CSE105 – Structure Programming

COMMENTS, BASIC DATA TYPES

Comments

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

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

- Comments may appear almost anywhere in a program, either on separate lines or on the same lines as other program text.
- Comments may extend over more than one line.

```
/* Name: test.c
   Purpose: Prints a text line.
   Author: MSJ */
```

Comments

- *Warning:* Forgetting to terminate a comment may cause the compiler to ignore part of your program:

```
printf("My ");    /* forgot to close this comment...
printf("cat ");
printf("has ");   /* so it ends here */
printf("fleas");
```

```
#include <stdio.h>

int main(void)
{
    printf("My ");    /* forgot to close this comment...
    printf("cat ");
    printf("has ");   /* so it ends here */
    printf("fleas");
}
```

Comments

- Comments can also be written in the following way:

  `// This is a comment`

  - Ends automatically at the end of a line.
- Advantages of `//` comments:
  - Safer: there's no chance that an unterminated comment will accidentally consume part of a program.
  - Multiline comments stand out better.

```
#include <stdio.h>

int main(void)
{
    printf("My ");     // forgot to close this comment...
    printf("cat ");
    printf("has ");    // so it ends here
    printf("fleas");
}
```

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 `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
```

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

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 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

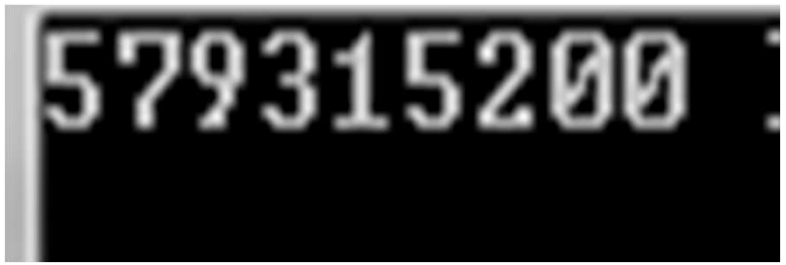
Integer Types

- 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

```c
#include <stdio.h>

int main(void)
{
    int X = 215327680000 ;
    printf("%d ",X);
    return 0;
}
```



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 Types – long long (in Linux)

- Two additional standard integer types, `long long int` and `unsigned long long int`.
- Both `long long` types are required to be at least 64 bits wide.
- The range of `long long int` values is typically -2^{63} (–9,223,372,036,854,775,808) to $2^{63} - 1$ (9,223,372,036,854,775,807).
- The range of `unsigned long long int` values is usually 0 to $2^{64} - 1$ (18,446,744,073,709,551,615).

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).

```
#include <stdio.h>

int main(void)
{
    int X = 32767 ;
    printf("%d ",X);
    return 0;
}
```

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`

```c
#include <stdio.h>

int main(void)
{
    int X = 0200 ;
    printf("%d ",X);
    return 0;
}
```

```
128
```

Integer Constants

- ***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  0xFf  0xFF  0Xff  0XfF  0XFf  0XFF`

```c
#include <stdio.h>

int main(void)
{
    int X = 0x200 ;
    printf("%d ",X);
    return 0;
}
```

```
512
```

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.

`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:

```
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.
- When reading or writing a ***long long*** integer, put the letters **ll** 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.***
- In single-precision format, the exponent is 8 bits long, while the fraction occupies 23 bits. The maximum value is approximately 3.40×10^{38}, with a precision of about 6 decimal digits.

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.

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`:

```
double d;

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

- 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.

Character Sets

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

Character Sets

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

```
char ch;

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

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

Operations on Characters

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

Operations on Characters

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

```
char ch;
int i;

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

Operations on Characters

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

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

- Comparisons such as `ch>=‘a’` are done using the integer values of the characters involved.

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++) …
```

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.
- C allows the use of the words `signed` and `unsigned` to modify `char`:

```
signed char sch;
unsigned char uch;
```

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 */
```

- `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`:

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

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

- 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:

```
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:

```
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`**

- 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.
  - They are usually implemented as macros for additional speed.

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');
```

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

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;
}
```