Chapter 7

Basic Types

Basic Types

- C's ***basic*** (built-in) ***date types are:***
  - **Integer** types, including
    - signed/unsigned integers
    - long integers
    - short integers
  - **Floating** types, including
    - float
    - double
    - long double
  - **Character** types, including
    - signed/unsigned characters

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 the number is negative
  - The largest 16-bit signed integer has the binary representation `0111111111111111`, which has the value 32,767 = $(2^{15} – 1)$, i.e., 32K -1
  - The largest 32-bit signed integer is `01111111111111111111111111111111`, which has the value 2,147,483,647 = $(2^{31} – 1)$, i.e., 2G -1
- An integer with no sign bit is said to be ***unsigned***
  - The largest 16-bit unsigned integer is 65,535 = $(2^{16} – 1)$, i.e., 64K -1
  - The largest 32-bit unsigned integer is 4,294,967,295 = $(2^{32} – 1)$, i.e., 4G -1
- By default, integer variables are signed integers
- To tell the compiler that a variable has no sign bit, declare it as `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 than ordinary integers
- The specifiers `long` and `short`, as well as `signed` and `unsigned`, can be combined with `int` to form various integer types
- Only six combinations produce different types:

```
signed   short int  same as short int  same as short
unsigned short int  same as unsigned short

signed         int  same as       int  same as signed
unsigned       int  same as unsigned

signed   long  int  same as long int   same as long
unsigned long  int  same as unsigned long
```

- The order of the specifiers does not matter
- The word `signed` can be dropped
- The word `int` can be dropped

Integer Types

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

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

- Typical ranges of values for the integer types 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 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)
  - hexadecimal (base 16)

Octal and Hexadecimal Numbers

Decimal	Octal	Hexadecimal
0	0	0
1	1	1
…..	…..	…..
7	7	7
8	$(10)_8$	8
9	$(11)_8$	9
10	$(12)_8$	$(A)_{16}$
11	$(13)_8$	$(B)_{16}$
12	$(14)_8$	$(C)_{16}$
13	$(15)_8$	$(D)_{16}$
14	$(16)_8$	$(E)_{16}$
15	$(17)_8$	$(F)_{16}$
16	$(20)_8$	$(10)_{16}$
17	$(21)_8$	$(11)_{16}$
18	$(22)_8$	$(12)_{16}$
….	….	…..

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 $(237)_8$ represents the decimal number $2 \times 8^2 + 3 \times 8^1 + 7 \times 8^0 = 128 + 24 + 7 = (159)_{10}$
- 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 $(1AF)_{16}$ has the decimal value: $1 \times 16^2 + 10 \times 16^1 + 15 \times 16^0 = 256 + 160 + 15 = (431)_{10}$

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 must begin with `0x` or `0X`:

  `0xf   0xff   0X7fff`

- The letters in a hexadecimal constant may be either upper or lower case:

  `0xff   0xfF   0xFf   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 force the compiler to treat a constant as a unsigned, just follow it with the letter `U` (or `u`):

  `15U   0377U   0x7fffU`

- `L` and `U` may be used in combination:

  `0xffffffffUL`

  The order of the `L` and `U` does not matter, nor does their case

Integer Overflow

- When arithmetic operations are performed on integers, it is possible that the result will be too large to represent it
- If the result can not be represented as an `int` (because it requires too many bits), we say that ***overflow*** has occurred

Reading and Writing Integers

- When reading or writing an *unsigned* decimal integer, put the letter `u` instead of `d` in the conversion specification

```
unsigned int i;
scanf("%u", &i);
printf("%u", i);
```

- When reading or writing an *unsigned* octal integer, put the letter `o` instead of `d` in the conversion specification

```
unsigned int i;
scanf("%o", &i);
printf("%o", i);
```

- When reading or writing a *unsigned* hexadecimal integer, put the letter `x` instead of `d` in the conversion specification

```
unsigned int i;
scanf("%x", &i);
printf("%x", i);
```

Reading and Writing Integers

- When reading or writing a *short* integer, put the letter `h` *in front of* `d`, `u`, `o`, or `x` in the conversion specification

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

- When reading or writing a *long* integer, put the letter `l` ("ell," not "one") *in front of* `d`, `u`, `o`, or `x` in the conversion specification

```
long i;
scanf("%ld", &i);
printf("%ld", i);
```


Program: Summing a Series of Numbers (Revisited)

- The `sum.c` program (Chapter 6) sums a series of integers
- One problem with this program is that the sum (or one of the input numbers) might exceed the largest value allowed for an `int` variable
- Here is what might happen if the program is run on a machine whose integers are 16 bits long:

```
This program sums a series of integers.
Enter integers (0 to terminate): 10000 20000 30000 0
The sum is: -5536
```

- When overflow occurs with signed numbers, the outcome is undefined.
- The program can be improved by using `long` variables

sum2.c

```
/* Sums a series of numbers (using long variables) */

#include <stdio.h>

int main(void)
{
  long n, sum = 0;

  printf("This program sums a series of integers.\n");
  printf("Enter integers (0 to terminate): ");

  scanf("%ld", &n);
  while (n != 0) {
    sum += n;
    scanf("%ld", &n);
  }
  printf("The sum is: %ld\n", sum);

  return 0;
}
```


Floating Types

- C provides three ***floating types***, corresponding to different floating-point formats:
  - `float` Single-precision floating-point
    suitable when the amount of precision is not critical
  - `double` Double-precision floating-point
    provides enough precision for most programs
  - `long double` Extended-precision floating-point
    rarely used
- The C standard does not 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 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***

$$number = sign \times fraction \times 2^{exponent}$$

- In single-precision format
  - the exponent is 8 bits long, while the fraction occupies 23 bits
- In double-precision format
  - the exponent is 11 bits long, while the fraction occupies 52 bits

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`)
- Floating constants can be written using `e` format in a variety of ways
- Valid ways of writing the number 57.0 include,

  `57.0 57.  57.0e0  57E0  5.7e1  5.7e+1 .57e2  570.e-1`
- A floating constant
  - must contain a decimal point *or* an exponent (or both)
- 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`)

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` :

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

- ***Note***: Use `l` only in a `scanf` format string, not in a `printf` format string
- In a `printf` format string, both the `e`, `f` and `g` conversions can be used to write either `float` or `double` values (**`C99`** legalize the use of `%lf`, `%le`, and `%lg`, in calls of `printf`, although the `l` has no effect.)

Character Types

- `char` is a numerical type that has the same properties as `int`
- `char` typically consumes a single byte
- Its main use is to represent characters
- 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 (from `0000000` to `1111111`)
- ASCII is often extended to a 256-character code that provides the characters necessary for Western European, Asian, and many African languages

Character Sets

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

```
char ch;

ch = 'a';    /* lower-case a has the value 97 */
ch = 'A';    /* upper-case A has the value 65 */
ch = '0';    /* zero  has the value 48 */
ch = ' ';    /* space has the value 32 */
```

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

Operations on Characters

- Working with characters in **C** is simple, because of the fact that:
  - *C treats characters as small integers*
- In ASCII, character codes range
  - from $(0000000)_2$, i.e., $(0)_{10}$
  - to $(1111111)_2$, i.e., $(127)_{10}$
- 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

```
int i;
i = 'a';        /* i is now 97   */

char ch;
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
- The following example converts a lower-case letter to upper case:

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

- Comparisons such as `ch >= 'a'` are done using the integer values of the characters involved
- The following example shows a `for` statement, which its control variable steps through all the upper-case letters:

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

Signed and Unsigned Characters

- The `char` type—like the `int` types—exists in both signed and unsigned versions
  - Signed characters have values between `-128` and `127`
  - Unsigned characters have values between `0` and `255`
- `C` allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;

char sch;   /* means it is a signed char */
```


Escape Sequences

- A character constant is usually one character enclosed in single quotes
- However, certain special characters—including the new-line character—can not be written in this way, because they are invisible (nonprinting) or because they can not be entered from the keyboard
- ***Escape sequences*** provide a way to represent these characters

Character Sequences

- A complete list of character escapes:

Name	*Escape Sequence*
Alert (bell)	`\a`
Backspace	`\b`
Form feed	`\f`
New line	`\n`
Carriage return	`\r`
Horizontal tab	`\t`
Vertical tab	`\v`
Backslash	`\\`
Single quote	`\'`
Double quote	`\"`

Character-Handling Functions

- Calling **C**'s `toupper` library function is a fast and portable way to convert case:

```
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:

```
#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:

```
char ch;

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

- In this case, `scanf` does not 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`

```
scanf(" %c", &ch);
```


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 one character

```
putchar(ch);
```

- `getchar` reads one character

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

- Like `scanf`, `getchar` does not skip white-space characters as it reads

Reading and Writing Characters Using **getchar** and **putchar**

- Using `getchar` and `putchar` (rather than `scanf` and `printf`) saves execution time
  - `getchar` and `putchar` are much simpler than `scanf` and `printf`, which are designed to read and write many kinds of data in a variety of formats
  - `getchar` and `putchar` are usually implemented as macros for additional speed

Reading and Writing Characters Using **getchar** and **putchar**

- Consider the following `scanf` loop that skips 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');
```

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

- Moving the call of `getchar` into the controlling expression allows us to condense the loop

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

- The `ch` variable isn't even needed; we can just compare the return value of `getchar` with the new-line character

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


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

- `getchar` is useful in loops that skip characters as well as loops that search for characters
- A statement that uses `getchar` to skip an unindefinite number of blank characters

```
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:

```
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:

```
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`

length.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;
}
```


Program: Determining the Length of a Message

- `length2.c` is a shorter program that eliminates the variable used to store the character read by `getchar`

length2.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 be
  - of the same size (the same number of bits) and
  - 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***
- The rules for performing implicit conversions are somewhat complex, primarily because `C` has so many different arithmetic types
- `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 (`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 (***how?***)
  - Converting a floating-point number to `int`, on the other hand,
    - causes the fractional part of the number to be lost
    - 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
  - the operand of the *narrower* type is converted 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 int` 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***
  - Perform *integral promotion* on ***both operands***, i.e., convert a character or short integer to type `int` (or to `unsigned int` in some cases)
  - *promote* the operand whose type is narrower

Type Conversion During Assignment

- 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; /* the value of c is converted to int */
f = i; /* the value of i is converted to float */
d = f; /* the value of 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 */
```

- It is a good idea to append the `f` suffix to a floating-point constant if it will be assigned to a `float` variable:

```
f = 3.14159f;
```

- Without the suffix, the constant `3.14159` would have type `double`

Casting

- C provides ***casts***
- A cast expression has the form

  (*type-name*) *expression*

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

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

Will it *correctly* work at all times?

- The difference between `f` and `(int) f` is the fractional part of `f`, which was dropped during the cast

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;
```


Casting

- Casts are sometimes necessary to avoid overflow:

```
long i;
int j = 1000;

i = j * j;    /* overflow may occur */
```

- Using a cast might avoid the problem:

```
i = (long) j * j;
```

- The statement

```
i = (long) (j * j);    /*** WRONG ***/
```

would not work, since the overflow would already have occurred by the time of the cast

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

On Obelix:

```
sizeof(char)  = 1     sizeof(unsigned char)  = 1
sizeof(short) = 2     sizeof(unsigned short) = 2
sizeof(int)   = 4     sizeof(unsigned int)   = 4
sizeof(long)  = 4     sizeof(unsigned long)  = 4

sizeof(float)       =  4
sizeof(double)      =  8
sizeof(long double) = 12
```

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:

```
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:

```
printf("Size of int: %zu\n", sizeof(int));
```