THE FUNDAMENTAL DATA TYPES

Declarations, Expressions, and Assignments

- Variables and constants are the objects that a prog. manipulates.
- All variables must be declared before they can be used.

```
#include <stdio.h>
int main(void)
{
   int a, b, c;                          /*declaration*/
   float x, y = 3.3, z = -7.7;           /*declaration with initialization*/

   printf("Input two integers:  "); /*function call*/
   scanf("%d%d",&b, &c);                 /*function call*/
   a = b + c;                            /*assignment*/
   x = y + z;                            /*assignment*/
}
```

Declarations, Expressions, and Assignments

- <u>Declarations</u>
  - associate a type with each variable declared
  - This tells the compiler to set aside an appropriate amount of memory space to hold values associated with variables.
  - This also enables the compiler to instruct the machine to perform specified operation correctly.

    b + c (integer addition)

    y + z (real number addition)

Declarations, Expressions, and Assignments

- <u>Expressions</u>
  - Meaningful combinations of constants, variables, operators, and function calls.
  - A constant, variable, or function call itself is also an expression

    a+b

    sqrt(7.333)

    5.0 * x – tan(9.0 / x)
  - Most expressions have a value.

    i = 7 assignment expression

<Examples of statements>

```
i = 7;
printf("The plot thickens!\n");
3.777;
a + b ;
```

Perfectly legal, but they are not useful

Declarations, Expressions, and Assignments

- <u>Assignments</u>

  variable = expr ; assignment statement

<Mathematical equation>	<Assignment expression>
x + 2 =0	x + 2 = 0 /*wrong*/
x = x + 1 (meaningless)	x = x + 1

!! Although they look alike, the assignment operator in C and the equal sign in mathematics are NOT COMPARABLE

The Fundamental Data Types

Fundamentl data types: long form		
char	signed char	unsigned char
signed short int	signed int	signed long int
unsigned short int	unsigned int	unsigned long int
float	double	long double

Fundamentl data types		
char	signed char	unsigned char
short	int	long
unsigned short	unsigned	unsigned long
float	double	long double

Fundamentl types grouped by functionality			
Integral types	char	signed char	unsigned char
	short	int	long
	unsigned short	unsigned	unsigned long
Floating types	float	double	long double

Characters and the Data Type **char**

- type **char**
  - A variable of type **char** can be used to hold small integer values.
  - 1 byte (8 bits) in memory space
    - 2^8, or 256, distinct values
      - including lower- and uppercase letters, digits, punctuation, and special char.s such as % and +
      - including white space blank, tab, and newline

Characters and the Data Type **char**

- Most machines use either ASCII or EBCDIC character codes to represent a character in bits.
- ASCII character code
  - a character encoding-scheme
  - A character constant has its corresponding integer value.

    ‘a’ (97) ‘b’ (98) ‘c’ (99) …

    ‘A’ (65) ‘B’ (66) ‘C’ (67) …

    ‘0’ (48) ‘1’ (49) ‘2’ (50) …

    ‘&’ (38) ‘*’ (42) ‘+’ (43) …
  - No particular relationship btwn the value of the character constant representing a digit and the digit’s intrinsic integer value. **‘2’ ≠ 2**

Characters and the Data Type **char**

- Nonprinting and hard-to-print characters require an escape sequence.
- \ (backslash character)
  - an escape character
  - is used to escape the usual meaning of the character that follows it.

Special Characters		
Name of character	Written in C	Integer value
alert	\a	7
backslash	\\	92
double quote	\"	34
newline	\n	10
null character	\0	0
single quote	\'	39

```
printf("%c", '\a');                    or   putchar('\a');
printf("\"abc\"");                     /* "abc" is printed */
printf("%cabc%c", '\", '\");           /* 'abc' is printed */
```

Characters and the Data Type **char**

- Characters are treated as small integers

```
char c = 'a';
printf("%c", c);   /* a is printed */
printf("%d", c);   /* 97 is printed */
printf("%c%c%c", c, c+1, c+2);    /* abc is printed */

char c;
int i;
for ( i = 'a'; i<= 'z'; ++i )
   printf ("%c", i);                /* abc…z is printed */
for ( c = '0'; i<= '9'; ++c )
   printf ("%d ", c);               /* 48 49 … 57 is printed */
```

Characters and the Data Type **char**

char c = 'a';

- **c** is stored in memory in 1 byte as 01100001 (97)

- The type **char** holds 256 distinct values
  - **singed char** : -128 ~ 127
  - **unsigned char** : 0 ~ 255

The Data Type **int**

- type **int**
  - the principal working type of the C language
  - integer values
  - stored in either 2 bytes (=16 bits) or in 4 bytes (=32 bits)
    - 64-bit OS: 4 bytes or 8 bytes
  - holds 2^{32} distinct states (in case of 4 bytes)

    $-2^{31}, -2^{31}+1, \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots, 2^{31}-1$

(-2,147,483,648) (2,147,483,647)

```
#define BIG 2000000000        /* 2 billion */
int main(void)
{
   int a, b = BIG, c= BIG;
   a = b + c; /* out of range? */
   ......
```

integer overflow !!

The Integral Types **short**, **long**, and **unsigned**

- The type **int** is "natural" or "usual" type for working with integers
- The other integral types, such as **char**, **short**, and **long**, are intended for more specialized use.
  - **short** (2 bytes)
    - when the storage is of concern
  - **long** (4 bytes or 8 bytes)
    - when large integer values are needed
  - **short** $\leq$ **int** $\leq$ **long**

The Integral Types **short**, **long**, and **unsigned**

- Type **int** and **unsigned** are stored in a machine WORD.
  - 2 bytes, 4 bytes (, or 8 bytes)

unsigned u;

$0 \leq u \leq 2^{wordsize}-1$

$0 \leq u \leq 2^{32}-1$ (+4294967295 , 4 billion)

- Suffixes can be appended to an integer constant to specify its type.

Combining long and unsigned		
Suffix	Type	Example
u or U	unsigned	37U
l or L	long	37L
ul or UL	unsigned long	37UL

The Floating Types

- 3 floating types
  - **float**, **double**, **long double**
  - holds real values such as 0.001, 2.0, and 3.14159
  - A suffix appended to a floating constant to specify its type

Combining long and unsigned		
Suffix	Type	Example
f or F	float	3.7F
l or L	long double	3.7L

  - The working floating type in C is **double**.
    - the constants 1.0 and 2.0 : **double**
    - the constant 3 : **int**

The Floating Types

- Floating constant
  - decimal notation: **123456.7**
  - exponential notation

    1.234567e5

    = 1.234567×10^5

    = 123456.7 (decimal point shifted five places to the RIGHT)

    1.234567e-3

    = 1.234567×10^{-3}

    = 0.001234567 (decimal point shifted three places to the LEFT)

Floating constant parts for 333.77777e-22		
Integer	Fraction	Exponent
333	77777	e-22

The Floating Types

- Floating constant
  - may contain an integer part, a decimal point, a fractional part, and an exponential part.
  - MUST contain either a decimal point or an exponential part or both.
  - If a decimal point is present, either an integer part or fractional part or both MUST be present.

<Examples>	<NOT Examples>
3.14159	3.14,159
314.159e-2	314159
0e0 (⇔ 0.0)	.e0
1.	-3.14159 (floating constant expr.)

The Floating Types

- Possible values of a floating type
  - <u>Precision</u>
    - the # of significant decimal places that a floating value carries.
  - <u>Range</u>
    - The limits of the largest and smallest positive floating values that can be represented in a variable of that type
- type **float**
  - stored in 4 bytes
  - Precision of 6 significant figures & Range of 10^{-38} to 10^{38}

    $0.d_1d_2d_3d_4d_5d_6 \times 10^n$

    , where each d_i is a decimal digit, the first digit, d_1, is positive (i.e. non-zero), and $-38 \leq n \leq 38$

The Floating Types

- type **double**
  - stored in 8 bytes
  - Precision of 15 significant figures & Range of 10^{-308} to 10^{308}

    $0.d_1d_2 \ldots d_{15} \times 10^n$

    , where each d_i is a decimal digit, the first digit, d_1, is positive, and $-308 \leq n \leq 308$

x = 123.45123451234512345; **/* 20 significant digits */**

$0.123451234512345 \times 10^3$ (15 significant digits)

(1) NOT all real numbers are representable

(2) floating arithmetic operations need not be exact

The Use of typedef

- **typedef**
  - allows the programmer to explicitly associate a type with an identifier

```
typedef char              uppercase;
typedef  int              INCHES, FEET;
typedef unsigned long     size_t;

int main(void)
{
    uppercase u;
    INCHES   length, width;
    …
}
```

(1) abbreviating long declarations

(2) having type names that reflect the intended use

The sizeof Operator

- **sizeof**
  - a unary operator to find the # of bytes needed to store an object

    sizeof(object)

    : **object** can be a type such as **int** or **float**, or an expression such as **a+b.**

```
/* Compute the size of some fundamental types. */
#include <stdio.h>
int main(void)
{
   printf("The size of some fundamental types is computed.\n\n");
   printf("          char:%3u byte \n", sizeof(char));
   printf("         short:%3u bytes\n", sizeof(short));
   printf("           int:%3u bytes\n", sizeof(int));
   printf("         float:%3u bytes\n", sizeof(float));
   printf("        double:%3u bytes\n", sizeof(double));
}
```

The **sizeof** Operator

- **sizeof**

  sizeof(char) = 1

  sizeof(char) $<$ sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long)

  sizeof(signed) = sizeof(unsigned) = sizeof(int)

  sizeof(float) $\leq$ sizeof(double) $\leq$ sizeof(long double)

  - **sizeof(…)** looks that it is a function, but it is not. An Operator.
  - The type returned by the operator **sizeof** is typically **unsigned**.

The use of **getchar()** and **putchar()**

- **getchar(), putchar()**
  - macros defined in stdio.h
  - **getchar()**
    - reads a character from the keyboard
  - **putchar()**
    - prints a character on the screen

```
#include <stdio.h>
int main(void)
{
    int c;
    while ( (c = getchar()) != EOF) {
        putchar(c);
        putchar(c);
    }
    return 0;
}
```

The use of **getchar()** and **putchar()**

- the identifier, **EOF**

  #define EOF (-1) **,in the header file, stdio.h**

  - "end-of-file"
  - What is actually used to signal an end-of-file mark is system-dependent.
  - The int value -1 is often used.

int c;

c is an **int**, it can hold all possible char. values as well as the special value **EOF**.

(c = getchar()) != EOF;

c = getchar();

The subexpression **c = getchar()** gets a value from the keyboard and and assigns it the variable **c**,

,and the value of the subexpression takes on that value as well.

c = getchar() != EOF ⇔ **c = (getchar() != EOF)**

The use of **getchar()** and **putchar()**

'a' ⇔ 97

'a'+1 ⇔ 'b'

'z' – 'a' ⇔ 'Z' – 'A' ⇔ 25

A lowercase letter, **c** :

c + 'A' – 'a' has a value of the corresponding uppercase letter.

```
#include <stdio.h>
int main(void)
{
    int c;
    while ( (c = getchar()) != EOF)
        if ( c>= 'a' && c<= 'z')
            putchar(c + 'A' – 'a');
        else
            putchar(c);
     return 0;
}
```

General Conversion Rules

- For binary operations with operands f different types, the "**lower**" type is **promoted** to the "**higher**" type before operation proceeds.

- For assignment operations, the value of the **right side** is converted to the type of the **left**, which is the type of the result.

Informal Conversion Rules

If there is no unsigned operands,

- If either operand is long double, convert the other to long double
- Otherwise, if either operand is double, convert the other to double
- Otherwise, if either operand is float, convert the other to float
- Otherwise, convert char and short to int
- Then, if either operand is long, convert the other to long

Usual Arithmetic Conversions

- If either operand is long double, convert the other to long double
- Otherwise, if either operand is double, convert the other to double
- Otherwise, if either operand is float, convert the other to float
- Otherwise, the **integral promotions** are performed on both operands; then, if either operand is unsigned long int, the other is converted to unsigned long int
- Otherwise, if one operand is **long** int and the other is **unsigned** int, the effect is system-dependent
- Otherwise, if one operand is long, convert the other to long
- Otherwise, if either operand is unsigned int, the other is converted to unsigned int
- Otherwise, both operands have type int.

Usual Arithmetic Conversions

- occurs when the operands of a binary operator are evaluated.

```
int i = 3;
float f = 3.0, g;
```

i + f (the type **float**)

long double → **double** → **float** → **Integral Promotion**

unsigned long → [**long** *op* **unsigned**] → **long** → **unsigned** → **int**

long (if long is long enough)
or
unsigned long (otherwise)

- ✓ **Automatic conversion**
- ✓ **Implicit conversion**
- ✓ **Widening**
- ✓ **Promotion**
- ✓ **coercion**

Integral Promotions

- A char, a short integer, or an integer bit-field, all either signed or not, or an object of enumeration type, may be used in an expression whenever an integer may be used.
- If all the values of the original type in an expression can be represented by an **int**, then the value is converted to an **int**; otherwise the value is converted to **unsigned int**.

```
short x, y;
x + y (the type int, not short)
```

```
char c = 'A';
printf("%c\n", c);     /* the type of expr. C is int, not char */
```

Informal Conversion Rules

- Comparisons between **signed** and **unsigned** values are **machine-dependent** because they depend on the sizes of the various integer types

- if one operand is **long** int and the other is **unsigned int**, the effect depends on whether a long int can represent all values of an unsigned int in the system;
  - if so, the unsigned int operand is converted to **long** int;
  - if not, both are converted to **unsigned long** int.

Informal Conversion Rules

- **float** is not automatically converted to **double**
  - **float** is for saving storage in large arrays, or less often, to save time on machines where double-precision arithmetic is particularly expensive.
- **double** to **float** conversion is implementation-dependent (rounded or truncated)
- **float** to **int** causes **truncation** of any fractional part

Informal Conversion Rules

- Longer integers are converted to shorted ones or chars by **dropping** the excess **high-order** bits (e.g. the value of c is unchanged)

```
int i;
char c;

i=c;
c=i;
```

Conversions and Casts

Declarations			
char c;	short s;	int i;	
long l;	unsigned u;	unsigned long ul;	
float f;	double d;	long double ld;	
Expression	Type	Expression	Type
c - s / i	int	u * 7 - i	unsigned
u * 2.0 - i	double	f * 7 - i	float
c + 3	int	7 * s * ul	unsigned long
c + 5.0	double	ld + c	long double
d + s	double	u - ul	unsigned long
2 * i / l	long	u - l	system-dependent

d = i; **Widening**

- The value of **i** is converted to a double and then assigned to **d**

i = d; **Narrowing**

- Loss of Information. The fraction part of d will be discarded.

Conversions and Casts

- Casts
  - Explicit conversions

(double) i

- casts, or converts, the value of **i** so that the expr. has type double
- The variable **i** itself remains unchanged.

<Examples>

```
l = (long) ('A' + 1.0);
f = (float) ((int)d + 1);
d = (double) i / 3;
```

<NOT Examples>

```
(double) x = 77; /* equivalent to ((double) x) = 77, Error*/
```

- The cast operator (*type*) is an unary operator.

(float) i + 3 ⇔ **((float) i) + 3**

Hexadecimal and Octal Constants

Hexadecimal digits and their corresponding decimal values										
Hexadecimal digit:	0	1	…	9	A	B	C	D	E	F
Decimal value:	0	1	…	9	10	11	12	13	14	15

Hexadecimal Constants:

2A $\Leftrightarrow$ $2 \times 16^1 + A \times 16^0 = 2 \times 16^1 + 10 \times 16^0 = 42$

5B3 $\Leftrightarrow$ $5 \times 16^2 + B \times 16^1 + 3 \times 16^0 = 5 \times 16^2 + 11 \times 16^1 + 3 \times 16^0 = 1459$

Octal Constants:

75301

$\Leftrightarrow 7 \times 8^4 + 5 \times 8^3 + 3 \times 8^2 + 0 \times 8^1 + 1$

Decimal	Binary	Hexadecimal	Octal
0	00000000	00	000
1	00000001	01	001
2	00000010	02	002
3	00000011	03	003
.....			
31	00011111	1F	037
32	00100000	20	040
.....			
188	10111100	BC	274
.....			
254	11111110	FE	376
255	11111111	FF	377

Hexadecimal and Octal Constants

```
#include <stdio.h>
int main(void)
{
    printf(%d   %x   %o\n", 19, 19, 19);          /* 19  13  23 */
    printf(%d   %x   %o\n", 0x1c, 0x1c, 0x1c);    /* 28  1c  34 */
    printf(%d   %x   %o\n", 017, 017, 017);       /* 15   f  17 */
    printf(%d\n", 11 + 0x11 + 011);               /* 37 */
    printf(%x\n", 2097151);                       /* 1fffff */
    printf(%d\n", 0x1FfFFf);                      /* 2097151 */
    return 0;
}
```