Chapter 7

Basic Types

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`

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.
- 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.
- To tell the compiler that a variable has no sign bit, declare it to be `unsigned`.
- Unsigned numbers are primarily useful for systems programming and low-level, machine-dependent applications.

Integer Types

- The range of values represented by each of the six integer types varies from one machine to another.
- However, the C standard requires that `short int`, `int`, and `long int` must each cover a certain minimum range of values.
- Also, `int` must not be shorter than `short int`, and `long int` must not be shorter than `int`.

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 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.

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.

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

- 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`.

Reading and Writing Floating-Point Numbers

- The conversion specification `%f` is used for reading and writing single-precision floating-point numbers.
- When reading a value of type `double`, put the letter `l` in front of `f`:

```
double d;

scanf("%lf", &d);
```

- *Note:* Use `l` only in a `scanf` format string, not a `printf` string.
- In a `printf` format string, the `f` conversion 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 `f`.

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.

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.
- 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:

```
for (ch = 'A'; ch <= 'Z'; ch++) …
```

- Disadvantages of treating characters as numbers:
  - Can lead to errors that won't be caught by the compiler.
  - Allows meaningless expressions such as `'a' * 'b' / 'c'`.
  - Can hamper portability, since programs may rely on assumptions about the underlying character set.

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`.
- 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:

```
do {
  scanf("%c", &ch);
} while (ch != '\n');
```

- Rewriting this loop using `getchar` gives us the following:

```
do {
  ch = getchar();
} while (ch != '\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.

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.

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

- 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).
  - When an integer is converted to `float`, the worst that can happen is a minor loss of precision.
  - Converting a floating-point number to `int`, on the other hand, causes the fractional part of the number to be lost. Worse still, the result will be meaningless if the original number is larger than the largest possible integer or smaller than the smallest integer.

The Usual Arithmetic Conversions

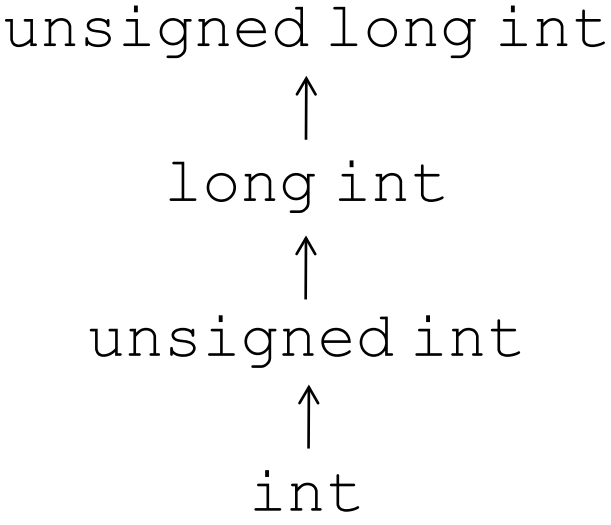
- Strategy behind the usual arithmetic conversions: convert operands to the "narrowest" type that will safely accommodate both values.
- Operand types can often be made to match by converting the operand of the narrower type to the type of the other operand (this act is known as ***promotion***).
- Common 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:



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        */
```

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 */
```

Casting

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

```
float quotient;
int dividend, divisor;

quotient = dividend / divisor;
```

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

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

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

Casting

- C regards (*type-name*) as a unary operator.
- Unary operators have higher precedence than binary operators, so the compiler interprets

```
(float) dividend / divisor
```

  as

```
((float) dividend) / divisor
```

- Other ways to accomplish the same effect:

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

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.