Chapter 7

Basic Types

C’s basic (built-in) types:

- Integer types, including long integers, short integers, and unsigned integers
- Floating types (float, double, and long double)
- char
- _Bool (C99)
Integer Types

- C supports two fundamentally different kinds of numeric types: integer types and floating types.
- Values of an **integer type** are whole numbers.
- Values of a floating type can have a fractional part as well.
- The integer types, in turn, are divided into two categories: signed and unsigned.

Signed and Unsigned Integers

- The leftmost bit of a **signed** integer (known as the **sign bit**) is 0 if the number is positive or zero, 1 if it’s negative.
  - 2’s complement format is used for signed numbers
  - The largest 16-bit integer has the binary representation 0111111111111111, which has the value 32,767 (2^{15} – 1).
  - The largest 32-bit integer is 01111111111111111111111111111111, which has the value 2,147,483,647 (2^{31} – 1).
- An integer with no sign bit (the leftmost bit is considered part of the number’s magnitude) is said to be **unsigned**.
  - The largest 16-bit unsigned integer is 65,535 (2^{16} – 1).
  - The largest 32-bit unsigned integer is 4,294,967,295 (2^{32} – 1).
Signed and Unsigned Integers

- By default, integer variables are signed in C—the leftmost bit is reserved for the sign.
  - Signed integers are stored in the 2’s complement number format system
- To tell the compiler that a variable has no sign bit, declare it to be unsigned.

Integer Types

- The `int` type is usually 32 bits, but may be 16 bits on older CPUs.
  - `Long` integers may have more bits than ordinary integers; `short` integers may have fewer bits.
  - The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form integer types.
- Only six combinations produce different types:
  ```
  short int  unsigned short int
  int        unsigned int
  long int   unsigned long int
  ```
Integer Types

• Typical ranges of values for the integer types on a 16-bit machine:

<table>
<thead>
<tr>
<th>Type</th>
<th>Smallest Value</th>
<th>Largest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>long int</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>

Integer Types

• Typical ranges on a 32-bit machine:

<table>
<thead>
<tr>
<th>Type</th>
<th>Smallest Value</th>
<th>Largest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long int</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>
 Integer Types

• Typical ranges on a 64-bit machine:

<table>
<thead>
<tr>
<th>Type</th>
<th>Smallest Value</th>
<th>Largest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>−32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int</td>
<td>−2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
<tr>
<td>long int</td>
<td>−2^{63}</td>
<td>2^{63}–1</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0</td>
<td>2^{64}–1</td>
</tr>
</tbody>
</table>

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

 Integer Constants

• **Constants** are numbers that appear in the text of a program.
• C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).
Octal and Hexadecimal Numbers

• Octal numbers use only the digits 0 through 7.
• Each position in an octal number represents a power of 8.
  – The octal number 237 represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = 159$.
• A hexadecimal (or hex) number is written using the digits 0 through 9 plus the letters A through F, which stand for 10 through 15, respectively.
  – The hex number 1AF has the decimal value $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = 431$.

Integer Constants

• **Decimal** constants contain digits between 0 and 9, but must not begin with a zero:
  15  255  32767
• **Octal** constants contain only digits between 0 and 7, and must begin with a zero:
  017  0377  077777
• **Hexadecimal** constants contain digits between 0 and 9 and letters between a and f, and always begin with 0x:
  0xf  0xff  0x7fff
• The letters in a hexadecimal constant may be either upper or lower case:
  0xff  0xFF  0x7ff  0Xff  0XFF  0xFF  0XFf  0XFF
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.

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.

Reading and Writing Integers

- Reading and writing unsigned, short, and long integers requires new conversion specifiers.
- When reading or writing an unsigned integer, use the letter u, o, or x instead of d in the conversion specification.

```c
unsigned int u;

scanf("%u", &u);  /* reads u in base 10 */
printf("%u", u);    /* writes u in base 10 */
scanf("%o", &u);   /* reads u in base 8 */
printf("%o", u);    /* writes u in base 8 */
scanf("%x", &u);  /* reads u in base 16 */
printf("%x", u);    /* writes u in base 16 */
```
Reading and Writing Integers

• When reading or writing a short integer, put the letter h in front of d, o, u, or x:

```c
short s;
scanf("%hd", &s);
printf("%hd", s);
```

• When reading or writing a long integer, put the letter l (“ell,” not “one”) in front of d, o, u, or x.

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.
- The C standard doesn’t state how much precision the float, double, and long double types provide, since that depends on how numbers are stored.
- Most modern computers follow the specifications in IEEE Standard 754 (also known as IEC 60559).

The IEEE Floating-Point Standard

- IEEE Standard 754 was developed by the Institute of Electrical and Electronics Engineers.
- It has two primary formats for floating-point numbers: single precision (32 bits) and double precision (64 bits).
- Numbers are stored in a form of scientific notation, with each number having a **sign**, an **exponent**, and a **fraction (mantissa)**.
- In single-precision format, the exponent is 8 bits long, while the fraction occupies 23 bits. The maximum value is approximately $3.40 \times 10^{38}$, with a precision of about 6 decimal digits.
Floating Types

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Smallest Positive Value</th>
<th>Largest Value</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>$1.17549 \times 10^{-38}$</td>
<td>$3.40282 \times 10^{38}$</td>
<td>6 digits</td>
</tr>
<tr>
<td>double</td>
<td>$2.22507 \times 10^{-308}$</td>
<td>$1.79769 \times 10^{308}$</td>
<td>15 digits</td>
</tr>
</tbody>
</table>

- On computers that don't follow the IEEE standard, this table won't be valid.
- 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`.

Floating Types

- Macros that define the characteristics of the floating types can be found in the `<float.h>` header.
- In C99, the floating types are divided into two categories.
  - **Real floating types** (`float`, `double`, `long double`)
  - **Complex types** (`float_Complex`, `double_Complex`, `long double_Complex`)
Floating Constants

• Floating constants can be written in a variety of ways.

• Valid ways of writing the number 57.0:
  57.0  57.  57.0e0  57E0  5.7e1  5.7e+1
  .57e2  570.e-1

• A floating constant must contain a decimal point and/or an exponent; the exponent indicates the power of 10 by which the number is to be scaled.

• If an exponent is present, it must be preceded by the letter E (or e). An optional + or – sign may appear after the E (or e).

Floating Constants

• By default, floating constants are stored as double-precision numbers.

• To indicate that only single precision is desired, put the letter F (or f) at the end of the constant (for example, 57.0F).

• To indicate that a constant should be stored in long double format, put the letter L (or l) at the end (57.0L).
Reading and Writing Floating-Point Numbers

- The conversion specifications %e, %f, and %g are used for reading and writing single-precision floating-point numbers.
- When reading a value of type double, put the letter l in front of e, f, or g:
  ```c
double d;
scanf("%lf", &d);
```
- *Note:* Use l only in a scanf format string, not a printf string.
- In a printf format string, the e, f, and g conversions can be used to write either float or double values.
- When reading or writing a value of type long double, put the letter L in front of e, f, or g.

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:

```c
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 (in binary), 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:

```c
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:
  ```c
  if ('a' <= ch && ch <= 'z')
      ch = ch - 'a' + 'A';
  ```
- Comparisons such as `'a' <= ch` are done using the integer values of the characters involved.
- These values depend on the character set in use, so programs that use `<, <=, >, and >= to compare characters may not be portable.

Operations on Characters

- The fact that characters have the same properties as numbers has advantages.
- For example, it is easy to write a `for` statement whose control variable steps through all the upper-case letters:
  ```c
  for (ch = 'A'; ch <= 'Z'; ch++) ...
  ```
Signed and Unsigned Characters

- The char type—like the integer types—exists in both signed and unsigned versions.
- Signed characters normally have values between –128 and 127. Unsigned characters have values between 0 and 255.
- Some compilers treat char as a signed type, while others treat it as an unsigned type. Most of the time, it doesn’t matter.
- C allows the use of the words signed and unsigned to modify char:
  ```c
  signed char sch;
  unsigned char uch;
  ```

Character-Handling Functions

- Calling C’s toupper library function is a fast and portable way to convert case:
  ```c
  ch = toupper(ch);
  ```
- toupper returns the upper-case version of its argument.
- Programs that call toupper need to have the following #include directive at the top:
  ```c
  #include <ctype.h>
  ```
- The C library provides many other useful character-handling functions.
Reading and Writing Characters Using scanf and printf

- The `%c` conversion specification allows `scanf` and `printf` to read and write single characters:
  ```c
  char ch;
  scanf("%c", &ch);  /* reads one character */
  printf("%c", ch);  /* writes one character */
  ```
- `scanf` doesn't skip white-space characters.
- To force `scanf` to skip white space before reading a character, put a space in its format string just before `%c`:
  ```c
  scanf(" %c", &ch);
  ```

Since `scanf` doesn’t normally skip white space, it’s easy to detect the end of an input line: check to see if the character just read is the new-line character.

A loop that reads and ignores all remaining characters in the current input line:
```c
do {
    scanf("%c", &ch);
} while (ch != '\n');
```

When `scanf` is called the next time, it will read the first character on the next input line.
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:
  ```c
  putchar(ch);
  ```
- Each time `getchar` is called, it reads one character, which it returns:
  ```c
  ch = getchar();
  ```
- `getchar` returns an `int` value rather than a `char` value, so `ch` will often have type `int`.
- Like `scanf`, `getchar` doesn’t skip white-space characters as it reads.

---

Reading and Writing Characters Using getchar and putchar

- Consider the `scanf` loop that we used to skip the rest of an input line:
  ```c
  do {
      scanf("%c", &ch);
  } while (ch != '\n');
  ```
- Rewriting this loop using `getchar` gives us the following:
  ```c
  do {
      ch = getchar();
  } while (ch != '\n');
  ```
Reading and Writing Characters Using getchar and putchar

• Moving the call of getchar into the controlling expression allows us to condense the loop:
  
  ```c
  while ((ch = getchar()) != '\n')
  ;
  ```

• The ch variable isn’t even needed; we can just compare the return value of getchar with the newline character:
  
  ```c
  while (getchar() != '\n')
  ;
  ```

• getchar is useful in loops that skip characters as well as loops that search for characters.

• A statement that uses getchar to skip an indefinite number of blank characters:
  
  ```c
  while ((ch = getchar()) == ' ')
  ;
  ```

• When the loop terminates, ch will contain the first nonblank character that getchar encountered.
Reading and Writing Characters Using getchar and putchar

- Be careful when mixing `getchar` and `scanf`.
- `scanf` has a tendency to leave behind characters that it has “peeked” at but not read, including the new-line character:
  ```c
  printf("Enter an integer: ");
  scanf("%d", &i);
  printf("Enter a command: ");
  command = getchar();
  `scanf` will leave behind any characters that weren’t consumed during the reading of `i`, including (but not limited to) the new-line character.
- `getchar` will fetch the first leftover character.

Program: Determining the Length of a Message

- The `length.c` program displays the length of a message entered by the user:
  ```c
  Enter a message: Brevity is the soul of wit.
  Your message was 27 character(s) long.
  ```
- The length includes spaces and punctuation, but not the new-line character at the end of the message.
- We could use either `scanf` or `getchar` to read characters; most C programmers would choose `getchar`.
- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`. 
```c
/* Determines the length of a message */
#include <stdio.h>

int main(void) {
    char ch;
    int len = 0;

    printf("Enter a message: ");
    ch = getchar();
    while (ch != '\n') {
        len++;
        ch = getchar();
    }
    printf("Your message was %d character(s) long.\n", len);

    return 0;
}
```

```c
/* Determines the length of a message */
#include <stdio.h>

int main(void) {
    int len = 0;

    printf("Enter a message: ");
    while (getchar() != '\n')
        len++;
    printf("Your message was %d character(s) long.\n", len);

    return 0;
}
```
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 \texttt{short} and a 32-bit \texttt{int}, the compiler will arrange for the \texttt{short} value to be converted to 32 bits.
  - If we add an \texttt{int} and a \texttt{float}, the compiler will arrange for the \texttt{int} to be converted to \texttt{float} format.

Type Conversion

- Because the compiler handles these conversions automatically, without the programmer’s involvement, they’re known as \textit{implicit conversions}.
- C also allows the programmer to perform \textit{explicit conversions}, using the cast operator.
- The rules for performing implicit conversions are somewhat complex, primarily because C has so many different arithmetic types.
Type Conversion

- Implicit conversions are performed:
  - When the operands in an arithmetic or logical expression don’t have the same type. (C performs what are known as the usual arithmetic conversions.)
  - 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.
- Chapter 9 discusses the last two cases.

The Usual Arithmetic Conversions

- Strategy behind the usual arithmetic conversions: convert operands to the “narrowest” type that will safely accommodate both values.
- 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 character or short integer to type int (or to unsigned int in some cases).
- The rules for performing the usual arithmetic conversions can be divided into two cases:
  - The type of either operand is a floating type.
  - Neither operand type is a floating type.
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 appropriate and, especially, never mix them with signed integers.

Example of the usual arithmetic conversions:

```c
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 */
```
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:

```c
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 */
```

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

Casting

- Cast expressions also let us force the compiler to perform conversions.
- Example:

  ```c
  float quotient;
  int dividend, divisor;
  
  quotient = dividend / divisor;
  ```

- To avoid truncation during division, we need to cast one of the operands:

  ```c
  quotient = (float) dividend / divisor;
  ```

- Casting `dividend` to `float` causes the compiler to convert `divisor` to `float` also.
Casting

- C regards \( \text{type-name} \) as a unary operator.
- Unary operators have higher precedence than binary operators, so the compiler interprets
  \((\text{float}) \text{dividend} \div \text{divisor}\)
as
  \(((\text{float}) \text{dividend}) \div \text{divisor}\)
- Other ways to accomplish the same effect:
  \[
  \text{quotient} = \text{dividend} \div (\text{float}) \text{divisor}；
  \text{quotient} = (\text{float}) \text{dividend} \div (\text{float}) \text{divisor}；
  \]

Casting

- Casts are sometimes necessary to avoid overflow:
  \[
  \text{long} \ i; \\
  \text{int} \ j = 1000; \\
  i = j \times j; \quad /* \text{overflow may occur} */
  \]
- Using a cast avoids the problem:
  \[
  i = (\text{long}) \ j \times j; 
  \]
- The statement
  \[
  i = (\text{long}) \ (j \times j); \quad /* \text{WRONG} */
  \]
  wouldn’t work, since the overflow would already have occurred by the time of the cast.
**Type Definitions**

- The `#define` directive can be used to create a “Boolean type” macro:
  ```
  #define BOOL int
  ```
- There’s a better way using a feature known as a **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; */
  ```

**Advantages of Type Definitions**

- Type definitions can make a program more understandable.
- If the variables `cash_in` and `cash_out` will be used to store dollar amounts, declaring `Dollars` as
  ```
  typedef float Dollars;
  ```
  and then writing
  ```
  Dollars cash_in, cash_out;
  ```
  is more informative than just writing
  ```
  float cash_in, cash_out;
  ```
Advantages of Type Definitions

- Type definitions can also make a program easier to modify.
- To redefine Dollars as double, only the type definition need be changed:
  ```c
  typedef double Dollars;
  ```
- Without the type definition, we would need to locate all float variables that store dollar amounts and change their declarations.

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:
  ```c
  typedef long int ptdiff_t;
  typedef unsigned long int size_t;
  typedef int wchar_t;
  ```
- In C99, the <stdint.h> header uses typedef to define names for integer types with a particular number of bits.
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)`.

• When applied to an expression—as opposed to a type—`sizeof` doesn't require parentheses.
  – We could write `sizeof i` instead of `sizeof(i)`.

• Parentheses may be needed anyway because of operator precedence.
  – The compiler interprets `sizeof i + j` as `(sizeof i) + j`, because `sizeof` takes precedence over binary `+`. 
The sizeof Operator

- Printing a sizeof value requires care, because the type of a sizeof expression is an implementation-defined type named size_t.
- In C89, it's best to convert the value of the expression to a known type before printing it:
  
  ```c
  printf("Size of int: %lu\n",
         (unsigned long) sizeof(int));
  ```
- The printf function in C99 can display a size_t value directly if the letter z is included in the conversion specification:
  
  ```c
  printf("Size of int: %zu\n", sizeof(int));
  ```