Reminder

• I am back!

• HW04 due Thursday 22 Feb electronically by noon

• HW grades are coming. Please be patient.

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm
Review and Outline

• Last time:
  – C program structure
  – Intro to a basic C program
  – Defining simple variables and doing arithmetic
  – Formatted input/output via printf() and scanf()
  – How variables’ values are stored on a computer
  – More on formatted I/O

• Today:
  – Comments
  – Functions
  – Conditional structures:
    • if, if/else, if/elseif/else
    • switch/case
  – Loops
    • Count-controlled loops: for()
    • Conditioned controlled loops: while(), do
  – Random numbers
Comments Inside Your Code
/* Traditional C defines comments by using opening and closing comment markers as shown in this example. These comments may span multiple lines. */

// However, it is often better to write // multiple line comments in this way // to make the extended comments more // clear in the text of your code

int a = 15; // C99 and C++ allow single line comments w/ the double slash int b = 6; /* this type of comment is allowed, but not preferred */
Functions
Introduction to Functions

• C is a simple language

• Its utility is extended through the use of re-usable functions

• Some functions are found in standard libraries like \texttt{stdio.h} and \texttt{math.h}

• Users can write functions too – for many good reasons!
“Intrinsic” Functions:
While the text makes reference to intrinsic functions, it’s more accurate to call them the C Standard Library Functions.

These are not part of the C language, but reside in a library of useful functions that evolved along with C and is now also standardized across compiler distributions and hardware platforms.

To use these functions it was necessary to first include a header file so the compiler would recognize their input/output interfaces. The actual code is pre-compiled and is linked to your source code to make a working program.
C Standard Library Functions

“Intrinsic” Functions:
While the text makes reference to intrinsic functions, it's more accurate to call them the C Standard Library Functions.

These are not part of the C language itself, but rather, a library of functions that have evolved over time. These functions are now also part of the standard compiler's and hardware's design.

To use these functions, you would reference the appropriate header file — the actual code is linked to your source code to make a working program.

Common C Standard Libraries we will use:
  • stdio.h
  • math.h

Best to get in the habit of including these in every program you write!

You don’t even know WHERE these files are on galileo in order to use them.
Note that C's math functions take and return parameters that are of type `double`:

```c
double sqrt(double x); // prototype for sqrt function
```

You'll find a line like this in the math.h header file.

The compiler reads this from `<math.h>`, then when it encounters a call to `sqrt()` in your code, it can check that you are calling it correctly:

- giving the right number of parameters,
- using the output value properly
- etc...

For example:

```c
int i = sqrt(10.);  // This will generate a warning.
float q = sqrt(10., 2.); // This will generate an error.
```
User-Defined Functions

• Why would one want to write their own functions?

  – Avoid duplicating the same code many times within a program: streamlined simplicity

  – Re-usable functions are easier to maintain and modify

  – Functions are portable – can be used in other programs

  – Simplicity: code something intricate once and call it via a simple single line rather than multi-line complications
User-Defined Functions: Example Use Case

Why Write Functions?

```c
#include <stdio.h>
#include <math.h>
int main () {
    double x0 = 0.0;
    double y0 = 0.0;
    double x1 = 1.0;
    double y1 = 2.0;
    double x2 = 4.0;
    double y2 = 1.0;
    double x3 = 3.0;
    double y3 = 0.0;

    double d01 = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) );
    printf ("d01 is %f\n",d01);

    double d12 = sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
    printf ("d12 is %f\n",d12);

    double d23 = sqrt( (x3-x2)*(x3-x2) + (y3-y2)*(y3-y2) );
    printf ("d23 is %f\n",d23);

    return(0);
}
```

Consider this program, which calculates the total distance for a trip through four points.

Notice that the program repeatedly uses similar statements to calculate the lengths of the segments of the trip.

If we ever needed to change the program (say, to print travel times) we'd need to remember to modify each of these statements.

No functions – repetitive!
Three main parts: Prototype, Definition, and Calls

User-Defined Functions: Use Case Example

Designing a Function:

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

double distance(double xstart, double ystart, 
                double xend, double yend);

int main () {
    // (coordinates omitted for brevity)...

double d01 = distance(x0,y0,x1,y1);
printf("d01 is \%f\n",d01);

double d12 = distance(x1,y1,x2,y2);
printf("d12 is \%f\n",d12);

double d23 = distance(x2,y2,x3,y3);
printf("d23 is \%f\n",d23);

return(0);
}
```

Function prototype

Using ("calling") the function

We could easily modify the distance function to return, say, travel time (adjusted for a headwind from a given direction!). We'd only need to make the change in one place: the function definition.
Prototype, Arguments and Return:

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

double distance(double xstart, double ystart, double xend, double yend);

int main () {
    // (coordinates omitted for brevity)...

double d01 = distance(x0,y0,x1,y1);
    printf("d01 is \%f\n",d01);

double d12 = distance(x1,y1,x2,y2);
    printf("d12 is \%f\n",d12);

double d23 = distance(x2,y2,x3,y3);
    printf("d23 is \%f\n",d23);

    return(0);
}
```

The prototype defines the syntax for the function. (What arguments it takes, and what type of data it returns.)

The names of the arguments in prototype, function call and function definition don’t need to match, but the types do.

Our function takes four “doubles” as arguments, and returns a double.

```c
double distance (double xinit, double yinit, double xfinal, double yfinal) {
    double d;
    d = sqrt( (xfinal-xinit)*(xfinal-xinit) + (yfinal-yinit)*(yfinal-yinit) );
    return(d);
}
```
User-Defined Functions: Reusable?

Making Functions Re-useable:

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

double distance(double xstart, double ystart, 
    double xend, double yend );

int main () {
    // (coordinates omitted for brevity)...
    double d01 = distance(x0,y0,x1,y1);
    printf ("d01 is \%.1f\n",d01);
    double d12 = distance(x1,y1,x2,y2);
    printf ("d12 is \%.1f\n",d12);
    double d23 = distance(x2,y2,x3,y3);
    printf ("d23 is \%.1f\n",d23);
    return(0);
}

double distance (double xinit, double yinit, 
    double xfinal, double yfinal ) {
    double d;
    d = sqrt( (xfinal-xinit)*(xfinal-xinit) + 
        (yfinal-yinit)*(yfinal-yinit) );
    return(d);
}
```

Some day, the prototype for your function could be moved into an external header file, to be included as needed...

and the function itself could be added to your own library of functions, for later use. We'll see how to do this later.

```c
mylib.h

distance()

etc...
```
Conditional Executions in C
Linear Code Execution in C

• Simple pieces of code have lines of instructions that are executed linearly, calling external functions like `printf`:

```c
#include <stdio.h>

int main(void)
{
    int num = 5;
    printf("Hello, world. num = %d\n", num);
    return 0;
}
```

• This is not typical however
• Most programs need to do more intricate things in their execution
Conditional Executions in C: if statements

"if" Statement Syntax:
A simple "if" statement can be written in two different ways. Here's the more general way to write one:

Syntax:
```c
if (CONDITION) {
    BLOCK of statements
}
```
No semicolon.

Example:
```c
if (a > 1) {
    printf("Hello There!\n");
    b = a * 2;
    printf("b is: %d\n", b);
}
```

Alternatively, if you only have one line in your block of statements, you can omit the curly brackets and write it like this:

Syntax:
```c
if (CONDITION)
    statement;
```

Example:
```c
if (a > 1)
    printf("Hello There!\n");
```
Conditional Checks

Relational and Logical Operators:

These operators test or combine logical expressions. The answer to a test is either true (not 0) or false (0). Any non-zero value is considered true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical NOT. Invert a test or true/false value</td>
<td>1</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or equal</td>
<td>2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or equal</td>
<td>2</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
<td>3</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality</td>
<td>3</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Remember: Use parentheses to prevent precedence perplexity.
Sometimes a simple “if” statement isn't enough. You may want to choose between two or more different blocks of code, based on some test.

In that case, you can add an “else” clause to your “if” statement.

The second block of code will only be executed when CONDITION is false.

For more complicated cases, you can add multiple “else if” clauses.

Only one of these blocks (the first one whose CONDITION is “true”) will be executed. The others will be ignored.

The else block, if present, will be executed if none of the CONDITIONs are met.
Program Flow with “if” Statements:

The “if” statement is one of the ways you can control the flow of your program.

```c
if (i==1) {
    BLOCK of statements
} else if (i==2) {
    BLOCK of statements
} else if (i==3) {
    BLOCK of statements
} else {
    BLOCK of statements
}
```
Conditional Execution: *switch* and *case*

Switch Statements:

It's common for programs to look at a value and use it to decide which one of many alternative blocks of code to execute.

For convenience, C provides another construct for this special case: the *switch* statement.

`break` means "jump out of the switch statement and continue with the rest of the program".

If there's a *default* case, it matches any value.

```
switch (EXPRESSION) {
    case VALUE1:
        BLOCK of statements
        break;
    case VALUE2:
        BLOCK of statements
        break;
    case VALUE3:
        BLOCK of statements
        break;
    default:
        BLOCK of statements
}
```
Controlling Flow with “break”:

Using `break` in a switch statement gives us a lot of control over the flow of our program.

```c
switch (letter) {
    case 'A':
    case 'a':
        printf("A is for Apple\n");
        break;    // letter = 'A' or 'a'
    case 'B':
    case 'b':
        printf("B is for Bear\n");
        break;
    default:
        printf("Unknown letter\n");
}
```

If we omit a `break`, the program will continue to work its way through the switch statement, possibly matching other cases.

This lets us do the same thing for several different cases, working like an OR statement: `'A' || 'a'`.

We can also make one case a superset of another. Here, when `letter='A'`, the program will print:

Capitol A is for Apple
if/else versus switch/case

```
switch (letter) {
    case 'A':
    case 'a':
    case '1':
        printf("A\n");
        break;
    case 'B':
    case 'b':
    case '2':
        printf("B\n");
        break;
}
```

Everything that can be accomplished with `switch` can also be done with a sufficiently complicated set of `if` statements. The two snippets shown here do the same thing, but notice how much more readable the switch statement is.

```
if (letter=='A'||letter=='a'||letter=='1'){
    printf("A\n");
} else if (letter=='B'||letter=='b'||letter=='2'){
    printf("B\n");
}
```
The \texttt{?} Operator

As a shorter alternative to the regular if statement, C offers the special operator \texttt{?:}. This is a \textit{ternary} operator, meaning that it takes \textit{three} arguments. The syntax is as follows:

\begin{align*}
\text{CONDITION} \, \texttt{?} \, \text{STATEMENT} \, : \, \text{STATEMENT} \; ;
\end{align*}

\textbf{Example:}

\begin{verbatim}
(a==b) \, ? \, printf("it's true\n") \, : \, printf("it's false\n");
\end{verbatim}

This is just equivalent to the \texttt{if} statement shown at the right:

\begin{verbatim}
if (a==b)
  printf("it's true\n");
else
  printf("it's false\n");
\end{verbatim}
DANGER: ‘=’ versus ‘==’

Be careful not to confuse the = (assignment) operator with the == (equality comparison) operator. This is one of the most common C typos.

The code below produces unexpected results. Why?

```c
int a=0;
int b=1;
if (a=b)
    printf("they are equal\n");
else
    printf("they are not equal\n");
```

Prints: “they are equal”!

The programmer should have used “a==b”!

```c
int a=0;
int b=0;
if (a=b)
    printf("they are equal\n");
else
    printf("they are not equal\n");
```

Prints: “they are not equal”!

The value returned by the assignment operation “a=b” is just the left-hand side of the assignment (“a”, in this case) after the operation has completed (after “a” has been set equal to “b”).

```
if (a=b) is equivalent to
if (a)
```

18
DANGER: “=” versus “==”

Be careful not to confuse the = (assignment) operator with the == (equality comparison) operator. This is one of the most common C typos.

The code below produces unexpected results. Why?

```c
int a=0;
int b=1;
if (a=b)
    printf("the value of a is %d\n",a);
else
    printf("the value of a is not equal\n");
```

Take-away message:

Be careful to use “==” in conditionals and not “=“.

```c
int a=0;
int b=0;
if (a=b)
    printf("the value of a is %d\n",a);
else
    printf("they are not equal\n");
```

```c
if (a=b) is equivalent to
a=b;
if (a)
```
Return Values and Tests

- Values are returned for all conditional checks:
  - if false, the value is zero ..... the value of the expression $(5 < 3)$ is 0
  - if true, the value is not zero ..... the value of the expression $(5 > 3)$ is 1

- We can use these characteristics in our code:

```c
// fopen returns a non-NULL pointer if successful
FILE* inFile;
inFile = fopen("grades.dat","r"); // open grades.dat

if (inFile==NULL) {
    // exit program if file not found
    printf("Error: grades.dat not found!!\n");
    return(1);
}
```

The “if” statement could alternatively be written like this:

```c
if (!inFile) {
    printf("Error...
");
}
Return Values and Tests: From Functions

• One can call a function that performs some test

Returning Zero for Success:

It's common practice for functions returning an integer status value (instead of returning data) to return zero for “Success”, and non-zero to indicate an error. You'll often see code that takes advantage of this convention when making tests. For example:

```c
if ( function(param1,param2) ) {
    printf("Error !!!\n");
    return 1;
}
```

If “function” returns a non-zero (i.e., “true”) value, it means that something has gone wrong.

This isn't true for all functions in the Standard C Library. Check the documentation if you're not sure about a particular function.

“Did it fail?”
DANGER: Take Care with Floating-Point Numbers
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
Quiz time:

- Let's say $A = (10000000000.0 / 3.0) \times 3.0$
• Quiz time:
  – Lets say $A = (10000000000.0 / 3.0) * 3.0$
  – What is $A$?
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Lets say $A = (10000000000.0 / 3.0) \times 3.0$
  – What is $A$?
  – Well, we all know the calculation should yield 10000000000.0
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Let's say $A = (10000000000.0 / 3.0) * 3.0$
  – What is $A$?
  – Well, we all know the calculation should yield 10000000000.0
  – The computer would evaluate things in this manner:

    ```
    float A = (10000000000.0/3.0)*3.0;
    if (A==10000000000.0 ){
        printf("A=%lf\n",A);
    } else
        printf("A is not 10000000000.0 \n");
    ```
Quiz time:
- Let's say $A = (10000000000.0 / 3.0) * 3.0$
- What is $A$?
- Well, we all know the calculation should yield 10000000000.0
- The computer would evaluate things in this manner:

```c
float A = (10000000000.0/3.0)*3.0;
if (A==10000000000.0 ){
    printf(“A=%lf\n”,A);
} else
    printf(“A is not 10000000000.0 \n”);
```
- What will be printed?
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Lets say \( A = (10000000000.0 / 3.0) \times 3.0 \)
  – What is \( A \)?
  – Well, we all know the calculation should yield 10000000000.0

  – The computer would evaluate things in this manner:

    ```
    float A = (10000000000.0/3.0)*3.0;
    if (A==10000000000.0 ){
        printf("A=%lf\n",A);
    } else
        printf("A is not 10000000000.0 \n");
    ```

  – What will be printed?

    A is not 10000000000.0

    What?
The `==` operator compares two numbers and returns “true” if they are the same. This works fine for integers, but you shouldn’t use it for floating-point numbers. This example shows why:

```c
int main()
{
    double a=12345678.;
    double loga2 = log(a*a);
    double b=sqrt(exp(loga2));
    printf("b=%20.10lf  a=%20.10lf\n",b,a);
    return(0);
}
```

Output:

```
b=12345678.0000000224  a=12345678.0000000000000000
```

Clearly, “b” is not equal to “a” due to the limited precision of the calculations.
Comparing Floating-Point Data with ‘<’ or ‘>’:

Instead of ==, use inequalities to see if the difference between the floating-point numbers is less than some threshold (chosen by you).

```c
int main(){
    const double SMALL=1e-6;
    double a=12345678.;
    double loga2 = log(a*a);
    double b=sqrt(exp(loga2));
    if (fabs(a-b) < SMALL)
        printf("a=b\n");
    else
        printf("a!=b\n");
    return(0);
}
```

\[ b = \sqrt{e^{\ln(a^2)}} = \sqrt{a^2} = a \]

The `fabs()` function returns the absolute value of a floating-point number. It is part of the Standard C Library.

Notice that we've also introduced an “else” statement, after our “if”. We'll talk about that next.
Loops!
Loops

• Loops allow computers to do what computers do best:
  – Execute repetitive boring tasks efficiently and accurately over and over again

• Two types:

  • **Count-controlled Loops:**
    We use these when we **know, beforehand**, how many times we want to repeat a series of tasks.

  • **Condition-controlled Loops:**
    These are used when we **don't know** how many repetitions will be needed, but we know that we want to stop when some well-defined thing happens.
Count-Controlled Loops!
Count-controlled loops: **for** ()

A “for” loop is a **count**-controlled loop. It has the form shown below.

**General form of a “for” loop:**

```c
for ( initialize counter ; test condition ; counter update ) {
    statement block;
}
```

**Example:**

```c
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number %d\n", i);
}
```

**Output:**

```
Loop number 0
Loop number 1
Loop number 2
Loop number 3
Loop number 4
Loop number 5
Loop number 6
Loop number 7
Loop number 8
Loop number 9
```
How a `for()` loop works

The for statement:

```c
for (i = 0; i < 10; i++)
```

Initialize:

```
i = 0;
```

Test:

```
i < 10 ?
```

If never true, no loops

NO

YES

Increment:

```
i++;
```

Print:

```
printf("loop number %d\n", i);
```
for() loop: Important Parts

• Every for() loop has four important parts:
  – counter initialization
  – test condition
  – counter update
  – body of execution

```c
for (initialize counter; test condition; counter update) {
    statement block;
}
```

The for statement is very flexible because:
• Any valid C expression can be used for initialization or update, and
• Any valid condition can be used for the test condition.

Here are some creative uses of the for statement:

```c
for (i=0; i < n; i++) {
    Loop n times from i=0 to i=n-1
}
```

```c
for (i=0; i < m; i+=2) {
    Loop \( \sim m/2 \) times i = 0, 2, 4,...
}
```

```c
for (i=100; i > 0; i--) {
    Loop 100 times, decrementing i
}
```

Compound statements are also allowed. (This may be a little too creative.)

```c
for (i=0, j=0; i<1000; i++, j=exp(i))
```
Example: Good for() loop usage

```
int i;
for (i = 0 ; i < 10 ; i+=2) {
    printf("loop number \%d\n", i);
}
```

Do all iterator math with the loop updater.

```
int i;
for (i = 0 ; i < 20 ; i++) {
    float a = i*0.5;
    printf("counter= \%f\n", a);
}
```

Use integer iterators to avoid rounding errors with floats.

```
const int NLOOPS=10;
int i;
for (i = 0 ; i < NLOOPS ; i++) {
    printf("loop number \%d\n", i);
}
printf("completed \%d loops", NLOOPS);
```

Use constants to define fixed NLOOPS, especially if you need to use the same value throughout your code.
Example: BAD for() loop usage

```
int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number \&d\n", i);
    i = i+1;
}  How many iterations?

float a;
for (a = 0 ; a < 10 ; a+=0.5) {
    printf("counter= \&f\n", a);
}  9.999999 < 10.0
    Do we loop 20 or 21 times?

int i;
for (i = 0 ; i < 10 ; i++) {
    printf("loop number \&d\n", i);
}
printf("completed \&d loops", i);
```

Note 1: It's extremely bad form to operate on the counter variable within the for loop. This leads to confusing code.

Note 2: It's dangerous to use a float as your counter. Rounding errors may cause the loop run an unexpected number of times.

Note 3: It's also bad practice to use a counter variable outside of the loop.

In this case i = 10, a value not used in the loop.
We can use a `break` statement to prematurely exit a loop:

```c
#include <stdio.h>
int main ()
{
    int n;
    for (n=10; n>0; n--) {
        printf("%d, ", n);
        if (n==3) {
            printf("\ncountdown aborted!\n");
            break;
        }
    }
    return 0;
}
```

**Output:**

```
10, 9, 8, 7, 6, 5, 4, 3, 
countdown aborted!
```
for() loops and continue statements

You can use a `continue` statement to skip the rest of the current loop, and go directly to the next iteration:

```c
#include <stdio.h>
int main ()
{
    for (int n=10; n>0; n--)
    {
        if (n==5 || n==6) continue;
        printf("%d, ", n);
    }
    printf("GO!\n");
    return 0;
}
```

Output:

```
10, 9, 8, 7, 4, 3, 2, 1, GO!
```

Try to minimize the use of break/continue in all but the most obvious cases. Program flow that jumps around is more difficult to understand. This can sometimes be avoided with conditional loops.
It is very common to nest loops in programs, by placing one loop inside of another:

```c
const int NUMDAYS = 7;
const int NUMWEEKS = 14;
int day, week;

for (week=0 ; week<NUMWEEKS ; week++)
{
    gotoMovie(); // Done 14 times.
    for (day=0 ; day<NUMDAYS ; day++) {
        eat();     // Done 98 times
        study();
        sleep();
    }
}
```
Condition-Controlled Loops!
Sometimes, you can't tell ahead of time how many times a loop must run. For Example:

- “Do something until a convergence criterion is satisfied.”
- “Do something until the data are exhausted.”

This is where conditional loops are useful.

Conditional loops come in two flavors:

- **Pre-test Loops**: Check at the start of the loop to see if should be executed (again).
  - These loops may possibly **never** be executed.

- **Post-test Loops**: Check at the end of the loop to see if it is executed again.
  - These are **ALWAYS** executed at least once.
Pre-test Conditional Loops: while()

Syntax:

```cpp
while (condition) {
    BLOCK of statements
}
```

Example:

```cpp
cash = get_paid();
while (cash > 0) {
    error = spend_money_on_snacks(cash, 0.75);
    if (!error) cash = cash - 0.75;
    else break;
}
```

- Do we have cash?
- On successful snack acquisition, debit cash.
- We have too little cash to continue. Break loop here.
- Use non-zero status code to flag error on snack purchase.
Post-test Conditional Loops: do/while()

Syntax:
```
do {
    BLOCK of statements
} while (CONDITION);
```

This loop will always be executed at least once.

Here we call two functions with no return values.

Example:
```
do {
    goto_class();
    do_homework();
} while (!semester_over());
```

When the semester_over() function returns a TRUE value we can break the loop.
We’ll pick up from here next time.

See you Thursday in labs!