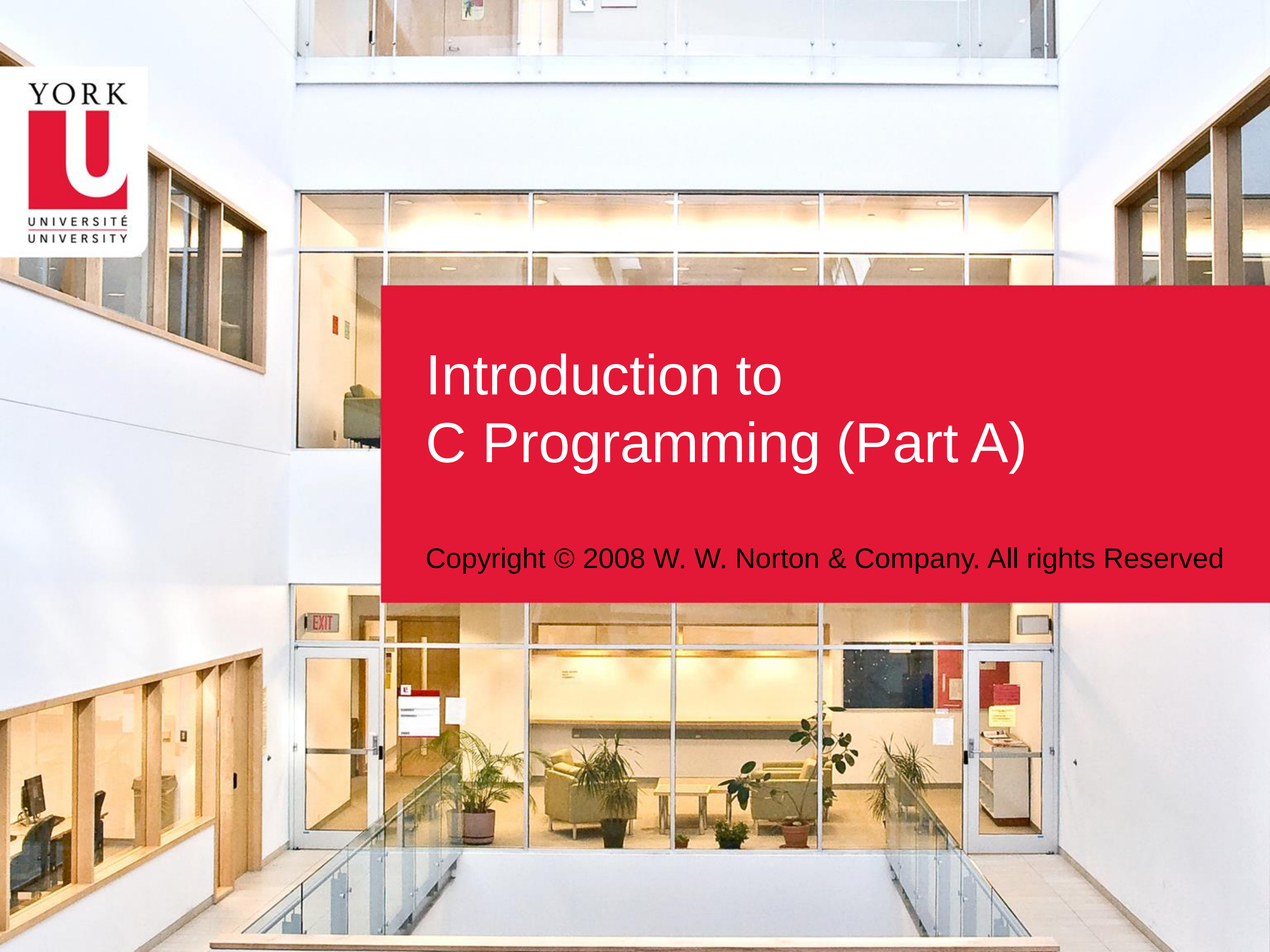


YORK U UNIVERSITÉ UNIVERSITY

Introduction to C Programming (Part A)

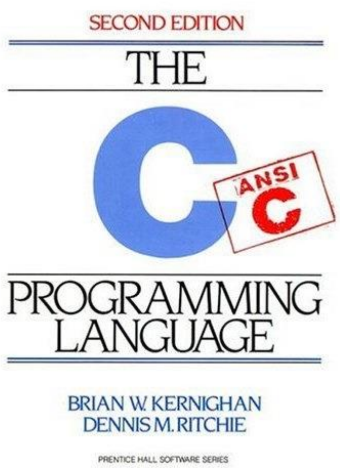
Overview (King Ch. 1-7)

- Introducing C (Ch. 1)
- C Fundamentals (Ch. 2)
- Formatted Input/Output (Ch. 3)
- Expressions (Ch. 4)
- Selection Statements (Ch. 5)
- Loops (Ch. 6)
- Basic Types (Ch. 7)

Chapter 1

Introducing C

Origins of C



- *C was developed at Bell Laboratories by mainly Ken Thompson & Dennis Ritchie (Turing Award in 1983)*
- *The language was stable enough by 1973 that UNIX could be rewritten in C.*
- *The 'R' in K&R C*

Standardization of C

- *K&R C*
  - Described in Kernighan and Ritchie, *The C Programming Language* (1978)
  - De facto standard
- *C89/C90*
  - ANSI standard X3.159-1989 (completed in 1988; formally approved in December 1989)
  - International standard ISO/IEC 9899:1990
- *C99*
  - International standard ISO/IEC 9899:1999
  - Incorporates changes from Amendment 1 (1995)

YORK UNIVERSITÉ UNIVERSITY

C-Based Languages

- ***C++*** includes all the features of C, but adds classes and other features to support object-oriented programming.
- ***Java*** is based on C++ and therefore inherits many C features.
- ***C#*** is a more recent language derived from C++ and Java.
- ***Perl*** has adopted many of the features of C.
- …

C Characteristics

- Properties of C
  - Low-level, Small, Permissive (assumes you know what you're doing)
- Strengths of C
  - Efficiency, Portability, Flexibility, Standard library, Integration with UNIX
- Weaknesses of C
  - Programs can be error-prone, difficult to understand, difficult to modify

Effective Use of C

- Learn how to avoid pitfalls.
- Use software tools (debuggers) to make programs more reliable.
- Take advantage of existing code libraries.
- Adopt a sensible set of coding conventions.
- Avoid “tricks” and overly complex code.
- Stick to the standard.

Chapter 2

C Fundamentals

Program: Printing a Pun

```
#include <stdio.h>

int main(void){
  printf("To C, or not to C: that is the question.\n");
  return 0;
}
```

- This program might be stored in a file named `pun.c`.
- The file name doesn't matter, but the `.c` extension is often required.

The GCC Compiler

- GCC is one of the most popular C compilers.
- GCC is supplied with Linux but is available for many other platforms as well.
- Using the gcc compiler (similar to using `cc)`:

```
% gcc -o pun pun.c
```

Directives

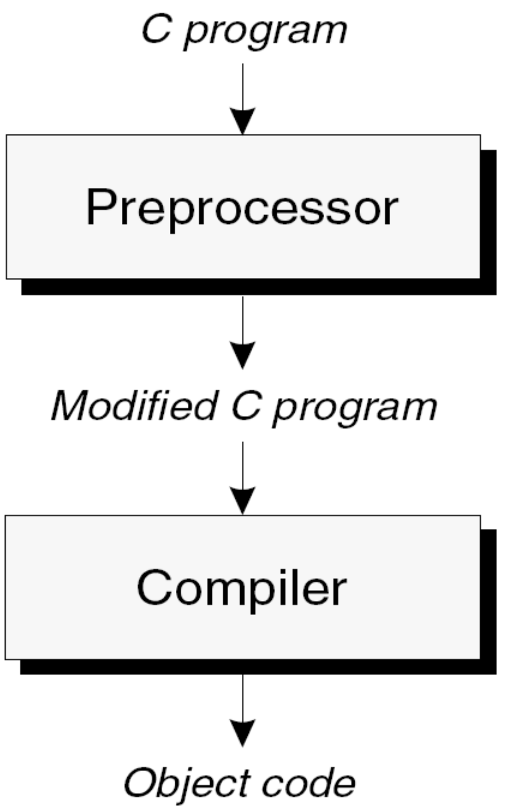
- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

```
#include <stdio.h>
```

`<stdio.h>` is a ***header*** containing information about C’s standard I/O library.

How the Preprocessor Works

- The preprocessor's role in the compilation process:



Compiling and Linking

- Before a program can be executed, three steps are usually necessary:
  - ***Preprocessing.*** The ***preprocessor*** obeys commands that begin with # (known as ***directives***)
  - ***Compiling.*** A ***compiler*** then translates the program into machine instructions (***object code***).
  - ***Linking.*** A ***linker*** combines the object code produced by the compiler with any additional code needed to yield a complete executable program.
- The preprocessor is usually integrated with the compiler.

Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a `return` statement to specify what value it “returns”:

```
return x + 1;
```

The Standard Library

- The C89 standard library is divided into 15 parts, with each part described by a header.
- C99 has an additional nine headers.

<assert.h>	<inttypes.h>†	<signal.h>	<stdlib.h>
<complex.h>†	<iso646.h>†	<stdarg.h>	<string.h>
<ctype.h>	<limits.h>	<stdbool.h>†	<tgmath.h>†
<errno.h>	<locale.h>	<stddef.h>	<time.h>
<fenv.h>†	<math.h>	<stdint.h>†	<wchar.h>†
<float.h>	<setjmp.h>	<stdio.h>	<wctype.h>†

†C99 only

The `main` Function

- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, many compilers will produce a warning message.

Statements

- A ***statement*** is a command to be executed when the program runs.
- `pun.c` uses only two kinds of statements
  - the `return` statement
  - the ***function call***
- C requires that each statement end with a semicolon. (two exceptions: compound statements, directives)
- `pun.c` calls `printf` to display a string:

```
printf("To C, or not to C: that is the question.\n");
```

Printing Strings

- The statement

  ```
  printf("To C, or not to C: that is the question.\n");
  ```

  could be replaced by two calls of `printf`:

  ```
  printf("To C, or not to C: ");
  printf("that is the question.\n");
  ```

- The new-line character can appear more than once in a string literal:

  ```
  printf("Brevity is the soul of wit.\n  --Shakespeare\n");
  ```

Comments

- A ***comment*** begins with `/*` and ends with `*/`.

```
/* This is a comment */
```

- Comments may extend over more than one line.

```
/* Name: pun.c
   Purpose: Prints a bad pun.
   Author: K. N. King */
```

- In C99, comments can also be written in the following way:

```
// This is a comment
```

Variables and Assignment

- Most programs need a way to store data temporarily during program execution.
- These storage locations are called ***variables.***
- Variables in C have
  - Type
  - Name
  - Value
  - Memory Address

Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

Declarations

- When `main` contains declarations, these must precede statements:

```
int main(void)
{
    declarations
    statements
}
```

- In C99, declarations don't have to come before statements.

Assignment

- A variable can be given a value by means of ***assignment:***

  `height = 8; /*`The number `8` is said to be a ***constant.****/

- Once a variable has been assigned a value, it can be used to compute the value of another variable:

```
height = 8;
length = 12;
width = 10;
vol = height * length * width; /* vol is now 960 */
```

- The right side of an assignment can be a formula (or ***expression,*** in C terminology) involving constants, variables, and operators.

Initialization

- The initial value of a variable may be included in its declaration:

  ```
  int height = 8;
  ```

  The value `8` is said to be an ***initializer.***

- Any number of variables can be initialized in the same declaration:

  ```
  int height = 8, length = 12, width = 10;
  ```

- Each variable requires its own initializer.

  ```
  int height, length, width = 10;
    /* initializes only width */
  ```

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the message

  `Height:` *h*

  where *h* is the current value of the `height` variable, we'd use the following call of `printf`:

```
printf("Height: %d\n", height);
```

- `%d` is a placeholder indicating where the value of `height` is to be filled in.

Printing the Value of a Variable

- `%d` is a placeholder for `int` variables
- `%f` is a placeholder for `float` variables
  - By default, `%f` displays a number with six digits after the decimal point. To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
  - Example: to print the line

    ```
    Profit: $2150.48
    ```

    use the following call of `printf`:

    ```
    printf("Profit: $%.2f\n", profit);
    ```

YORK UNIVERSITÉ UNIVERSITY

Printing the Value of Many Variables

- There’s no limit to the number of variables that can be printed by a single call of `printf`:

```
printf("Height: %d  Length: %d\n", height, length);
```

Printing Expressions

- `printf` can display the value of any numeric expression.
- The statements

  ```
  volume = height * length * width;
  printf("%d\n", volume);
  ```

  could be replaced by

  ```
  printf("%d\n", height * length * width);
  ```

Reading Input

- `scanf` is the C library's counterpart to `printf`.
- `scanf` requires a ***format string*** to specify the appearance of the input data.
- Example of using `scanf` to read an `int` value:

```
scanf("%d", &i);
/* reads an integer; stores into i */
```

- The `&` symbol is usually (but not always) required when using `scanf`.

Reading Input

- Reading a `float` value requires a slightly different call of `scanf`:

  ```
  scanf("%f", &x);
  ```

- `"%f"` tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn't have to).

Program: Converting from Fahrenheit to Celsius

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

- The program will allow temperatures that aren't integers.

celcius.c

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

  printf("Celsius equivalent: %.1f\n", celsius);

  return 0;
}
```

Program: Converting from Fahrenheit to Celsius

- We can name constants using a feature known as ***macro definition***:
```
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)
```
- When a program is compiled, the preprocessor replaces each macro by the value that it represents.
- Defining `SCALE_FACTOR` to be `(5.0f / 9.0f)` instead of `(5 / 9)` is important.
- Note the use of `%.1f` to display `celsius` with just one digit after the decimal point.

Chapter 3

Formatted Input/Output

The `printf` Function

- The `printf` function must be supplied with a ***format string,*** followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`,` *expr*`1,` *expr*`2,` …`);`

- The format string may contain both ordinary characters and ***conversion specifications,*** which begin with the `%` character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - `%d` is used for `int` values
  - `%f` is used for `float` values

The **`printf`** Function

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

Escape Sequences

- The `\n` code that is used in format strings is called an ***escape sequence.***
- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

- A partial list of escape sequences:

New line	`\n`	Backslash	`\\`
Horizontal tab	`\t`	Double Quotation	`\”`

The `scanf` Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

The **`scanf`** Function

- In many cases, a `scanf` format string will contain only conversion specifications:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively.

How `scanf` Works

- As it searches for a number, `scanf` ignores ***white-space characters*** (space, horizontal and vertical tab, form-feed, and new-line).
- A call of `scanf` that reads four numbers:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over several lines:

```
   1
-20   .3
   -4.0e3
```

- `scanf` sees a stream of characters (¤ represents new-line):

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```
(`s` = skipped; `r` = read)

- `scanf` “peeks” at the final new-line without reading it.

Program: Adding Fractions

- The `addfrac.c` program prompts the user to enter two fractions and then displays their sum.
- Sample program output:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n",result_num, result_denom);

  return 0;
}
```

Chapter 4

Expressions

Operators

- C emphasizes expressions rather than statements.
- Expressions are built from variables, constants, and **operators**.
- C has a rich collection of operators, including
  - arithmetic operators `(+, -, *, /, %)`
  - relational operators `(==, !=, >, <, >=, <=)`
  - logical operators `(!, &&, ||)`
  - assignment operators `(=, -=, *=, /=, %=)`
  - increment and decrement operators `(++, --)`

and others

Increment and Decrement Operators

- The increment and decrement operators are tricky:
  - They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).
- Example 1:

```
i = 1;
printf("i is %d\n", ++i);   /* prints "i is 2" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

- Example 2:

```
i = 1;
printf("i is %d\n", i++);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

Increment and Decrement Operators

- `++i` means “increment `i` immediately,” while `i++` means “use the old value of `i` for now, but increment `i` later.”
- **How much later**? The C standard doesn’t specify a precise time, but it’s safe to assume that `i` will be incremented **before the next statement is executed**.

Operator Precedence

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add **parentheses**, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.

Expression Evaluation

- Table of operators discussed so far:

Precedence	***Name***	***Symbol(s)***	***Associativity***
1	increment (postfix)	`++`	left
	decrement (postfix)	`--`	
2	increment (prefix)	`++`	right
	decrement (prefix)	`--`	
	unary plus	`+`	
	unary minus	`-`	
3	multiplicative	`*  /  %`	left
4	additive	`+  -`	left
5	relational	`<  <=  >  >=  ==  !=`	left
6	logical	`!  |  &&`	left
7	assignment	`=  *=  /=  %=  +=  -=`	right

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right and is ***right associative*** if it groups from right to left.
- For example, the binary arithmetic operators (`*`, `/`, `%`, `+`, and `–`) are all left associative, so

  `i - j – k` is equivalent to `(i - j) - k`
  `i * j / k` is equivalent to `(i * j) / k`

Implementation-Defined Behavior

- The C standard deliberately leaves parts of the language unspecified.
- Leaving parts of the language unspecified reflects C's emphasis on efficiency, which often means matching the way that hardware behaves.
- It's best to avoid writing programs that depend on implementation-defined behavior.

Order of Subexpression Evaluation

- Example:

```
i = 2;
j = i * i++;
```

- It’s natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:
  1. The second operand (the original value of i) is fetched, then `i` is incremented.
  2. The first operand (the new value of `i`) is fetched.
  3. The new and old values of `i` are multiplied, yielding 6.

Undefined Behavior

- Statements such as `j = i * i++;` cause ***undefined behavior.***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers.
  - The program may not compile in the first place.
  - If it compiles it may not run.
  - If it does run, the program may crash, behave erratically, or produce meaningless results.
- Undefined behavior should be avoided.

Chapter 5

Selection Statements

Statements

- Most of C's statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break`, `continue`, and `goto`. (`return` also belongs in this category.)
- Other C statements:
  - Compound statement
  - Null statement

Logical Expressions

- Several of C's statements must test the value of an expression to see if it is "**true**" or "**false**."
- For example, an `if` statement might need to test the expression `i < j`; a true value would indicate that `i` is less than `j`.
- In many programming languages, an expression such as `i < j` would have a special "Boolean" or "logical" type.
- In C, a comparison such as `i < j` yields an integer: either 0 (false) or 1 (true).

Boolean Values in C89

- For many years, the C language lacked a Boolean type, and there is none defined in the C89 standard.
- Ways to work around this limitation
  - declare an `int` variable and then assign it either 0 or 1:

```
int flag;
flag = 0;
…
flag = 1;
```

  - define macros with names such as `TRUE` and `FALSE`:

```
#define TRUE      1
#define FALSE     0
flag = FALSE;
…
flag = TRUE;
```

Boolean Values in C99

- C99 provides the `_Bool` type.
  - A Boolean variable can be declared by writing

```
_Bool flag;
```

- Or include `<stdbool.h>` header that:
  - defines a macro, `bool`, that stands for `_Bool`
  - supplies macros named `true` and `false`, which stand for 1 and 0, respectively, so:

```
bool flag; /* same as _Bool flag;*/
flag = false;
…
flag = true;
```

The `if` Statement

- The `if` statement allows a program to choose between two alternatives by testing an expression.
- Syntax

  `if (` *expression* `)` *statement*

- Example:

```
if (line_num == MAX_LINES)
   line_num = 0;
```

- Confusing `==` (equality) with `=` (assignment) is perhaps the most common C programming error.

Compound Statements

- To make an `if` statement control two or more statements, use a ***compound statement.***
- A compound statement has the form

  `{` *statements* `}`
- Example of a compound statement used inside an `if` statement:

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

The **`else`** Clause

- An `if` statement may have an `else` clause:

  `if (` *expression* `)` *statement* `else` *statement*
- The statement that follows the word `else` is executed if the expression has the value 0.
- Example:

```
if (i > j)
  max = i;
else
  max = j;
```

Cascaded **`if`** Statements

- This layout avoids the problem of excessive indentation when the number of tests is large:

```
if ( expression )
   statement
else if ( expression )
   statement
…
else if ( expression )
   statement
else
   statement
```

Example Cascaded **`if`** Statement

- A cascaded `if` statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

The **`switch`** Statement

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

- A `switch` statement
  - may be easier to read than a cascaded if statement
  - often faster than if statements. **Why?**

The Role of the `break` Statement

- Executing a `break` statement causes the program to "break" out of the `switch` statement; execution continues at the next statement after the `switch`.
- The `switch` statement is really a form of "computed jump."
- When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression.
- A case label is nothing more than a marker indicating a position within the `switch`.

The Role of the **`break`** Statement

- Without `break` (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Illegal grade");
}
```

- If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal grade
```

Chapter 6

Loops

Iteration Statements

- C provides three iteration statements:
  - The `while` statement
  - The `do` statement
  - The `for` statement

The `while` Statement

- The `while` statement has the form

  `while (` *expression* `)` *statement*

- *expression* is the controlling expression; *statement* is the loop body.
- Example:

```
i = 10;

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

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.

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);
```

The `for` Statement

- The `for` statement is closely related to the `while` statement and can be replaced by an equivalent `while` loop:

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

- *expr1* is an initialization step that's performed once
- *expr2* controls loop termination
- *expr3* is an operation to be performed at the end of each loop iteration.

Infinite Loops

- C programmers sometimes deliberately create an ***infinite loop***:
  - Using `while` loop

    `while (1) …`

  - Using `for` loop

    `for (;;) …`

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.
- Example:

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

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.
- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:

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

The **`break`** Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end.
- Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape only one level 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 */
}
```

Chapter 7

Basic Type

Basic Types

- C's ***basic*** (built-in) ***types:***
  - Integer types, including long integers, short integers, signed and unsigned integers
  - Floating types (`float`, `double`, and `long double`) can have a fractional part as well
  - `char`
  - `_Bool` (C99)

Integer Type Specifiers

- Sign (the leftmost bit is reserved for the sign)
  - signed (default)
  - unsigned (primarily useful for systems programming and low-level, machine-dependent applications)
- Long/Short (bits to be used)
  - long (integers may have more bits than ordinary integers)
  - short (integers may have fewer bits)
- Only six combinations produce different types:

```
short int     unsigned short int
int           unsigned int
long int      unsigned long int
```

- The order of the specifiers doesn't matter. Also, the word `int` can be dropped (`long int` can be abbreviated to just `long`).

YORK U UNIVERSITÉ UNIVERSITY

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

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

Integer Overflow

- When arithmetic operations are performed on integers, it's possible that the result will be too large to represent.
- For example, when an arithmetic operation is performed on two `int` values, the result must be able to be represented as an `int`.
- If the result can't be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred.

Integer Overflow

- The behavior when integer overflow occurs depends on whether the operands were signed or unsigned.
  - When overflow occurs during an operation on *signed* integers, the program's behavior is undefined.
  - When overflow occurs during an operation on *unsigned* integers, the result *is* defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result.

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
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).
  - Numbers are stored in a form of scientific notation, with each number having a ***sign,*** an ***exponent,*** and a ***fraction.***

Floating Types

- Characteristics of `float` and `double` when implemented according to the IEEE standard:

Type	***Smallest Positive Value***	***Largest Value***	***Precision***
`float`	1.17549×10^{-38}	3.40282×10^{38}	6 digits
`double`	2.22507×10^{-308}	1.79769×10^{308}	15 digits

- In fact, on some machines, `float` may have the same set of values as `double`, or `double` may have the same values as `long double`.
- Characteristics of the floating types can be found in the `<float.h>` header.

Character Sets

- The values of type `char` can vary from one computer to another, because different machines may have different underlying 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 Types

- 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 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.

ASCII Table (128 first characters)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

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.
- What is the purpose of the above code snippet?

Character-Handling Functions

- The C library provides many useful character-handling functions. To use them programs need to have the following directive at the top:

```
#include <ctype.h>
```

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);
    ```
  - `getchar` it reads one character, which it returns:

    ```
    ch = getchar();
    ```
- Moving the call of `getchar` into the controlling expression allows us to condense a loop that reads many characters:

```
while ((ch = getchar()) != '\n')
  ;
```

Type Conversion

- For a computer to perform an arithmetic operation, the operands must usually be of the same size (the same number of bits) and be stored in the same way.
- When operands of different types are mixed in expressions, the C compiler may have to generate instructions that change the types of some operands so that hardware will be able to evaluate the expression.
  - If we add a 16-bit `short` and a 32-bit `int`, the compiler will arrange for the `short` value to be converted to 32 bits.
  - If we add an `int` and a `float`, the compiler will arrange for the `int` to be converted to `float` format.

YORK UNIVERSITÉ UNIVERSITY

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.
- The rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.

The Usual Arithmetic Conversions

- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
    - Convert the non-floating type operand to the floating type of the other operand.
  - 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:

      `int` → `unsigned int` → `long int` → `unsigned long int`

The Usual Arithmetic Conversions

- Example of the usual arithmetic conversions:

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;

i = i + c;     /* c is converted to int                */
i = i + s;     /* s is converted to int                */
u = u + i;     /* i is converted to unsigned int       */
l = l + u;     /* u is converted to long int           */
ul = ul + l;   /* l is converted to unsigned long int */
f = f + ul;    /* ul is converted to float             */
d = d + f;     /* f is converted to double             */
ld = ld + d;   /* d is converted to long double        */
```

Explicit Conversion: Casting

- We sometimes need a greater degree of control over type conversion. C provides ***casts.***
- A cast expression has the form

  (*type-name*) *expression*

  type-name specifies the type to which the expression should be converted.
- Example using a cast expression to compute the fractional part of a `float` value:

```
float f, frac_part;

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

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.
- For example, on a 32-bit machine, `sizeof(int)` is normally 4.