
          / pow(roff*roff - ron*ron,3);
    printf("U = %g\n",u);
  }

  if(fabs(r) > roff){                     /* cutoff */
    printf("U = 0\n");
  }

  return 0;
}
```

3.6.2 `else` Statement

It is easy to think of a situation where we have two cases and want to do something different for each case. Using an `if` statement we could write this as we did in the previous section

```
if(case1){
   statement1;
}
if(case2){
   statement2;
}
```

If the two cases are exclusive (they can't both happen at the same time) we can use the `else` statement instead of the second case.

```
if(case1){
   statement1;
} else {
   statement2;
}
```

We can apply this to our problem of testing the sign of numbers

```
if(x < 0){
   printf("x is negative\n");
} else {
  printf("x is positve\n");
}
```

This is not quite the same as our original program as `0` is counted as part of the positive numbers. We can still remedy this with `if-else` construct by *nesting* another `if-else` statement in the `else` part of the of the statement we already wrote. This is demonstrated in if-else-test.c

```
/************************************
 *Demonstrates the use of if-else
 *statements.  Prints the sign of the
 *user input number.
 ***********************************/

#include <stdio.h>

int main(){
  double x;
  printf("Input a number: \n");
  scanf("%lg",&x);

  if(x < 0){
    printf("x is negative\n");
  } else {
    if(x > 0){
      printf("x is positve\n");
    } else {
      printf("x is 0\n");
    }
  }
  return 0;
}
```

We apply this same form to our Lennard-Jones example in lj-if-else.c.

```
/*******************************************************************
 *Implements the Lennard-Jones interaction potential with a switch
 *cutoff.  The user is asked for the distance between the centers of
 *mass and the potential energy is returned.
 ******************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *r - the distance between the centers of mass
   *u - the potential energy.
   */
  double r, u;

  /**
   *ron - the distance at which we turn on switch.
   *roff - the distance at which we turn off the potential.
   */
  double ron = 2.5, roff = 3.;

  /**
   *sigma - the radius of the particles
   *epsilon - the depth of the attractive well
   */
  double sigma = 1, epsilon = 1;

  /**
   *r6 - the radius to the 6th power
   */
  double r6;

  printf("Input the distance between the centers of mass: \n");

  scanf("%lg",&r);

  if(r == 0){                       /* complete overlap */
    printf("Infinity energy!!!\n");

  } else {
    if(fabs(r) < ron){              /* regular Lennard-Jones */
      r6 = pow(sigma/r, 6);
      u = 4*epsilon*(r6*r6 - r6);
      printf("U = %g\n",u);

    } else {
      if(fabs(r) <= roff){          /* switched Lennard-Jones */
        r6 = pow(sigma/r, 6);
        u = 4*epsilon*(r6*r6 - r6);
        u *= ( pow(roff*roff - r*r, 2) * (roff*roff + 2*r*r -3*ron*ron))
          / pow(roff*roff - ron*ron,3);
        printf("U = %g\n",u);

      } else {                      /* cutoff */
        printf("U = 0\n");
      }
    }
  }
  return 0;
}
```

Notice that this logic statement has been simplified.

3.6.3 `else if` Statement

While an `if-else` statement is handy and works well if we only have to worry about two cases it becomes cumbersome when we need to work with multiple possible situations. Rather than using nested `if` statements we can use make use of one or more `else if` statements. The `if-else if-else` construct has the form

```
if(case1){
  statement1
} else if(case2){
   statement2
}
⋮
} else if(caseN-1){
   statementN-1
} else {
   statementN
}
```

Let's go back to our sign example. We can now write our sign example as in if-else_if-else-test.c.

```
/************************************
 *Demonstrates the use of if-else
 *statements.  Prints the sign of the
 *user input number.
 ************************************/

#include <stdio.h>

int main(){
  double x;
  printf("Input a number: \n");
  scanf("%lg",&x);

  if(x < 0){
    printf("x is negative\n");
  } else if(x > 0){
    printf("x is positve\n");
  } else {
    printf("x is 0\n");
  }
  return 0;
}
```

Finally, we can clean up our Lennard-Jones example as in lj-if-else_if-else.c.

```
/*****************************************************************
 *Implements the Lennard-Jones interaction potential with a switch
 *cutoff.  The user is asked for the distance between the centers of
 *mass and the potential energy is returned.
 *****************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *r - the distance between the centers of mass
   *u - the potential energy.
   */
  double r, u;

```

```
    /**
     *ron - the distance at which we turn on switch.
     *roff - the distance at which we turn off the potential.
     */
    double ron = 2.5, roff = 3.;

    /**
     *sigma - the radius of the particles
     *epsilon - the depth of the attractive well
     */
    double sigma = 1, epsilon = 1;

    /**
     *r6 - the radius to the 6th power
     */
    double r6;

    printf("Input the distance between the centers of mass: \n");

    scanf("%lg",&r);

    if(r == 0){                        /* complete overlap */
      printf("Infinity energy!!!\n");

    } else if(fabs(r) < ron){          /* regular Lennard-Jones */
      r6 = pow(sigma/r, 6);
      u = 4*epsilon*(r6*r6 - r6);
      printf("U = %g\n",u);

    } else if(fabs(r) <= roff){        /* switched Lennard-Jones */
      r6 = pow(sigma/r, 6);
      u = 4*epsilon*(r6*r6 - r6);
      u *= ( pow(roff*roff - r*r, 2) * (roff*roff + 2*r*r -3*ron*ron))
        / pow(roff*roff - ron*ron,3);
      printf("U = %g\n",u);

    } else {                           /* cutoff */
      printf("U = 0\n");
    }

    return 0;
  }
```

3.6.4 `switch` Statement

Another conditional or breaking statement that we will we may use (though not often in this course) is the `switch` statement. This statement is not capable of evaluating arbitrary logical expressions (thus limiting its usefulness). Rather, it executes a comparison between values. The general format of the statement is:

```
switch(expression) {
   case constant-expression :
      statements;
   ⋮
   case constant-expression :
      statements;
   default:
      statements;
}
```

Individual `case` statements are often separated by `break` statements. For example, consider the following

```
int i;

switch (i){
  case 0:
    printf("i = 0\n");
    break;
  case 1:
    printf("i = 1\n");
    break;
  default:
    printf("i is not 1 or 0\n");
}
```

If we leave out the `break` statement between the first two cases then `case 0` would fall through to `case 1`. That is, if `i` equal to `0` in our example we would see the output

```
i = 0
```

However, if we left out the break statement as in

```
int i;

switch (i){
  case 0:
    printf("i = 0\n");

  case 1:
    printf("i = 1\n");
    break;
  default:
    printf("i is not 1 or 0\n");
}
```

For `i` equal to `0` we would see the output

```
i = 0
i = 1
```

but for `i` equal to `1` we would still see

```
i = 1
```

We can make use of this behaviour to combine several case. All of these features are used in seasons.c.

```
/*****************************************************************
 *Given a date in the format dd/mm/yyyy it outputs the date in the
 *format month dd, yyyy and informs us of the season.  Note that we do
 *not check very hard if it is, in fact, a valid date.
 ****************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *day - day of the month
   *month - number of the month
   *year - four digit year
   */
  int day, month, year;

  /* get the date for the user */
  printf("Enter a date (dd/mm/yyyy): ");
  scanf("%d/%d/%d", &day, &month, &year);

  /* write out the name of the month */
  switch(month){
  case 1:
    printf("January");
    break;
  case 2:
    printf("February");
    break;
  case 3:
    printf("March");
    break;
  case 4:
    printf("April");
    break;
  case 5:
    printf("May");
    break;
  case 6:
    printf("June");
    break;
  case 7:
    printf("July");
    break;
  case 8:
    printf("August");
    break;
  case 9:
    printf("September");
    break;
  case 10:
    printf("October");
    break;
  case 11:
    printf("November");
```

```
      break;
    case 12:
      printf("December");
      break;
    default:
      /* they entered an incorrect value for the month */
      printf("Not a month: %d\n",month);
      return 1;
    }

    /* print out the rest of the date */
    printf(" %d, %d is in the", day, year);

    /* print out the season.  We don't worry too much about the exact
       that it changes but we try to come close */
    switch(month){
    case 1:
    case 2:
      printf(" winter");
      break;
    case 3:
      if(day > 21){
        printf(" spring");
      } else {
        printf(" winter");
      }
      break;
    case 4:
    case 5:
      printf(" spring");
      break;
    case 6:
      if(day > 21){
        printf(" summer");
      } else {
        printf(" spring");
      }
      break;
    case 7:
    case 8:
      printf(" summer");
      break;
    case 9:
      if(day > 21){
        printf(" fall");
      } else {
        printf(" summer");
      }
      break;
    case 10:
    case 11:
        printf(" fall");
```

```
  case 12:
    if(day > 21){
      printf(" winter");
    } else {
      printf(" fall");
    }
    break;
  }
  printf(".\n");
  printf("In Edmonton this is the");

  /* print out the season in Edmonton */
  switch(month){
  case 1:
  case 3:
  case 4:
  case 10:
  case 11:
  case 12:
    printf(" winter");
    break;
  case 5:
    printf(" spring");
    break;
  case 6:
  case 7:
  case 8:
    printf(" summer");
    break;
  case 9:
    printf(" fall");
    break;
  }
  printf(".\n");

  return 0;
}

```

3.7 Loops

One of the areas where computers excel and humans do not is repetitive tasks. Not only are they more accurate they are also significantly faster. We can access this important feature of computers in C by using *loops*.

There are three different loop structures in C: `while`, `do` and `for`. Each of these loops has its own advantages and natural uses. However, we will see that the `for` loop is the most powerful and the one we will use more of the time.

From the lab you have, or will have seen, one obvious use of loops, iterating over a range of numbers. Other uses include, but are not limited to, summations of series and iterative solutions to equations. Anywhere we need to repeat a task we can use a loop.

Series Summation: π

Numerical and approximate methods for calculating π have been around for over 2000 years. Before the advent of computers the value of π was known to over 700 decimal places (not binary places) and was considered one of the greatest human achievements in its time. It was later shown that there was an error few hundred places in. Since computers have appeared on the scene π has been calculated to over one trillion decimal places using advances algorithms, computers and arbitrary precision floating point numbers.

Using the built-in `double` type in C we can only hold the first fifteen decimal places or so. However, we can still use the following method to do it. The series representation of $\arctan(x)$ is

$$\arctan(x) = x - x^3/3 + x^5/5 - x^7/7 \ldots \tag{III.3}$$

Now, we could apply this directly using

$$\pi = 4\arctan(1) = 4(1 - 1/3 + 1/5 - 1/7 \ldots) \tag{III.4}$$

but this converges very slowly. It would take us a few hundred iterations to get two or three decimal places. However, we can write

$$\begin{aligned}\pi &= 16\arctan(1/5) - 4\arctan(1/239)\\ &= 16/5 - 4/239 - (16(1/5)^3 - 4(1/239)^3)/3 + (16(1/5)^5 - 4(1/239)^5)/5 - (16(1/5)^7 - 4(1/239)^7)/7 \ldots\\ &= \sum_{i=0}^{\infty}(-1)^i\frac{16(1/5)^{2i+1} - 4(1/239)^{2i+1}}{2i+1}\end{aligned} \tag{III.5}$$

which converges much faster at a rate of about 1.4 decimal places per iteration.

Root Finding: Bisection Method

Often, when faced with a physical problem, we are required to find the roots of an equation. An example of an iterative method to do this is the bisection method.

To illustrate the bisection method we will find the positive root of the function

$$y = f(x) = x^2 - 1 \tag{III.6}$$

with an accuracy in x of 10^{-1}. The bisection method is as follows

1. First bracket the root of the equation $y = f(x)$ from above and below at points $x = a$ and $x = b$. That is, if $a > 0$ then we must have $b < 0$. For our example try $a = 0.5$ and $b = 1.75$. See Fig. III.2(a).

2. Calculate the mid-point of the bracket, $c = (a + b)/2$, and the value of $f(c)$. In this case $c = 1.125$ and $f(c) = 1.265625$. See Fig. III.2(b).

3. If $f(c) > 0$ replace the upper bound with c otherwise replace the lower bound with c. In our example $f(c) > 0$ so we set $b = 1.125$. See Fig. III.2(c).

4. If $f(c) = 0$ or if $|a - b| < \epsilon$ where ϵ is our maximum error we have found our root. If not return to Step 2. We have $|a - b| > \epsilon$ so we goto Step 2. See Fig. III.2(d).

3.7.1 `while` Loop

The `while` loop repeats a set of instructions while a logical expression is true, hence the name. This has the general form

```
while(expression){
```

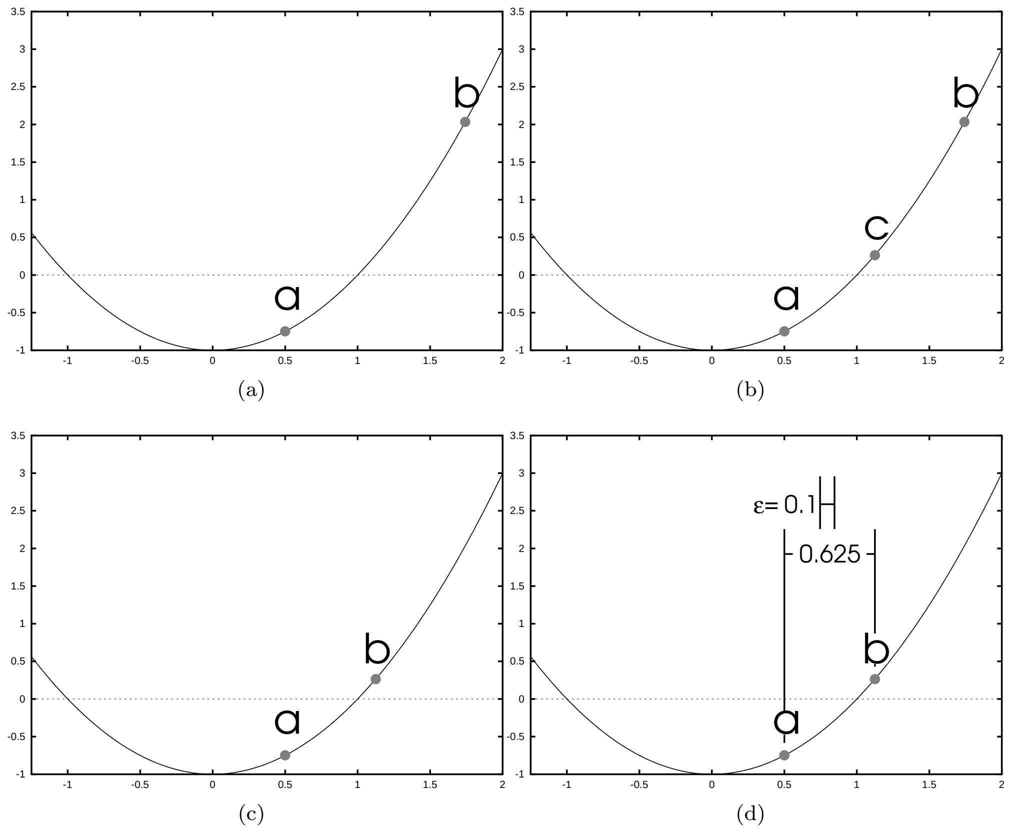


Figure III.2: Illustration of the bisection method for $y = x^2 - 1$.(a) The root is first bracketed by $a = .5$ and $b = 1.75$. (a) the midpoint $c = 1.125$ is found. (c) $f(c) > 0$ so set $b = c = 1.125$. (d) $|a - b| > \epsilon$ so goto (a).

```
    statements;
    ⋮
}
```

We have to be careful with this. If *expression* is always true then our loop will never exist and we will have an *infinite loop.*

If you think back to the first lab you saw a program call `yes`. This program merely printed `y` to the screen over and over. We can now write that program ourselves. You can find it in myyes.c

```
/***********************************************************
 *A program that prints "y" over and over.
 ***********************************************************/

#include <stdio.h>

int main(){
  while(1){
    printf("y\n");
  }
  return 0;
}
```

Note: If and when you get caught in an infinite loop (it happens to us all) use `<ctrl>-c` to force the program to quit.

expression can be any logical statement we like. Often we would like to iterate through the loop a set number of times. We can do this with the use of a *counter* variable that we increment each time through the loop. When we are at the top of the loop we check if it is still smaller than the number of iterations we wish to do. For example, in myyes10.c we print out `y` 10 times.

```
/***********************************************************
 *A program that prints "y" over and over.
 ***********************************************************/

#include <stdio.h>

int main(){
  int i, n =10;
  i = 0;
  while(i<n){
    printf("y\n");
    i++;
  }
  return 0;
}
```

This gives us the tools to sum a simple series. Since we typically (but not always) know how many times we want to iterate our summation we can use a counter and a `while` loop to achieve the number of iterations we require. Let's say that we want to iterate through our π series ten times. This should give us about 14 digits of π. We might write this mathematically as

$$\pi = \sum_{i=0}^{9} (-1)^i \frac{16(1/5)^{2i+1} - 4(1/239)^{2i+1}}{2i+1} \tag{III.7}$$

This may be translated into C as in pi-while.c.

```
/**********************************************
 *A program to calculate a few digits of pi.
 *********************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *pi - will hold the value of pi.  we must initialize this to 0
   *sign - this determines whether we are adding or subtracting each term.
   */
  double pi=0,sign=-1;

  /**
   *i - our counter variable
   *n - the number or iterations
   */
  int i, n = 10;

  /* initialize i */
  i =0;
  while(i<n){
    /* flip the sign of sign */
    sign*=-1;
    /* add to our previous total */
    pi+= sign* (16*pow(.2,2*i+1) - 4*pow(1./239.,2*i+1))/(i*2 + 1);
    /* print out our current iteration and total */
    printf("%2d\t%.20g\n",i,pi);
    /* increment i */
    i++;
  }

  /* print out the true value of pi truncated at a few decimal places */
  printf("true\t%.20g\n",3.14159265358979323846);
  /* print out a number to show the decimal places */
  printf("\t%.20Lg\n",0.12345678901234567890l);
  return 0;
}
```

We may not always know how many times we wish to iterate a loop. For example, when using the bisection method it is rare that we know how many times we have to iterate our algorithm to find our root with the desired accuracy. In the cases where we do know we probably can find the root easier than we could the number of iterations. For the bisection method we exit the loop when the difference in our a and b points is less that our desired accuracy ϵ. In fact, in bisect-while.c we don't have a counter variable at all.

```
/*********************************************************
 *This program uses the bisection method to find the positive root of
 *the equation
 *
 *y = x^2 -1
 ***********************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *y_a - the value of a*a -1
   *y_b - the value of b*b -1
   *y_c - the value of c*c -1.
   */
  double y_a, y_b, y_c;

  /**
   *a - one bracket
   *b - the other bracket
   *c - thhe midpoint
   */
  double a = .5, b = 1.75, c;

  /**
   *epsilon - the desired accuracy of our root.
   */
  double epsilon = .1;

  /* initialize y_a and y_b */
  y_a = a*a-1;
  y_b = b*b-1;

  /* determine if they are the same sign, we cannot continue if they
     are */
  if(y_a*y_b > 0){
    printf("y(a=%g) and y(b=%g) are the same sign\n", y_a, y_b);
    return 1;
  }

  /* find c and y_c */
  c = (a + b) /2;
  y_c = c*c - 1;
  /* begin our loop */
  while(fabs(a-b) > epsilon || y_c == 0){

    /* determine if y_c is the same sign as y_a or y_b */
    if(y_a*y_c > 0){
      a = c;
      y_a = y_c;
    } else {
      b = c;
      y_b = y_c;
    }

    /* calculate the new midpoint */
```

```
      c = (a + b) /2;
      y_c = c*c - 1;

      /* print information about where we are */
      printf("a = %9.7g\tb = %9.7g\n",a, b);
      printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);
    }


    return 0;
  }
```

3.7.2 `do` Loop

The `do` loop is similar in structure and use to the `while` loop but differs in one important aspect; the continuation check is done at the end of the loop, not the beginning. This is important as it means that the loop will always run at least once. This makes it natural for problems such as bisection where we wish to iterate through the problem at lest once before checking our conditional statement.

The general structure of the `do` loop is

```
do{
   statements;
}while(expression);
```

Due to its structure a `do` loop may also be called a `do-while` loop. One should also be careful when writing or reading C programs not to confuse `do` and `while` loops. For example, the `{}` is optional so it is possible to write

```
do printf(‘‘i=%d\n’’,i++);
while(i<n);
```

which is not the same as

```
printf(‘‘i=%d\n’’,i++);
while(i<n);
```

The later statement may, in fact, be an infinite loop, depending on the initial value of `i`.

Casting our summation of π into a `do` loop we have the program pi-do.c

```
/***********************************************
 *A program to calculate a few digits of pi.
 **********************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *pi - will hold the value of pi.  we must initialize this to 0
   *sign - this determines whether we are adding or subtracting each term.
   */
  double pi=0,sign=-1;

  /**
```

```
     *i - our counter variable
     *n - the number or iterations
     */
    int i, n = 10;

    /* initialize i */
    i =0;
    do{
      /* flip the sign of sign */
      sign*=-1;
      /* add to our previous total */
      pi+= sign* (16*pow(.2,2*i+1) - 4*pow(1./239.,2*i+1))/(i*2 + 1);
      /* print out our current iteration and total */
      printf("%2d\t%.20g\n",i,pi);
      /* increment i */
      i++;
    }while(i<n);

    /* print out the true value of pi truncated at a few decimal places */
    printf("true\t%.20g\n",3.14159265358979323846);
    /* print out a number to show the decimal places */
    printf("\t%.20Lg\n",0.12345678901234567890l);
    return 0;
  }
```

Notice that the output of the program is exactly the same as for the `while` loop version. The only difference is that the loop will always execute at least once regardless of the value of `n`.

bisect-do.c is a somewhat different story as it lends itself much better to the `do` format.

```
  /**********************************************************
   *This program uses the bisection method to find the positive root of
   *the equation
   *
   *y = x^2 -1
   **********************************************************/

  #include <stdio.h>
  #include <math.h>

  int main(){
    /**
     *y_a - the value of a*a -1
     *y_b - the value of b*b -1
     *y_c - the value of c*c -1.
     */
    double y_a, y_b, y_c;

    /**
     *a - one bracket
     *b - the other bracket
     *c - thhe midpoint
     */
    double a = .5, b = 1.75, c;

    /**
     *epsilon - the desired accuracy of our root.
     */
    double epsilon = .1;

    /* initialize y_a and y_b */
    y_a = a*a-1;
    y_b = b*b-1;

    /* determine if they are the same sign, we cannot continue if they
       are */
    if(y_a*y_b > 0){
      printf("y(a=%g) and y(b=%g) are the same sign\n", y_a, y_b);
      return 1;
    }

    /* begin our loop */
    do{
      /* calculate the new midpoint */
      c = (a + b) /2;
      y_c = c*c - 1;

      /* print information about where we are */
      printf("a = %9.7g\tb = %9.7g\n",a, b);

      /* determine if y_c is the same sign as y_a or y_b */
      if(y_a*y_c > 0){
        a = c;
        y_a = y_c;
      } else {
        b = c;
        y_b = y_c;
      }

      printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);

    }while(fabs(a-b) > epsilon || y_c == 0);

    /* calculate the new midpoint */
    c = (a + b) /2;
    y_c = c*c - 1;
    /* print information about where we are */
    printf("a = %9.7g\tb = %9.7g\n",a, b);
    printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);



    return 0;
  }
```

Note that the statements immediately before and after the loop are now different. Also the order of the statements inside the loop and the output is different as well. In this case the `do` fits the structure of our algorithm better. This does not necessarily reduce the amount of code that we write but does make the translation into C code somewhat easier.

3.7.3 `for` Loop

The most versatile, powerful and commonly used loop in C is the `for` loop. In structure and function it is much like an extended `while` loop. In fact, it can always be used in place of `while` in whatever problem we might be faced with.

The general format of the `for` loop is

```
for(initialization;continuation;increment){
   statements;
}
```

`initialization` is executed only the first time through the loop. *`continuation`* is equivalent to the *`expression`* in our `while` loop. A `for` loop will continue while *`continuation`* is true. Finally, the *`increment`* statement is executed every time through the loop *except* the first. This is commonly used to increment our counter variable.

Note: Be careful when you write `for` statements that you separate *`initialization`*, *`continuation`* and *`increment`* with `;`s and not `,`s. The compiler will usually catch this common mistake but `,` is also an operator and is most often used in `for` loops.

A simple `for` loop might look something like this

```
for(i = 0; i < n; i++){
  printf(``i = %d\n'',i);
}
```

The equivalent statement with a `while` loop would be

```
i = 0;
while(i < n){
  printf(``i = %d\n'',i);
  i++;
}
```

Notice that we may use either `i++` or `++i` to increment our counter in either case. It is the side-effect that is important (incrementing) not the value of `i` as the value of *`continuation`* as already been determined. Most programmers choose to use `i++`. Perhaps this is because it looks more like `i` is being assigned a new value.

Using a `for` loop we may rewrite our π summation as in pi-for.c

```
/**********************************************
 *A program to calculate a few digits of pi.
 *********************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *pi - will hold the value of pi.  we must initialize this to 0
   *sign - this determines whether we are adding or subtracting each term.
   */
  double pi=0,sign=-1;

  /**
   *i - our counter variable
   *n - the number or iterations
   */
  int i, n = 10;

  /* initialize i */
  for(i =0;i<n;i++){
    /* flip the sign of sign */
    sign*=-1;
    /* add to our previous total */
    pi+= sign* (16*pow(.2,2*i+1) - 4*pow(1./239.,2*i+1))/(i*2 + 1);
    /* print out our current iteration and total */
    printf("%2d\t%.20g\n",i,pi);
    /* increment i */
  }

  /* print out the true value of pi truncated at a few decimal places */
  printf("true\t%.20g\n",3.14159265358979323846);
  /* print out a number to show the decimal places */
  printf("\t%.20Lg\n",0.12345678901234567890l);
  return 0;
}
```

We can also apply a `for` loop to our bisection problem as in bisect-for.c

```
/************************************************************
 *This program uses the bisection method to find the positive root of
 *the equation
 *
 *y = x^2 -1
 ************************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *y_a - the value of a*a -1
   *y_b - the value of b*b -1
   *y_c - the value of c*c -1.
   */
  double y_a, y_b, y_c;

  /**
   *a - one bracket
   *b - the other bracket
   *c - thhe midpoint
   */
  double a = .5, b = 1.75, c;

```

```
    /**
     *epsilon - the desired accuracy of our root.
     */
    double epsilon = 1e-30;

    /* initialize y_a and y_b */
    y_a = a*a-1;
    y_b = b*b-1;

    /* determine if they are the same sign, we cannot continue if they
       are */
    if(y_a*y_b > 0){
      printf("y(a=%g) and y(b=%g) are the same sign\n", y_a, y_b);
      return 1;
    }

    /* find c and y_c */
    c = (a + b) /2;
    y_c = c*c - 1;
    /* begin our loop */
    for(;fabs(a-b) > epsilon || y_c == 0;){

      /* determine if y_c is the same sign as y_a or y_b */
      if(y_a*y_c > 0){
        a = c;
        y_a = y_c;
      } else {
        b = c;
        y_b = y_c;
      }

      /* calculate the new midpoint */
      c = (a + b) /2;
      y_c = c*c - 1;

      /* print information about where we are */
      printf("a = %9.7g\tb = %9.7g\n",a, b);
      printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);
    }


    return 0;
  }
```

It is important to note that in the above code *`initialization`* and *`increment`* are missing. Since these arguments are missing nothing is done when these would typically be calculated. All the arguments of a `for` loop are, in fact, optional. When the *`initialization`* and *`increment`* are omitted the `for` loop is exaclty equivalent to a `while` loop. Using a `for` loop as we did, however, is considered poor style as a `while` loop is more obvious and easier to read.

The above example doesn't mean that we should exclude the `for` loop from the bisection method or other algorithms that can be written with just a `while` loop. Consider the case where $\epsilon = 1 \times 10^{-30}$ and try changing and running the `bisect-for.c` code. The bisection method is guareenteed to converge except when we run into problems with the finite precision of numbers. This ϵ is to small and we enter an infinte loop. We would like to be able to catch such errors. A simple way is to limit the number of iterations allowed. Once we have exceeded the maximum number of iterations we assume that the algorithm failed or was programmed incorrectly. This is implemented in bisect-for-2.c

```
/*********************************************************
 *This program uses the bisection method to find the positive root of
 *the equation
 *
 *y = x^2 -1
 *********************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *y_a - the value of a*a -1
   *y_b - the value of b*b -1
   *y_c - the value of c*c -1.
   */
  double y_a, y_b, y_c;

  /**
   *a - one bracket
   *b - the other bracket
   *c - thhe midpoint
   */
  double a = .5, b = 1.75, c;

  /**
   *epsilon - the desired accuracy of our root.
   */
  double epsilon = 1e-30;

  /**
   *i - counter
   *maxiter - maximum number of iterations
   */
  int i, maxiter = 100;

  /* initialize y_a and y_b */
  y_a = a*a-1;
  y_b = b*b-1;

  /* determine if they are the same sign, we cannot continue if they
     are */
  if(y_a*y_b > 0){
    printf("y(a=%g) and y(b=%g) are the same sign\n", y_a, y_b);
    return 1;
  }

  /* find c and y_c */
  c = (a + b) /2;
  y_c = c*c - 1;
  /* begin our loop */
  for(i=0;(fabs(a-b) > epsilon || y_c == 0) && i < maxiter;i++){

    /* determine if y_c is the same sign as y_a or y_b */
    if(y_a*y_c > 0){
      a = c;
      y_a = y_c;
    } else {
      b = c;
      y_b = y_c;
    }

    /* calculate the new midpoint */
    c = (a + b) /2;
    y_c = c*c - 1;

    /* print information about where we are */
    printf("a = %9.7g\tb = %9.7g\n",a, b);
    printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);
  }

  /* test if we found the root or not */
  if(i == maxiter){
    printf("Maximum iterations execeeded : %d.  No roots found.\n",maxiter);
  }
  return 0;
}
```

The , Operator

Although we don't need to use it very often the , operator can make our life and our `for` loops easier. Expressions with this operator are of the form

expression1,expression2

expression1 is computed and the value discarded. *expression2* is then calculated the value kept as the value for the entire expression. If *expression1* has no side-effect (e.g. there is no assignment) then *expression1* is useless.

Use of the , operator are demonstrated in comma.c

```
/*****************************************************************
 *Illustrates some uses of the comma operator by summing the integers
 *from 0 to 9 inclusive
 *****************************************************************/

#include <stdio.h>

int main(){
  int i, sum, n = 10;

  printf("\nFor Loop 1\n");
  for(i = 0, sum = 0;i<n; i++){
    sum+=i;
    printf("%3d -- sum = %3d\n",i, sum);
  }

  printf("\nFor Loop 2\n");
  for(i = 0,sum = 0; sum+=i,i<n; i++,printf("%3d -- sum = %3d\n",i, sum));

  printf("\nWhile Loop 1\n");
  i=0, sum=0;
  while(sum+=i,printf("%3d -- sum = %3d\n",i, sum),i++,i<n);

  return 0;
}
```

Note: The `,` operator cannot be used everywhere. An important place it cannot be used is in arguments to functions like `pow()`. Here the `,` has a different meaning and will be interpretted as extra arguments for the function.

3.7.4 Restricted Jumps

Some algorithms are easiest to implement if we can abandon a single iteration of loop or break out the loop altogether . This can be done with the keywords `break` and `continue`.

`break` Statement

`break` takes us from inside of a loop or `switch` statement to the point just outside the end of the loop or `switch` statement. For example, break.c

```
/*****************************************************************
 *Illustrates the use of 'break'.  This code will never count past
 *'bignumber'.
 *****************************************************************/

#include <stdio.h>

int main(){
  int i, bignumber = 10, n = 100;
  for(i=0; i<n; i++){
    printf("%d ",i);
    if(i>=bignumber){
      printf("\nI don't know how to count past %d!?!",bignumber);
      break;
    }
  }
  printf("\n");
  return 0;
}
```

A good use of the `break` statement is our bisection method. This is used in bisect-break.c

```
/************************************************************
```

```
 *This program uses the bisection method to find the positive root of
 *the equation
 *
 *y = x^2 -1
 ***********************************************************/

#include <stdio.h>
#include <math.h>

int main(){
  /**
   *y_a - the value of a*a -1
   *y_b - the value of b*b -1
   *y_c - the value of c*c -1.
   */
  double y_a, y_b, y_c;

  /**
   *a - one bracket
   *b - the other bracket
   *c - thhe midpoint
   */
  double a = 0, b = 4, c;

  /**
   *epsilon - the desired accuracy of our root.
   */
  double epsilon = .1;

  /* initialize y_a and y_b */
  y_a = a*a-1;
  y_b = b*b-1;

  /* determine if they are the same sign, we cannot continue if they
     are */
  if(y_a*y_b > 0){
    printf("y(a=%g) and y(b=%g) are the same sign\n", y_a, y_b);
    return 1;
  }

  /* find c and y_c */
  c = (a + b) /2;
  y_c = c*c - 1;
  /* begin our loop */
  while(fabs(a-b) > epsilon){
    if(y_c == 0){
      printf("%9.7g is an exact root\n",c);
      break;
    }

    /* determine if y_c is the same sign as y_a or y_b */
    if(y_a*y_c > 0){
      a = c;
      y_a = y_c;
    } else {
      b = c;
      y_b = y_c;
    }

    /* calculate the new midpoint */
    c = (a + b) /2;
    y_c = c*c - 1;

    /* print information about where we are */
    printf("a = %9.7g\tb = %9.7g\n",a, b);
    printf("y(c=%9.7g)=%9.7g\n\n", c, y_c);
  }


  return 0;
}
```

`continue` Statement

`continue` takes us to the end of the last line in the loop. This is still in the loop so we have not exited it yet. For example, continue.c

```
/*****************************************************************
 *Illustrates the use of 'coninue'.  This code will ignore negative
 *numbers.
 ****************************************************************/

#include <stdio.h>

int main(){
  int i, n = 100;
  for(i=-10; i<n; i++){
    if(i<0){
      printf("I'm ignoring %d\n",i);
      continue;
    }
    printf("%d ",i);
  }
  printf("\n");
  return 0;
}
```

Note: The `goto` statement does exist in C. Don't use it.