Control Structures

- Both languages support these control structures which function the same way in C and Java
  - **for** loops
  - **while** loops
  - **do-while** loops
  - **switch** statements
  - **if** and **if-else** statements
  - **braces** ({, }) are used to begin and end blocks

- Loop Practice:
  - Blackboard file: Class 9 - C Basics - part 2-Loop Problems.pdf
  - UMBC on-line file: http://www.csee.umbc.edu/courses/undergraduate/201/spring09/misc/loops.shtml

- Good examples of these loops found online at: http://www.tutorialspoint.com/cprogramming/c_loops.htm

Curly Braces {}

- C uses curly braces ({}) to group multiple statements together such as for control blocks and functions. The statements execute in order. Some languages let you declare variables on any line (C++). Other languages insist that variables are declared only at the beginning of functions (Pascal).

- C (C89) takes the middle road --variables may be declared within the body of a function, but they must follow a '{'. More modern languages like Java and C++ allow you to declare variables on any line, which is handy. C99 allows this, as do GNU extensions.

```
/* file: temp.c */
int main(){
  int I = 7;
  if(I == 7){
    int j = 1;
    I = i+j;
    int k;
    k = i *i;
  }
}
```

```
>gcc temp.c

>gcc temp.c -std=c89 -pedantic
temp.c: In function `main':
temp.c:10:3: warning: ISO C90
forbids mixed declarations and
code
```

if - else - else if

- If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.
- C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

```
if (<expression>) <statement>; // simple form with
                               //no {}'s or else clause
```

```
if (<expression>) {          // simple form with {}'s
  <statement>;               // to group statements
  <statement>;
}
```

```
if (<expression>) {          // full then/else form
  <statement>;} else {
  <statement>;
}
```

```
if (<expression1>) {         // as many levels as needed…
  <statement1>;
} else if (<expression2>){
  <statement2>;
} else {
  <statement3>;
}
```

Spacing Variation and Placement of {}'s

```
if(<expression>) {
  <statement>;
}else {
  <statement>;
}
```

```
if (<expression>)
{
  <statement>;
}
else {
  <statement>;
}
```

```
if (<expression>)
{
  <statement>;
}
else
{
  <statement>;
}
```

- In general, spacing is flexible in C
- Pick a convention and follow it...
- Unless your employer specifies a standard, in which case follow that
- If you are the employer, address coding style and consider making a style document for your employees

switch

- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

```
switch (<expression>) {
  case <const-expression-1>:
    <statement>;
    break;

  case <const-expression-2>:
    <statement>;
    break;

  case <const-expression-3>:  //combined case 3 and 4
  case <const-expression-4>:
    <statement>;
    break;

  case <const-expression-5>:  //no break mistake? maybe
    <statement>;

  case <const-expression-6>:
    <statement>;
     break;

  default: // optional
    <statement>;
    break;
```

Omitting the break statements is a common error –it compiles, but leads to inadvertent fall-through behavior. This behavior is just like the assembly jump tables it implements.

while

- A **while** loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.

```
while (<expression>) {
  <statement>;          // or multiple statements…
}
```

do while

- Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.

- A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. (-- not always desired!)

```
do {
  <statement>;          // or multiple statements…
} while (<expression>)
```

for

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

```
for (<initialization>; <continuation>; <action>)
{
  <statement>;
}

for (; <continuation>; <action>) {
  <statement>;
}
```

- <initialization>, <continuation>, and <action> are all optional
- May optionally declare a variable in <initialization>
- <condition> must be satisfied for every execution of <statement>, including the first iteration. In other words <condition> serves as the entry and the repeat condition
- <action> is code performed after the <statement code>

```
int i = 99;
for ( ; i != 0 ; )
{
  <statement>;
  --i ;
}
```

These two are equivalent:

```
for ( i = 99; i != 0 ; i = i - 1)
{
  <statement>;
}
```

break

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- As shown with the case statement, the **break** statement can be used to terminate a case in the **switch** statement.
- If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

```
while (<expression>) {

  <statement>;
  <statement>;

  if (<condition which can only be evaluated here>)
    break;

  <statement>;
  <statement>;
}

// control jumps down here on the break
```

continue

- The **continue** statement in C programming language works somewhat like the **break** statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

- For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do…while** loops, **continue** statement causes the program control passes to the conditional tests. (essentially, it drops to the close bracket of the control loop)

```
while (<expression>) {

  <statement>;
  <statement>;

  if (<condition which can only be evaluated here>)
    continue;

  <statement>;
  <statement>;

// control jumps down here on the continue
//   INSIDE of the END of the loop…
}
```

'?' - Conditional Expression -or- The Ternary Operator

- Syntax: **< Exp1 > ? < Exp2 > : < Exp3 >**
- Where **< Exp1 >** , **< Exp2 >** , and **< Exp3 >** are expressions. Notice the use and placement of the colon.
- The value of a **?** expression is determined like this: **< Exp1 >** is evaluated. If it is true, then **< Exp2 >** is evaluated and becomes the value of the entire **?** expression. If **< Exp1 >** is false, then **< Exp3 >** is evaluated and its value becomes the value of the expression.

```
If ( x < y )  {
   min = x;
} else {
   min = y;
}
```

These two are equivalent:

```
min = (x < y) ? x : y;
```

- use this operator sparingly, since it often makes code less readable, but sometimes it is an obvious fit.

Other Operators

- These other operators are very similar in C and Java
  - **<<, >>, &, |, ^ (bit operators*) (*much more on these later…)**
  - **<<=, >>=, &=, |=,^=**
  - **[ ] (brackets for arrays)**
  - **( ) parenthesis for functions and type casting (discussed later)**

Arrays [ ]

- Like most languages, C supports arrays as a basic data structure.
- Array indexing starts with 0.
- ANSI C requires that the size of the array be a constant (if specified)
- Declaring and initializing arrays

```
int  grades[44];
int  areas[10] = {1, 2, 3};
long  widths[12] = {0};
int  IQs[ ] = {120, 121, 99, 154};
```

Variable Size Arrays

- C99 allows the size of an array to be a variable.

```
int  nrStudents = 30;
     -------
int  grades[nrStudents];
```

2-D Arrays

- Like most languages, C supports multi-dimensional array
- Subscripting is provided for each dimension
- For 2-D arrays, the first dimension is the number of "rows", the second is the number of "columns" in each row

```
int board[ 4 ] [ 5 ];      // 4 rows, 5 columns

int x = board[ 0 ][ 0 ];   // 1st row, 1st column

int y = board[ 3 ][ 4 ];   // 4th(last) row,
                           // 5th(last) column
```

#defines

- The **#define** directive can be used to give names to important constants in your code. This makes your code more readable and more easily changeable.
- The compiler's preprocessor replaces every instance of the **#define** name with the text that it represents.
- Note that there is no terminating semi-colon

```
#define MIN_AGE 21
        ...
  if (myAge > MIN_AGE)
        ...

#define PI 3.1415
        ...
  double area = PI * radius * radius;
```

#define vs const

- **#define**
  - Pro: no memory is used for the constant
  - Con: cannot be seen when code is compiled since they are removed by the pre-compiler
  - Con: are not real variables and have no type

- **const variables**.
  - Pro: are real variables with a type
  - Pro: can be examined by a debugger
  - Con: take up memory

#define vs const - Examples

Based on the discussion from: <http://bytes.com/topic/c/answers/132247-define-versus-const>

- **Example #1**

```
#define   NUMBER1    -42
const int NUMBER2 = -42;

int main()
{
  int x = -NUMBER1;   // x is now 42
  int y = -NUMBER2;   // y is now -42 (?41?-error)
}
```

- **Example #2**

```
#define   NUMBER1    5 + 2
const int NUMBER2 = 5 + 2

int main()
{
  int x = 3 * NUMBER1;   // x is now 17
  int y = 3 * NUMBER2;   // y is now 21
}
```

but if, we said, instead:
define (5 + 2)
the answer would also be 21…
- can you say why?
Parenthesis can be a big help!!!

- **Example #3**

```
#define   NUMBER1 = 5 + 2;

int main()
{
  int x = NUMBER1 * 3;   // Will give a compiler error!!
}                        // Note the semicolon in the define…
```

typedefs

- C allows you to define new names for existing data types (NOT new data types)
- This feature can be used to give application-specific names to simple types

```
typedef int    Temperature;
typedef int[3] Row;
```

- Or to give simple names to complex types
  -more on this later
- Using **typedefs** makes future changes easier and makes the code more relevant to the application

Enumeration Constants

- C provides the enumas a list of named constant integer values (starting a 0 by default)
- Behave like integers
- Names in enummust be distinct
- Often a better alternative to #defines
- Example:

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC };

...

enum months thisMonth;

thisMonth = SEP; // ok
thisMonth = 42;  // unfortunately, also ok
```

Enumeration Constants

- **C functions (no explicit procedures)**
  - **Have a name**
  - **Have a return type (a void return type represents a procedure)**
  - **May have parameters**
- **Before a function may be called, its "prototype" (aka signature --name and parameters) must be known to the compiler so that it can verify that your code is calling the function correctly.**
- **This is accomplished in one of two ways**
  - **Provide the entire function definition prior to the calling code**
  - **Provide the function prototype prior to the calling code and provide the function definition elsewhere**
- **Unlike Java methods, a function in C is uniquely identified by its name. Therefore, there is no concept of method overloading in C as there is in Java. There can be only one main( )function in a C application.**
- **Our standards dictate that function names begin with an UPPERCASE letter**

A Simple C Program

```
#include <stdio.h>
typedef double Radius;
#define PI 3.1415

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius ){
  return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius ){
  return (2 * PI * radius );
}

int main( ){
  Radius radius = 4.5;
  double area = circleArea( radius );
  double circumference = Circumference( radius );

  // print the results
  printf("\n radius = %f, ",area);
  printf("circumference = %f\n\n",circumference);

  return 0;
}
```

Alternate Sample with prototypes (declaration before call, definition after call)

```
#include <stdio.h>
typedef double Radius;
#define PI 3.1415

/* function prototypes */
double CircleArea( Radius radius );
double Circumference( Radius radius );

int main( ){
  Radius radius = 4.5;
  double area = circleArea( radius );
  double circumference = Circumference( radius );

  // print the results
  printf("\n radius = %f, ",area);
  printf("circumference = %f\n\n",circumference);

return 0;
}

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius ){
return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius ){
return (2 * PI * radius );
}
```

Typical C Program

includes

Defines, typedefs, data type definitions, global variable declarations, function prototypes

`main()`

Function definitions

```
#include <stdio.h>

typedef double Radius;
#define PI 3.1415

/* function prototypes */
double CircleArea( Radius radius );
double Circumference( Radius radius );

int main( )
{
   Radius radius = 4.5;
   double area = circleArea( radius );
   double circumference = Circumference( radius );

   // print the results
   return 0;
}

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius )
{
   return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius )
{
   return (2 * PI * radius );
}
```