C Program Control

CS 2060 Week 3

Prof. Jonathan Ventura

Counter-controlled iteration with `while` loops

- Counter-controlled iteration requires:
  1. The **name** of a control variable (or loop counter)
  2. The **initial value** of the control variable.
  3. The **increment** or **decrement** by which the control variable is modified each time through the loop.
  4. The condition that tests for the **final value** of the control variable (i.e., whether looping should continue).

Counter-controlled iteration with while loops

```c
#include <stdio.h>

int main()
{
    unsigned int counter = 1; // intialization

    while ( counter <= 10 ) // iteration condition
    {
        printf( "%u\n", counter );
        ++counter; // increment
    }
}
```

Counter-controlled iteration with while loops

```c
#include <stdio.h>

int main()
{
    unsigned int counter = 0; // intialization

    while ( ++counter <= 10 ) // increment and iteration condition
    {
        printf( "%u\n", counter );
    }
}
```

Counter-controlled iteration with while loops

```c
#include <stdio.h>

int main()
{
    unsigned int counter = 0; // intialization

    while ( counter++ < 10 ) // increment and iteration condition
    {
        printf( "%u\n", counter );
    }
}
```

`unsigned` types

- Notes on `unsigned` types:
  - `unsigned int` denotes a variable that represents non-negative integers (0 to $2^{32} - 1$).
  - Be careful when using `unsigned` variables in arithmetic.
  - Output `unsigned int` using the format specifier %u.

for loops

- A for loop can be used to compactly produce counter-controlled iteration.
- The syntax of a for loop is as follows:

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

Counter-controlled iteration with for loops

```
for ( unsigned int counter = 1; counter <= 10; ++counter )
{
    printf( "%u\n", counter );
}

// ---- is equivalent to: ----

unsigned int counter = 1;
while ( counter <= 10 )
{
    printf( "%u\n", counter );
    ++counter;
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    for ( unsigned int counter = 1; counter <= 10; counter++ )
    {
        printf( "%u\n", counter );
    }
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    for ( unsigned int counter = 0; counter < 10; counter++ )
    {
        printf( "%u\n", counter+1 );
    }
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    unsigned int counter;
    for ( counter = 0; counter < 10; counter++ )
    {
        printf( "%u\n", counter+1 );
    }
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    unsigned int counter = 0;
    for ( ; counter < 10; counter++ )
    {
        printf( "%u\n", counter+1 );
    }
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    for ( unsigned int counter = 0; counter <= 100; counter += 10 )
    {
        printf( "%u\n", counter );
    }
}
```

Counter-controlled iteration with for loops

```c
#include <stdio.h>

int main()
{
    // infinite loop! need to use int, not unsigned int
    for ( unsigned int counter = 100; counter >= 0; --counter )
    {
        printf( "%u\n", counter );
    }
}
```

do...while loops

- The `do...while` construction is slightly different than `while`.
- The condition is tested *after* each iteration.
  - This means the loop is guaranteed to run at least once.

`do { statements } while ( condition );`

- This can help to reduce redundant code in some situations.

do...while loops

```
int total = 0;
int input = 0;
scanf( "%d", &input );
while( input >= 0 )
{
    total += input;
    scanf( "%d", &input ); // repeated scanf() call
}
```

do...while loops

```
int total = 0;
int input = 0;
do
{
    scanf( "%d", &input );
    if ( input < 0 ) break;
    total += input;
} while ( true );
```

Infinite loops

```c
#include <stdio.h>

int main()
{
    while ( true ) printf( "infinite loop!\n" );

    for ( ; ; ; ) printf( "infinite loop!\n" );

    do printf( "infinite loop!\n" ); while ( true );
}
```

`bool` type in C

- C does have a boolean type: `_Bool`
  - `_Bool` can only be 1 or 0
- The standard library header `stdbool.h` defines a type `bool` which is equivalent to `_Bool`
- `stdbool.h` also defines keywords `true` and `false` which evaluate to 1 and 0

break and continue

- break is used to immediately exit a loop structure.
- continue is used to stop the current iteration and continue on to the next one.

break statement

```
int total = 0;
int input = 0;
do
{
    scanf( "%d", &input );
    if ( input < 0 ) break;
    total += input;
} while ( true );
```

continue statement

```
float sum = 0;
int count = 0;
for ( int i = 0; i < 10; i++ )
{
    float input;
    scanf( "%f", &input );

    if ( input > 2. ) continue;

    sum += input;
    count++;
}
printf( "average: %f\n", sum / count );
```

ASCII Characters in C

- When programming, letters are actually represented by numbers.

ASCII Characters in C

- When programming, letters are actually represented by numbers.
- Typically the **ASCII (American Standard Code for Information Interchange) character set** is used (see Appendix B of the book).

ASCII Characters in C

- When programming, letters are actually represented by numbers.
- Typically the **ASCII (American Standard Code for Information Interchange) character set** is used (see Appendix B of the book).
- For example, a = 97, b = 98, c = 99, ...

ASCII Characters in C

- When programming, letters are actually represented by numbers.
- Typically the **ASCII (American Standard Code for Information Interchange) character set** is used (see Appendix B of the book).
- For example, a = 97, b = 98, c = 99, ...
- Unicode is a more modern version offering 16-bit and 32-bit character sets with many more characters from other alphabets / languages.

ASCII Characters in C

- When programming, letters are actually represented by numbers.
- Typically the **ASCII (American Standard Code for Information Interchange) character set** is used (see Appendix B of the book).
- For example, a = 97, b = 98, c = 99, ...
- Unicode is a more modern version offering 16-bit and 32-bit character sets with many more characters from other alphabets / languages.
- UTF-8 is 8-bit Unicode which is backward-compatible with ASCII.

ASCII Characters in C

- C provides a `char` type which can be used to represent characters.

ASCII Characters in C

- C provides a `char` type which can be used to represent characters.
- `char` is an 8-bit integer type.

ASCII Characters in C

- C provides a `char` type which can be used to represent characters.
- `char` is an 8-bit integer type.
- However, the C standard does not define whether `char` is signed (-128 to 127) or unsigned (0 to 255).

ASCII Characters in C

- C provides a `char` type which can be used to represent characters.
- `char` is an 8-bit integer type.
- However, the C standard does not define whether `char` is signed (-128 to 127) or unsigned (0 to 255).
- If you need to be sure you can use `signed char` or `unsigned char`.

ASCII Characters in C

- C provides a `char` type which can be used to represent characters.
- `char` is an 8-bit integer type.
- However, the C standard does not define whether `char` is signed (-128 to 127) or unsigned (0 to 255).
- If you need to be sure you can use `signed char` or `unsigned char`.
- ASCII only uses 0 to 127 so it does not matter for representing characters.

ASCII Characters in C

```c
#include <stdio.h>

int main()
{
    char letter = '*';
    char space = ' ';
    for ( int i = 0; i < 10; i++ )
    {
        printf("%c%c%c%c\n",
            letter,space,letter,space);
        printf("%c%c%c%c\n",
            space,letter,space,letter);
    }
}
```

- Use single quotes around a character to get its ASCII representation.
- Use %c to read/write a character using scanf/printf.

Reading characters with getchar

```c
#include <stdio.h>

int main()
{
    char letter = getchar();
    char space = getchar();
    for ( int i = 0; i < 10; i++ )
    {
        printf("%c%c%c%c\n",
            letter,space,letter,space);
        printf("%c%c%c%c\n",
            space,letter,space,letter);
    }
}
```

- Alternatively, `getchar()` can be used to read a single character.
- You may need to press Enter after entering characters in the console.

End-of-file marker: EOF

```c
#include <stdio.h>

int main()
{
    char input = getchar();
    while ( input != EOF )
    {
        printf("%c",input);
        input = getchar();
    }
}
```

- The `EOF` marker indicates "end of file," meaning the input has ended.
- You can enter EOF in the console using Control-D on Mac/Linux or Control-Z on Windows.

Multiple selection using switch

```
unsigned int num_red = 0, num_blue = 0;
char input;
while ( ( input = getchar() ) != EOF ) {
    switch ( input ) {
        case 'r':
            num_red++;
            break;

        case 'b':
            num_blue++;
            break;
    }
}
printf( "%u red and %u blue\n", num_red, num_blue );
```

- The switch statement is used to select among many options.
- The value of a control variable determines which option is selected.

Multiple selection using switch

```
unsigned int num_red = 0, num_blue = 0;
char input;
while ( ( input = getchar() ) != EOF ) {
    switch ( input ) {
        case 'r':
            num_red++;
            break;

        case 'b':
            num_blue++;
            break;
    }
}
printf( "%u red and %u blue\n", num_red, num_blue );
```

- Uniquely, cases are not delineated with curly braces inside a switch statement.
- Instead, you must use break to indicate the end of a case.

Multiple selection using switch

```
unsigned int num_red = 0, num_blue = 0;
char input;
while ( ( input = getchar() ) != EOF ) {
    switch ( input ) {
        case 'r':
        case 'R':
            num_red++;
            break;

        case 'b':
        case 'B':
            num_blue++;
            break;
    }
}
printf( "%u red and %u blue\n", num_red, num_blue );
```

- If the break is omitted, control will flow on to the next case.
- This makes it easy to assign the same code to multiple cases.

Multiple selection using switch

```
unsigned int num_red = 0, num_blue = 0, num_other = 0;
char input;
while ( ( input = getchar() ) != EOF ) {
    switch ( input ) {
        case 'r':
        case 'R':
            num_red++;
            break;

        case 'b':
        case 'B':
            num_blue++;
            break;

        default:
            num_other++;
    }
}
```

- The default case will be used if no other case label matches.
- A final break statement is not needed.

Boolean flags

- A Boolean variable (`bool`) can be used as a "flag" which tracks the state of some condition.
- Don't forget to include `stdbool.h` to use `bool` and `true` and `false`.

Boolean flags

```
unsigned int num_red = 0, num_blue = 0;
bool blue_state = true;
char input;
while ( ( input = getchar() ) != EOF ) {
    switch ( input ) {
        case 'r':
        case 'R':
            if ( blue_state ) printf("switching to red!\n");
            blue_state = false;
            num_red++;
            break;

        case 'b':
        case 'B':
            if ( !blue_state ) printf("switching to blue!\n");
            blue_state = true;
            num_blue++;
            break;
    }
}
printf( "%u red, %u blue\n", num_red, num_blue );
```

- Here the Boolean tracks whether we are currently reading blue or red.

Logical operators

- Logical operators are used to form Boolean expressions.
  - `&&` **(logical AND)**
  - `||` **(logical OR)**
  - `!` **(logical NOT)**
- Their functions are as follows:
  - `(a && b)` is true only if *both* `a` and `b` are true.
  - `(a || b)` is true if *either* `a` or `b` is true.
  - `(!a)` is true if `a` is false.

Logical operators

```c
#include <stdbool.h>

int main()
{
    bool a = true;
    bool b = true;
    bool c = false;
    bool d = true;

    if ( a && b )
        if ( c || d )
            printf("a and b and (c or d)");
}
```

Logical operators

```c
#include <stdbool.h>

int main()
{
    bool a = true;
    bool b = true;
    bool c = false;
    bool d = true;

    if ( ( a && b ) && ( c || d ) )
        printf("a and b and (c or d)");
}
```

Logical operators

```
char input = getchar();
if ( !( input == EOF ) )
{
    if ( input >= 97 && input <= 122 ) printf( "lowercase\n" );
    else if ( input >= 65 && input <= 90 ) printf( "uppercase\n" );
    else if ( input >= 48 && input <= 57 ) printf( "number\n" );
}
```

Short-circuit evaluation

- `&&` has higher precedence than `||`.
- This allows us to use **short-circuit evaluation** to terminate evaluation of the expression early.

  Example: `(gender == 1) && (age >= 18 || age <= 24)`
- If

  `(gender == 1)`

  is false, then

  `(age >= 18 || age <= 24)`

  does not need to be tested.

Short-circuit evaluation

```
int input;
if ( ( scanf("%d",&input) == 1 ) && input > 0 )
    printf("positive number\n");
```

- We can use short-circuit evaluation in seemingly dangerous ways.
- Here we rely on short-circuit evaluation to avoid testing bad input.

Next Week

- Reading: Chapter 5: C Functions
- Homework 3 due Monday 11:55 PM.
- Quiz Thursday.