Expressions & Conditionals

Operators

- C has many built-in operators
- We discussed the basic operators in the previous unit, but here will review and expand on those operators
- Basic operators include:
  - + addition
  - − subtraction
  - * multiplication
  - / division (floating point and integer division depending upon type)
  - % modulo (remainder division)

Increment and decrement

- Adding 1 to a variable is so common, we have a custom operator for it
- We have three methods of incrementing (the same is true for decrement) variables in C
- Using addition (not technically an increment operator):

```
a = a + 1; //
```

- Using the prefix increment operator

```
++a;
```

- Using the postfix increment operator

```
a++;
```

- What's the difference between these two statements?
  - A lot of the time, there is no difference in practice
  - **HOWEVER**
  - The semantics are technically different
  - How are they different?
    * `++a` increments `a` and then returns the value of `a` (meaning the value returned has already been incremented)
    * `a++` returns the value of `a` and then increments `a` (meaning the value returned has *not* yet been incremented)
  - When does this matter?

        * Two cases:

```c
int a = 5;
int b = 5;
int c = ++a; // c has the value of 6 after this executes, a has the
   value of 6
int d = a++; // d has the value of 5 after this executes, a has the
   value of 6
```

- All of the above also applies for the decrement operator `--`, which subtracts 1 from a variable instead of adding one

Relational and Equality Operators

- Relational and equality operators return boolean values (boolean meaning true or false)
- Relational operators
  - < less than
  - > greater than
  - <= less than or equal to
  - >= greater than or equal to
- Equality operators
  - == equals
  - != not equals

Unary Operators

- Unary means it only acts on a single variable (whereas so far other operators have acted on two variables/values)
- Unary operators
  - & address-of (gets the memory address of a variable)
  - * contents-of (gets the contents stored in a memory address)
  - − negation
  - + plus
  - ! logical negation
  - `(type)` type casting
    * *Type casting* allows us to convert one variable type into another type
    * Notice that this does **not** round, it truncates
    * For instance, common use is to typecase a floating point number into an integer:

```c
int b = (int)3.5; // 3.5 will be truncated to 3
```

Logical Operators

- `!` logical negation
- `&&` logical AND
- `||` logical OR

Precedence

- 1 (highest)
  - `++`/`--`Postfix increment and decrement
  - `()`Function calls
  - `[]` Array subscripting
  - `.` structure/union access
  - `->` structure/union member access through pointer
- 2 (second highest)
  - `++`/`--` Prefix increment and decrement
  - `+`/`-` unary plus and minus
  - `!` logical not
  - `(type)` type casting
  - `*` dereference
  - `&` address-of
- 3 (third highest)
  - `*` multiplication
  - `/` division
  - `%` remainder division
- 4 (forth highest)
  - `+` addition
  - `-` subtraction
- 5 (fifth highest)
  - `<` less than
  - `>` greater than
  - `<=` less than or equal to
  - `>=` greater than or equal to
- 6 (sixth highest)
  - `==` equal
  - `!=` not equal
- 7 (seventh highest)
  - `&&` logical AND

- 8 (eighth highest)
  - || logical OR
- 9 (ninth highest)
  - ?: ternary conditional
- 10 (tenth highest)
  - = assignment (and all other forms of assignment, +=, -=, *=, /=, %=)
- 11 (eleventh highest)
  - , comma (for creating multiple variables)

calculations.c

```c
/*
   Let's try writing some calculations...
 */
#include <stdio.h>

int main() {
   float value1 = 0.0, value2 = 0.0;
   int i1 = 0, i2 = 0;

   /*
      Prompt for values
    -/
   printf( "\t\tCalculation Program\n\n" );
   printf( "Please enter two values: " );
   scanf( "%f %f", &value1, &value2 );

   printf( "Their sum: %f\n", value1 + value2 );
   printf( "Their product: %f\n", value1 * value2 );
   printf( "The first minus the second: %f\n", value1 - value2 );
   printf( "The second minus the first: %f\n",  value2 - value1 );
   printf( "The first divided by the second: %f\n", value1 / value2 );
   printf( "The second divided by the first: %f\n", value2 / value1 );

   printf( "Let's try again, as if the values were int\n" );
   i1 = (int) value1;
   i2 = (int) value2;
   printf( "Their sum: %d\n", i1 + i2 );
   printf( "Their product: %d\n", i1 * i2 );
   printf( "The first minus the second: %d\n", i1 - i2 );
   printf( "The second minus the first: %d\n", i2 - i1 );
```

```
    printf( "The first divided by the second: %d\n", i1 / i2 );
    /*
       Just for the record, the modulus operator is only
       defined on int parameters...
     -/
    printf( "The modulus of the first by the second: %d\n", i1 % i2 );
    printf( "The second divided by the first: %d\n", i2 / i1 );
    printf( "The modulus of the second by the first: %d\n", i2 % i1 );
    return( 0 );
}
```

Conditionals

- There is no meaningful program that doesn't demonstrate some basic decision-making skills
- For instance, "I will continue driving through the intersection" is not a statement a human would act upon. But "I will stop if the light is red, go if the light is green, go only if I can safely pass if the light is yellow" might be a reasonable driving policy.
- A conditional tells the computer to only execute a block of code if a particular condition has been satisfied (i.e. that condition is true).
- `if...else` is the most common conditional statement
- `switch...case` are used as a shorthand version of `if...else`
- In C, logic is a form of arithmetic
  - `0` represents false
  - **any** other value represents true
  - Logical and arithmetic operators are treated as the same thing in C

selection.c

```
/*
   Let's try writing some conditional logic...
 -/
#include <stdio.h>

int main() {
    float value1 = 0.0, value2 = 0.0;
    /*
       Prompt for values
     -/
    printf( "\t\tConditional Logic Program\n\n" );
    printf( "Please enter two values: " );
    scanf( "%f %f", &value1, &value2 );

    if (value1 < value2) {
        printf( "value1 is less than value2\n" );
    }
    if (value1 > value2) {
        printf( "value1 is greater than value2\n" );
    }
    if (value1 <= value2) {
        printf( "value1 is less than or equal value2\n" );
    }
    if (value1 >= value2) {
        printf( "value1 is greater than or equal value2\n" );
    }
    if (value1 == value2) {
        printf( "value1 equals value2\n" );
    }
    if (value1 != value2) {
        printf( "value1 does not equals value2\n" );
    }
    /*
       Just for the record, due to rounding errors, it
       is very dangerous to test for equality on floating
       point numbers
     */
    return( 0 );
}
```

Relational Expressions

- Consider the following relational and equivalence operations

```
a < b     // 1 if a is less than b, 0 otherwise
a > b     // 1 if a is greater than b, 0 otherwise
a <= b    // 1 if a is less than or equal to b, 0 otherwise
a >= b    // 1 if a is greater than or equal to b, 0 otherwise
a == b    // 1 if a is equal to b, 0 otherwise
a != b    // 1 if a is not equal to b, 0 otherwise
```

- Please remember = is for **assignment** and == is for **equality**
- C does not have a dedicated boolean type that many programming languages have. Instead, integers represent booleans.

    - 0 means false
    - anything else means true
- Our two examples of conditionals are equivalent:

```
if (foo()) {
  // do something
}
if (foo() != 0) {
  // do something
}
```

Logical Expressions

- A way to evaluate operations over logical values (i.e. `0` for false and anything else for true)
- Gives a way to encode "this AND that" or "this OR that"
- Consider the following:

```
a || b     // 1 when EITHER a OR b is true, 0 otherwise
a && b     // 1 when BOTH a AND b are true, 0 otherwise
!a         // 1 when a is false, 0 otherwise
```

- We can string multiple logical expressions together to create compound logical expressions:

```
((a && b) || (c > d))
```

If-Else Statements

- Executes a block of code if particular conditions have been met
- Basic syntax:

```
if (/* condition goes here */) {
  /* if the condition is non-zero (true), this code will execute */
} else {
  /* if the condition is 0 (false), this code will execute */
}
```

- The first block executes if the condition is true, otherwise the second block executes
- The **`else`** is completely optional
- An **`if`** can directly follow an else, creating a chain of conditions to check:

```c
if (a > b) {
  c = a;
} else if (b > a) {
  c = b;
} else {
  c = 0;
}
```

- This code sets the variable `c` equal to the greater of the two variables `a` or `b`, or `0` if `a` and `b` are equal.
- What's the point of the `else` when you can just do this:

```c
if (a > b) {
  c = a;
}

if (a < b) {
  c = b;
}

if (a == b) {
  c = 0;
}
```

1. Multiple `if`'s could be true, there is no guaranteed mutual exclusion
2. Evaluating `if` statements takes time (since the condition must be checked).

- If you only have single statement to statement to execute in your `if` or `else`, you do not need to put a block.
  - **But you should.**
  - Consider the following

```c
if(5 < 10)
  printf("I am inside the if\n");
else
  printf("I am inside the else\n");
  printf("5 is not less than 10\n");
```

- Only the single statement following the `if` or `else` is associated with the conditional…so when will `5 is not less than 10` be printed?
  - **Every time**
  - Indentation cannot be trusted!

    - Here's a version with indentation reflecting the semantics of the program:

```c
if(5 < 10)
  printf("I am inside the if\n");
else
  printf("I am inside the else\n");
printf("5 is not less than 10\n");
```

- How do you fix this?
    - **Always use block statements!**

```c
if(5 < 10) {
  printf("I am inside the if\n");
} else {
  printf("I am inside the else\n");
  printf("5 is not less than 10\n");
}
```

Nesting

If statements can be nested, meaning you have `if` statements inside of `if` statements

nesting.c

```c
/*
   Let's try writing some nested conditional statements...
 -/
#include <stdio.h>

int main() {
   int temperature;
   /*
      Prompt for values
    -/
   printf( "\t\tNested Logic Program\n\n" );
   printf( "Please enter today's temperature: " );
   scanf( "%d", &temperature );

   if (temperature < 50) {
       printf( "Gosh, it feels cold...\n" );
       if (temperature < 32) {
            printf( "And it looks like it's freezing...\n" );
        }
        else if (temperature < 40) {
            printf( "And it's nearly freezing...\n" );
        }
        else {
            printf( "But atleast it's not freezing cold!\n" );
        }
    }
    else if (temperature > 90) {
        printf( "Gosh, it's hot...\n" );
        if (temperature > 110) {
            printf( "And it's just boiling... head for air conditioning
                ...\n" );
        }
        else if (temperature > 100) {
            printf( "Atleast it's not boiling...\n" );
        }
        else {
            printf( "What a heat wave!!\n" );
        }
    }
    else {
        printf( "Doesn't California have a nice climate!\n" );
    }
    return( 0 );
}
```

- The conditional expression is an if statement that can be assigned to a variable. It is commonly called the *ternary operator*
  - The syntax is the following:

```c
(/* logical expression goes here */) ? (/* if non-zero (true) */) : (/*
    if 0 (false) */)
```

- If the logical expression is true, the overall condition evaluates to the expression between the `?` and the `:`.
- If the logical expression is false, the overall condition evaluates to the expression after the `:`
- For example, if we want to set `c` to be the larger value of two variables `a` and `b`, we could write the following:

```c
c = (a > b) ? a : b;
```

Switch-Case Statement

- It's generally bad practice to string together more than two or three `if..else` statements together. The `switch...case` statement enables us to write many "cases" that could be handled by `if...else` in a cleaner manner
- Basic syntax:

```c
switch (/* integer or enum goes here */) {
case /* potential value of the aforementioned int or enum */:
  /* code */
case /* a different potential value */:
  /* different code */
/* insert additional cases as needed */
default:
  /* more code */
}
```

- The switch uses a variable, and integer or enum, to control which case to evaluate. This is a limiation; if you must compare more complicated data, you cannot use a `switch...case`
- This variable is compared against each `case`, one the comparison is true, that particular case will activate (execute)
  - Once a `case` has been activated, no other cases will be evaluated
- Typically, the last statement for each case is a `break` statement. The causes the program to jump to the statement following the closing } of the switch statement.
  - This basically ends the switch statement (and this behavior is probably your intuition behind each case
  - However, if you omit the `break`, the cases "fall throw" until the end of the switch or until a `break` is reached
- If no cases are matched and a `default` case is specified, the default case will execute.
  - Use of `default` is optional.

multiselect.c

```c
/*
   Let's try writing a switch statement...
 */
#include <stdio.h>

int main() {
   char letter;
   /*
```

```c
       Prompt for values
     */
    printf( "\t\tCase Statement Program\n\n" );
    printf( "Please enter a letter to inspect: " );
    scanf( "%c", &letter );

    /*
       Just for the record, you can only switch on a
       integral value.  The char datatype is just another
       name for the set of ints between 0 and 255, so you
       can switch on chars or ints
     */
    switch( letter ) {
      /*
         Individual letters must be single-quoted.
         Individual letters map directly to constant
         integer values based on the ASCII table which
         we will learn about in upcoming units.  The
         value of each case must be a constant value,
         not an expression or variable.  This often
         makes switch statements not applicable to your
         situation.
       */
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
      case 'y':
          /*
             Lacking break statements in the upper
             listed cases, they will all "fall thru"
             to the set of statements shown here.
             While at first this may seem very convenient,
             this is actually the number one programming
             bug worldwide.  Namely, that folks forget that
             all the above cases are collapsing down to
             the code shown below.  So use this form
             with great caution, as it often leads to
             bugs...
           */
          printf( "a nice lowercase vowel!\n" );
          break;
```

```c
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
        case 'Y':
            printf( "a nice UPPERCASE vowel!\n" );
            break;
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            printf( "a nice number!\n" );
            break;
        default:
            /*
               The default case is the one selected when
               no other cases actually match the switched
               data
             */
            printf( "this is not something I recognize...\n" );
            break;
    }
    return( 0 );
}
```

Exercises

Write a C program to check whether a given number is even or odd

```c
#include <stdio.h>
void main()
{
  int num1, rem1;

  printf("Input an integer : ");
  scanf("%d", &num1);
  rem1 = num1 % 2;
  if (rem1 == 0)
    printf("%d is an even integer\n", num1);
  else
    printf("%d is an odd integer\n", num1);
}
```

Write a C program to find whether a given year is a leap year or not.

```c
#include <stdio.h>
void main()
{
    int chk_year;

    printf("Input a year :");
    scanf("%d", &chk_year);
    if ((chk_year % 400) == 0)
        printf("%d is a leap year.\n", chk_year);
    else if ((chk_year % 100) == 0)
        printf("%d is a not leap year.\n", chk_year);
    else if ((chk_year % 4) == 0)
        printf("%d is a leap year.\n", chk_year);
    else
        printf("%d is not a leap year \n", chk_year);
}
```

Write a C program to read any day (7 days of the week) in integer form (as a number) and display day name using the corresponding word 1 - Monday 2 - Tuesday 3 - Wednesday … 6 - Saturday 7 - Sunday

```c
#include <stdio.h>
void main()
{
  int dayno;
  printf("Input Day No : ");
  scanf("%d",&dayno);
  switch(dayno)
  {
    case 1:
       printf("Monday \n");
       break;
    case 2:
       printf("Tuesday \n");
       break;
    case 3:
       printf("Wednesday \n");
       break;
    case 4:
       printf("Thursday \n");
       break;
    case 5:
       printf("Friday \n");
       break;
    case 6:
       printf("Saturday \n");
       break;
    case 7:
       printf("Sunday  \n");
       break;
    default:
       printf("Invalid day number. \nPlease try again ....\n");
       break;
  }
}
```