Lecture 4

Control Structure II

CPSC 275
Introduction to Computer Systems

Iteration Statements

- C's iteration statements are used to set up loops.
- A ***loop*** is a statement whose job is to repeatedly execute some other statement (the ***loop body***).
- In C, every loop has a ***controlling expression***.
- Each time the loop body is executed (an ***iteration*** of the loop), the controlling expression is evaluated.
  - If the expression is true (has a value that's not zero) the loop continues to execute.

Iteration Statements

- C provides three iteration statements:
  - The `while` statement is used for loops whose controlling expression is tested *before* the loop body is executed.
  - The `do` statement is used if the expression is tested *after* the loop body is executed.
  - The `for` statement is convenient for loops that increment or decrement a counting variable.

The **`while`** Statement

- Using a `while` statement is the easiest way to set up a loop.
- The `while` statement has the form

  `while (` *expression* `)` *statement*
- *expression* is the controlling expression; *statement* is the loop body.

The **`while`** Statement

- Example of a `while` statement:

```
i = 1;
while (i < n)  /* controlling expression */
  i = i * 2;   /* loop body */
```

- When a `while` statement is executed, the controlling expression is evaluated first.
- If its value is nonzero (true), the loop body is executed and the expression is tested again.
- The process continues until the controlling expression eventually has the value zero.
- What does the example loop compute?

The **`while`** Statement

- If multiple statements are needed, use braces to create a single compound statement:

```
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- Some programmers always use braces, even when they're not strictly necessary:

```
while (i < n) {
  i = i * 2;
}
```

The `while` Statement

- The following statements display a series of "countdown" messages:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```

- The final message printed is `T minus 1 and counting`.

7

The `while` Statement

- Observations about the `while` statement:
  - The controlling expression is false when a `while` loop terminates. Thus, when a loop controlled by `i > 0` terminates, `i` must be less than or equal to 0.
  - The body of a `while` loop may not be executed at all, because the controlling expression is tested *before* the body is executed.
  - A `while` statement can often be written in a variety of ways. A more concise version of the countdown loop:

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

8

Infinite Loops

- A `while` statement won't terminate if the controlling expression always has a nonzero value.
- C programmers sometimes deliberately create an ***infinite loop*** by using a nonzero constant as the controlling expression:

```
while (1) …
```

- A `while` statement of this form will execute forever unless its body contains a statement that transfers control out of the loop (`break`, `goto`, `return`) or calls a function that causes the program to terminate.

9

The `do` Statement

- General form of the `do` statement:

  `do` *statement* `while (` *expression* `) ;`

- When a `do` statement is executed, the loop body is executed first, then the controlling expression is evaluated.
- If the value of the expression is nonzero, the loop body is executed again and then the expression is evaluated once more.

10

The `do` Statement

- The countdown example rewritten as a `do` statement:

```
i = 10;
do {
  printf("T minus %d and counting\n", i);
  --i;
} while (i > 0);
```

- The `do` statement is often indistinguishable from the `while` statement.
- The only difference is that the body of a `do` statement is always executed at least once.

11

The `for` Statement

- The `for` statement is ideal for loops that have a "counting" variable, but it's versatile enough to be used for other kinds of loops as well.
- General form of the `for` statement:

  `for (` *expr1* `;` *expr2* `;` *expr3* `)` *statement*

  expr1, *expr2*, and *expr3* are expressions.

- Example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

12

The `for` Statement

- The `for` statement is closely related to the `while` statement.
- A `for` loop can almost always be replaced by an equivalent `while` loop:

```
expr1;
while ( expr2 ) {
  statement
  expr3;
}
```

- *expr1* is an initialization step that's performed only once, before the loop begins to execute.

The `for` Statement

- *expr2* controls loop termination (the loop continues executing as long as the value of *expr2* is nonzero).
- *expr3* is an operation to be performed at the end of each loop iteration.
- The result when this pattern is applied to the previous `for` loop:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  i--;
}
```


The `for` Statement

- Studying the equivalent `while` statement can help clarify the fine points of a `for` statement.
- For example, what if `i--` is replaced by `--i`?

```
for (i = 10; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

- The equivalent `while` loop shows that the change has no effect on the behavior of the loop:

```
i = 10;
while (i > 0) {
  printf("T minus %d and counting\n", i);
  --i;
}
```


Omitting Expressions in a `for` Statement

- C allows any or all of the expressions that control a `for` statement to be omitted.
- If the *first* expression is omitted, no initialization is performed before the loop is executed:

```
i = 10;
for (; i > 0; --i)
  printf("T minus %d and counting\n", i);
```

- If the *third* expression is omitted, the loop body is responsible for ensuring that the value of the second expression eventually becomes false:

```
for (i = 10; i > 0;)
  printf("T minus %d and counting\n", i--);
```


Omitting Expressions in a `for` Statement

- When the *first* and *third* expressions are both omitted, the resulting loop is nothing more than a while statement in disguise:

```
for (; i > 0;)
  printf("T minus %d and counting\n", i--);
```

  is the same as

```
while (i > 0)
  printf("T minus %d and counting\n", i--);
```

- The `while` version is clearer and therefore preferable.

Omitting Expressions in a `for` Statement

- If the *second* expression is missing, it defaults to a true value, so the `for` statement doesn't terminate (unless stopped in some other fashion).
- For example, some programmers use the following `for` statement to establish an infinite loop:

```
for (;;) …
```

`for` Statements in C99

- In C99, the first expression in a `for` statement can be replaced by a declaration.
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
  …
```

- The variable `i` need not have been declared prior to this statement.

`for` Statements in C99

- A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
  …
  printf("%d", i);
  /* legal; i is visible inside loop */
  …
}
printf("%d", i);    /*** WRONG ***/
```


`for` Statements in C99

- Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand.
- However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.
- A `for` statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
  …
```


The Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
- This effect can be accomplished by using a ***comma expression*** as the first or third expression in the `for` statement.
- A comma expression has the form

  expr1 , *expr2*

  where *expr1* and *expr2* are any two expressions.

The Comma Operator

- The comma operator makes it possible to "glue" two expressions together to form a single expression.
- Example:

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

- With additional commas, the `for` statement could initialize more than two variables.

Exiting from a Loop

- The normal exit point for a loop is at the beginning (as in a `while` or `for` statement) or at the end (the `do` statement).
- Using the `break` statement, it's possible to write a loop with an exit point in the middle or a loop with more than one exit point.

The **break** Statement

- The `break` statement can transfer control out of a switch statement, but it can also be used to jump out of a `while`, `do`, or `for` loop.
- Example:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

This loop checks whether a number `n` is prime by using a `break` statement to terminate the loop as soon as a divisor is found.

The **break** Statement

- A `break` statement transfers control out of the <u>innermost</u> enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape <u>only one level</u> of nesting.
- Example:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop.

The **continue** Statement

- The `continue` statement is similar to `break`:
  - `break` transfers control just past the end of a loop.
  - `continue` transfers control to a point just before the end of the loop body.
- With `break`, control leaves the loop; with `continue`, control remains inside the loop.
- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

The **continue** Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```


The **continue** Statement

- The same loop written without using `continue`:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i != 0) {
    sum += i;
    n++;
  }
}
```


The **goto** Statement

- The `goto` statement is capable of jumping to any statement in a function, provided that the statement has a ***label.***
- A label is just an identifier placed at the beginning of a statement:

  identifier : *statement*
- A statement may have more than one label.
- The `goto` statement itself has the form

  `goto` *identifier* ;
- Executing the statement `goto` *L*`;` transfers control to the statement that follows the label *L*, which must be in the same function as the `goto` statement itself.

The `goto` Statement

- If C didn't have a `break` statement, a `goto` statement could be used to exit from a loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    goto done;
done:
if (d < n)
  printf("%d is divisible by %d\n", n, d);
else
  printf("%d is prime\n", n);
```


The `goto` Statement

- The `goto` statement is rarely needed in everyday C programming.
- The `break`, `continue`, and `return` statements are sufficient to handle most situations that might require a `goto` in other languages.
- Nonetheless, the `goto` statement can be helpful once in a while.

The Null Statement

- A statement can be ***null***—devoid of symbols except for the semicolon at the end.
- The following line contains three statements:

```
i = 0; ; j = 1;
```

- The null statement is primarily good for one thing: writing loops whose bodies are empty.

The Null Statement

- Consider the following prime-finding loop:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

- If the `n % d == 0` condition is moved into the loop's controlling expression, the body of the loop becomes empty:

```
for (d = 2; d < n && n % d != 0; d++)
  /* empty loop body */ ;
```

- To avoid confusion, C programmers customarily put the null statement on a line by itself.

The Null Statement

- Accidentally putting a semicolon after the parentheses in an `if`, `while`, or `for` statement creates a null statement.
- Example:

```
if (d == 0);              /*** WRONG ***/
  printf("Error: Division by zero\n");
```

The call of `printf` isn't inside the `if` statement, so it's performed regardless of whether `d` is equal to 0.

Types

Basic Types

- C's ***basic*** (built-in) ***types:***
  - Integer types
  - Floating types
  - Character type

Integer Types

- Typical ranges on a 32-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	−32,768	32,767
`unsigned short int`	0	65,535
`int`	−2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	−2,147,483,648	2,147,483,647
`unsigned long int`	0	4,294,967,295

Integer Types

- Typical ranges on a 64-bit machine:

Type	*Smallest Value*	*Largest Value*
`short int`	−32,768	32,767
`unsigned short int`	0	65,535
`int`	−2,147,483,648	2,147,483,647
`unsigned int`	0	4,294,967,295
`long int`	-2^{63}	$2^{63}-1$
`unsigned long int`	0	$2^{64}-1$

- The `<limits.h>` header defines macros that represent the smallest and largest values of each integer type.

Integer Constants

- To force the compiler to treat a constant as a long integer, just follow it with the letter `L` (or `l`):

  `15L  0377L  0x7fffL`
- To indicate that a constant is unsigned, put the letter `U` (or `u`) after it:

  `15U  0377U  0x7fffU`
- `L` and `U` may be used in combination:

  `0xffffffffUL`

  The order of the `L` and `U` doesn't matter, nor does their case.

Floating Types

- C provides three ***floating types,*** corresponding to different floating-point formats:
  - `float` Single-precision floating-point
  - `double` Double-precision floating-point
  - `long double` Extended-precision floating-point

Floating Types

- `float` is suitable when the amount of precision isn't critical.
- `double` provides enough precision for most programs.
- `long double` is rarely used.
- Most modern computers follow the specifications in IEEE Standard 754.

Character Types

- The only remaining basic type is `char`, the character type.
- The values of type `char` can vary from one computer to another, because different machines may have different underlying character sets.

Character Sets

- Today's most popular character set is ***ASCII*** (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters.
- ASCII is often extended to a 256-character code known as ***Latin-1*** that provides the characters necessary for Western European and many African languages.

Character Sets

- A variable of type `char` can be assigned any single character:

```
char ch;

ch = 'a';    /* lower-case a */
ch = 'A';    /* upper-case A */
ch = '0';    /* zero         */
ch = ' ';    /* space        */
```

- Notice that character constants are enclosed in single quotes, not double quotes.

Operations on Characters

- Working with characters in C is simple, because of one fact: *C treats characters as small integers.*
- In ASCII, character codes range from 0000000 to 1111111, which we can think of as the integers from 0 to 127.
- The character `'a'` has the value 97, `'A'` has the value 65, `'0'` has the value 48, and `' '` has the value 32.
- Character constants actually have `int` type rather than `char` type.

Operations on Characters

- When a character appears in a computation, C uses its integer value.
- Consider the following examples, which assume the ASCII character set:

```
char ch;
int i;

i = 'a';       /* i is now 97   */
ch = 65;       /* ch is now 'A' */
ch = ch + 1;   /* ch is now 'B' */
ch++;          /* ch is now 'C' */
```


Operations on Characters

- Characters can be compared, just as numbers can.
- An `if` statement that converts a lower-case letter to upper case:

```
if ('a' <= ch && ch <= 'z')
  ch = ch - 'a' + 'A';
```

- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.

Reading and Writing Characters Using `scanf` and `printf`

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:

```
char ch;

scanf("%c", &ch);  /* reads one character */
printf("%c", ch);  /* writes one character */
```


Reading and Writing Characters Using `getchar` and `putchar`

- For single-character input and output, `getchar` and `putchar` are an alternative to `scanf` and `printf`.
- `putchar` writes a character:

```
putchar(ch);
```

- Each time `getchar` is called, it reads one character, which it returns:

```
ch = getchar();
```

- `getchar` returns an `int` value rather than a `char` value, so `ch` will often have type `int`.

Type Conversion

- Because the compiler handles these conversions automatically, without the programmer's involvement, they're known as ***implicit conversions.***
- C also allows the programmer to perform ***explicit conversions,*** using the cast operator.

Type Conversion

- Implicit conversions are performed:
  - When the operands in an arithmetic or logical expression don't have the same type.
  - When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
  - When the type of an argument in a function call doesn't match the type of the corresponding parameter.
  - When the type of the expression in a `return` statement doesn't match the function's return type.

The Usual Arithmetic Conversions

- The usual arithmetic conversions are applied to the operands of most binary operators.
- If `f` has type `float` and `i` has type `int`, the usual arithmetic conversions will be applied to the operands in the expression `f + i`.
- Clearly it's safer to convert `i` to type `float` (matching `f`'s type) rather than convert `f` to type `int` (matching `i`'s type).

The Usual Arithmetic Conversions

- Operand types can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***).
- Common promotions include the ***integral promotions,*** which convert a `char` or `short` to type `int` (or to `unsigned int` in some cases).

The Usual Arithmetic Conversions

- ***The type of either operand is a floating type.***
  - If one operand has type `long double`, then convert the other operand to type `long double`.
  - Otherwise, if one operand has type `double`, convert the other operand to type `double`.
  - Otherwise, if one operand has type `float`, convert the other operand to type `float`.
- Example: If one operand has type `long int` and the other has type `double`, the `long int` operand is converted to `double`.

The Usual Arithmetic Conversions

- ***Neither operand type is a floating type.*** First perform integral promotion on both operands.
- Then use the following diagram to promote the operand whose type is narrower:

```
unsigned long int
        ↑
     long int
        ↑
   unsigned int
        ↑
       int
```


The Usual Arithmetic Conversions

- When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value.
- This rule can cause obscure programming errors.
- It's best to use unsigned integers as little as possible and, especially, never mix them with signed integers.

Conversion During Assignment

- The usual arithmetic conversions don't apply to assignment.
- Instead, the expression on the right side of the assignment is converted to the type of the variable on the left side:

```
char c;
int i;
float f;
double d;

i = c;   /* c is converted to int   */
f = i;   /* i is converted to float */
d = f;   /* f is converted to double */
```


Conversion During Assignment

- Assigning a floating-point number to an integer variable drops the fractional part of the number:

```
int i;

i = 842.97;     /* i is now 842 */
i = -842.97;    /* i is now -842 */
```

- Assigning a value to a variable of a narrower type will give a meaningless result (or worse) if the value is outside the range of the variable's type:

```
c = 10000;    /*** WRONG ***/
i = 1.0e20;   /*** WRONG ***/
f = 1.0e100;  /*** WRONG ***/
```


Casting

- Although C's implicit conversions are convenient, we sometimes need a greater degree of control over type conversion.
- For this reason, C provides ***casts.***
- A cast expression has the form

  `(` *type-name* `)` *expression*

  type-name specifies the type to which the expression should be converted.

Casting

- Using a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;
frac_part = f - (int) f;
```

- The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast.
- Cast expressions enable us to document type conversions that would take place anyway:

```
i = (int) f;   /* f is converted to int */
```

61

Type Definitions

- A new type can be created by ***type definition:***

```
typedef int Bool;
```

- `Bool` can now be used in the same way as the built-in type names.
- Example:

```
Bool flag;   /* same as int flag; */
```

62

Type Definitions and Portability

- Type definitions are an important tool for writing portable programs.
- One of the problems with moving a program from one computer to another is that types may have different ranges on different machines.
- If `i` is an `int` variable, an assignment like

```
i = 100000;
```

  is fine on a machine with 32-bit integers, but will fail on a machine with 16-bit integers.

63

Type Definitions and Portability

- For greater portability, consider using `typedef` to define new names for integer types.
- Suppose that we're writing a program that needs variables capable of storing product quantities in the range 0–50,000.
- We could use `long` variables for this purpose, but we'd rather use `int` variables, since arithmetic on `int` values may be faster than operations on `long` values. Also, `int` variables may take up less space.

64

Type Definitions and Portability

- Instead of using the `int` type to declare quantity variables, we can define our own "quantity" type:

```
typedef int Quantity;
```

  and use this type to declare variables:

```
Quantity q;
```

- When we transport the program to a machine with shorter integers, we'll change the type definition:

```
typedef long Quantity;
```

- Note that changing the definition of `Quantity` may affect the way `Quantity` variables are used.

65

Type Definitions and Portability

- The C library itself uses `typedef` to create names for types that can vary from one C implementation to another; these types often have names that end with `_t`.
- Typical definitions of these types:

```
typedef long int ptrdiff_t;
typedef unsigned long int size_t;
typedef int wchar_t;
```

66

The `sizeof` Operator

- The value of the expression
  `sizeof ( ` *type-name* ` )`
  is an unsigned integer representing the number of bytes required to store a value belonging to *type-name.*
- `sizeof(char)` is always 1, but the sizes of the other types may vary.
- On a 32-bit machine, `sizeof(int)` is normally 4.

The `sizeof` Operator

- The `sizeof` operator can also be applied to constants, variables, and expressions in general.
  - If `i` and `j` are `int` variables, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i + j)`.