Physics 2660: Fundamentals of Scientific Computing

Lecture 5
Instructor: Prof. Chris Neu (chris.neu@virginia.edu)
Reminder

• HW03 due Thursday 23 Feb electronically by noon

• My office hours:
  – 3:30-5pm Tuesdays in Room 022-C (our computer lab)

• TA office hours
  – In Room 022-C
    • Mondays 7-9:30pm
    • Tuesdays 4:30-6:30pm
    • Wednesdays 7-9:30pm

• Grading of HW01, HW02 coming!

• Register for piazza! Only 33 out of 70 so far

• You will need the second textbook, Lyons, in about 2 weeks.
Review and Outline

• Last time:
  – How variables’ values are stored on a computer
  – More on formatted I/O
  – Operators
  – Functions

• Today:
  – Conditional structures:
    • if, if/else, if/elseif/else
    • switch/case
  – Loops
    • Count-controlled loops: for()More on loops
    • conditioned controlled loops: while(), do
  – Scope of variables
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:
```
if (CONDITION) {
    BLOCK of statements
}
```

Example:
```
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:
```
if (CONDITION)
    statement;
```

Example:
```
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><code>!</code></td>
<td>Logical NOT. Invert a test or true/false value</td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>2</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>2</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less or equal</td>
<td>2</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater or equal</td>
<td>2</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Equality</td>
<td>3</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
<td>3</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></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.*
Conditional Execution: if and else

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.
Conditional Execution: if and else

Program Flow with “if” Statements:

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

```plaintext
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.

```c
switch (EXPRESSION) {
  case VALUE1:
    BLOCK of statements
    break;
  case VALUE2:
    BLOCK of statements
    break;
  case VALUE3:
    BLOCK of statements
    break;
  default:
    BLOCK of statements
}
```
Conditional Execution: `switch` and `case`

Controlling Flow with “break”:

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

```
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

```c
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.

```c
if (letter=='A'||letter=='a'||letter=='1'){
    printf("A\n");
} else if (letter=='B'||letter=='b'||letter=='2'){
    printf("B\n");
}
```
As a shorter alternative to the regular if statement, C offers the special operator “?:”. This is a ternary operator, meaning that it takes three arguments. The syntax is as follows:

```
CONDITION ? STATEMENT : STATEMENT;
```

Example:

```
(a==b) ? printf("it's true\n") : printf("it's false\n");
```

This is just equivalent to the if statement shown at the right:

```
if (a==b)
    printf("it's true\n");
else
    printf("it's false\n");
```
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”);
```

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”);
```

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”).
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?

```
int a=0;
int b=1;
if (a=b)
    printf("the answer is ");
else
    printf("the answer is ");
```

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

Take-away message:

Be careful to use `==` in conditionals and not `=`.
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 a 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.
DANGER: Take Care with Floating-Point Numbers
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Let's say \( A = \left(10000000000.0 / 3.0\right) \times 3.0 \)
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – Let's say $A = (10000000000.0 / 3.0) \times 3.0$
  – What is $A$?
Quiz time:

– Let's say $A = \left(\frac{10000000000.0}{3.0}\right) \times 3.0$
– What is $A$?
– Well, we all know the calculation should yield 10000000000.0
DANGER: Take Care with Floating-Point Numbers

• Quiz time:
  – 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:

```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");
```
Quiz time:

- Let's 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:

```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:
  – Let's 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:

    ```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?
    A is not 10000000000.0
    ```
Floats and Comparisons

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=12345678.0000000000
```

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);
}
```

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

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!
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:
```
for (i = 0; i < 10; i++)
```

Initialize:
```
i = 0;
```

Test:
```
i < 10?
```

If never true, no loops

NO

YES

Increment:
```
i++;
```

printf("loop number %d\n", i);

Continue...
for() loop: Important Parts

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

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:

```
for (i=0 ; i < n ; i++) {
    // Loop n times from i=0 to i=n-1
}
for (i=0 ; i < m ; i+=2) {
    // Loop ~m/2 times i = 0, 2, 4,...
}
for (i=100 ; i > 0 ; i--) {
    // Loop 100 times, decrementing i
}
```

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

```
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);
}

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

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

Do all iterator math with the loop updater.

Use integer iterators to avoid rounding errors with floats.

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

**Bad for Loop Usage:**

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

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

```c
float a;
for (a = 0; a < 10; a+=0.5) {
    printf("counter= %f
", a);
}  
9.9999999 < 10.0
Do we loop 20 or 21 times?
```

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

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

**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.
for() loops and break statements

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.
Nested 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: \texttt{while()}

\textbf{Syntax:}

\begin{verbatim}
while (condition) {
    BLOCK of statements
}
\end{verbatim}

\textbf{Example:}

\begin{verbatim}
cash = get_paid();
while (cash > 0) {
    error = spend_money_on_snacks(cash, 0.75);
    if (!error) cash = cash - 0.75;
    else break;
}
\end{verbatim}

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

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

This loop will always be executed at least once.

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

Here we call two functions with no return values.

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!