CPNM Lecture 2 - Basic Data Types and Formatted Input/Output

Mridul Sankar Barik

Jadavpur University

November 13, 2022

Basic Data Types I

- Numeric: integer and floating point
  - Value of an integer type are whole numbers, while values of a floating type call have a fractional part
- Character: Used to store a single character

Integer Types I

- Integers can be `signed` and `unsigned`
- The leftmost bit of a signed integer (known as the sign bit) is 0 if the number is positive or zero, 1 if it is negative
- An integer with no sign bit (the leftmost bit is considered part of the number's magnitude) is said to be `unsigned`
- By default integer variables in C are `signed`
- C's integer types come in different sizes. The `int` type is usually 32 bits (16 bits on older CPUs)
- **long** integers: greater range than normal integer
- **short** integers: shorter range and lesser storage

Integer Types II

- Six variations

```
short int
unsigned short int
int
unsigned int
long int
unsigned long int
```

- The order of the type specifiers doesn't matter
- Type specifiers can be abbreviated by dropping the word `int`
- The range of values represented by each of the six integer types varies from one machine to another
- 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 Types III

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

```
#include<stdio.h>
#include<limits.h>

int main() {

   printf("The number of bits in a byte %d\n", CHAR_BIT);

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);
   printf("The maximum value of UNSIGNED SHORT INT = %d\n", USHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);
   printf("The maximum value of UNSIGNED INT = %u\n", UINT_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);
   printf("The maximum value of UNSIGNED LONG = %lu\n", ULONG_MAX);

   return(0);
}
```

Integer Types IV

- C99 provides two additional standard integer types, `long long int` and `unsigned long long int` $\Rightarrow$ at least 64 bits wide
- **Integer Constants**: numbers that appear in the text of a program, not numbers that are read, written, or computed
- C allows integer constants to be written in decimal (base 10), octal (base 8), or hexadecimal (base 16).
  - **Decimal constants**: contains digits between 0 and 9, but must not begin with a zero; ex: 15, 255, 32767
  - **Octal constants**: contains only digits between 0 and 7, and must begin with a zero: ex; 017, 0377, 077777
  - **Hexadecimal constants**: contain digits between 0 and 9 and letters between a and f and always begin with Ox; ex: Oxf, Oxff, Oxfff
- Example:

Integer Types V

```
#include<stdio.h>

int main(void){

    int i = 32767;
    //int i = 0x7fff;
    //int i = 077777;

    printf("In decimal = %d\n", i);
    printf("In Hexadecimal = %x\n", i);
    printf("In Octal = %o\n", i);

    return(0);
}
```

Floating Point Type I

- Allows storage of numbers with digits after the decimal point, or numbers that are too large or small
- C provides three floating types: Corresponding to different floating-point format:
  `float` - **Single-precision** floating-point
  `double` - **Double-precision** floating-point
  `long double` - **Extended-precision** floating-point
- Most computers follow the specifications in IEEE Standard 754

Type	Smallest Positive Value	Largest Value	Precision
float	$1.17549 times 10^{-38}$	3.40282×10^{38}	6 digits
double	$2.22507 times 10^{-38}$	1.79769×10^{38}	15 digits

- Example: Verify the precision

Floating Point Type II

```
#include<stdio.h>

int main(void){

    float f1 = 12.0000123;
    float f2 = 12.0000128;

    double d1 = 12.000000000000123;
    double d2 = 12.000000000000124;

    printf("f1=%.15f f2=%.15f \n", f1, f2);
/* prints f1=12.000012397766113 f2=12.000012397766113*/
    printf("d1=%.15lf d2=%.15lf \n", d1, d2);
/* prints d1=12.000000000000012 d2=12.000000000000012*/
}
```

- Macros that define the characteristics of floating types are in the <float.h> header

Floating Point Type III

```
#include <stdio.h>
#include <float.h>

int main (){
   printf("The maximum value of float = %.10e\n", FLT_MAX);
   printf("The minimum value of float = %.10e\n", FLT_MIN);

   printf("The number of digits in the mantissa part of \
        a float = %d\n", FLT_MANT_DIG);
}
```

- **Floating Point Constants**
- Different ways of writing the number 57.0
  57.0, 57., 57.0e0, 57EO, 5.7e1, 5.7e+1, .57e2, 570.e-1
- By default, floating point constants are stored as double-precision numbers
- Reading and Writing Floating-Point Numbers: conversion specifications %e, %f, and %g are used for reading and writing single-precision floating-point numbers

Floating Point Type IV

- %f prints the corresponding number as a decimal floating point number, e.g. 321.65.
- %e prints the number in scientific notation or exponential format, e.g. 3.2165e+2.
- %g prints the number in the shortest of these two representations

```
#include<stdio.h>

int main(void){

    float a = 3214.65;
    float b = 160000000;

    printf("%f\n", a); /* prints 3214.649902*/
    printf("%e\n", a); /* prints 3.214650e+03*/
    printf("%g\n", a); /* prints 3214.65*/

    printf("%f\n", b); /* prints 160000000.000000*/
    printf("%e\n", b); /* prints 1.600000e+08*/
```

Floating Point Type V

```
    printf("%g\n", b); /* prints 1.6e+08*/

    return(0);
}
```

- 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:

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

Character Types I

- The values of type char can vary from one computer to another
- Most popular character set is ASCII (American Standard Code for Information Interchange), a 7-bit code capable of representing 128 characters
- A character constant is one character in single quotes
- ```
  char ch;
  ch = 'a';
  ch = 'A';
  ch = '0';
  ch = ' ';
  ```
- C treats characters as small integers
  ```

Character Types II

```
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'*/
```

- Characters can be compared, just as numbers are
- **Signed and Unsigned Characters:** Signed characters normally have values between -128 and 127, while unsigned characters have values between 0 and 255.
- **Escape Sequences:** Special notation to print certain special characters, including the new-line character which cannot be written as they are invisible

ASCII

- American Standard Code for Information Interchange → developed by the American National Standards Institute (ANSI)
- Each alphabetic, numeric, or special character is represented with a 7-bit binary number

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII

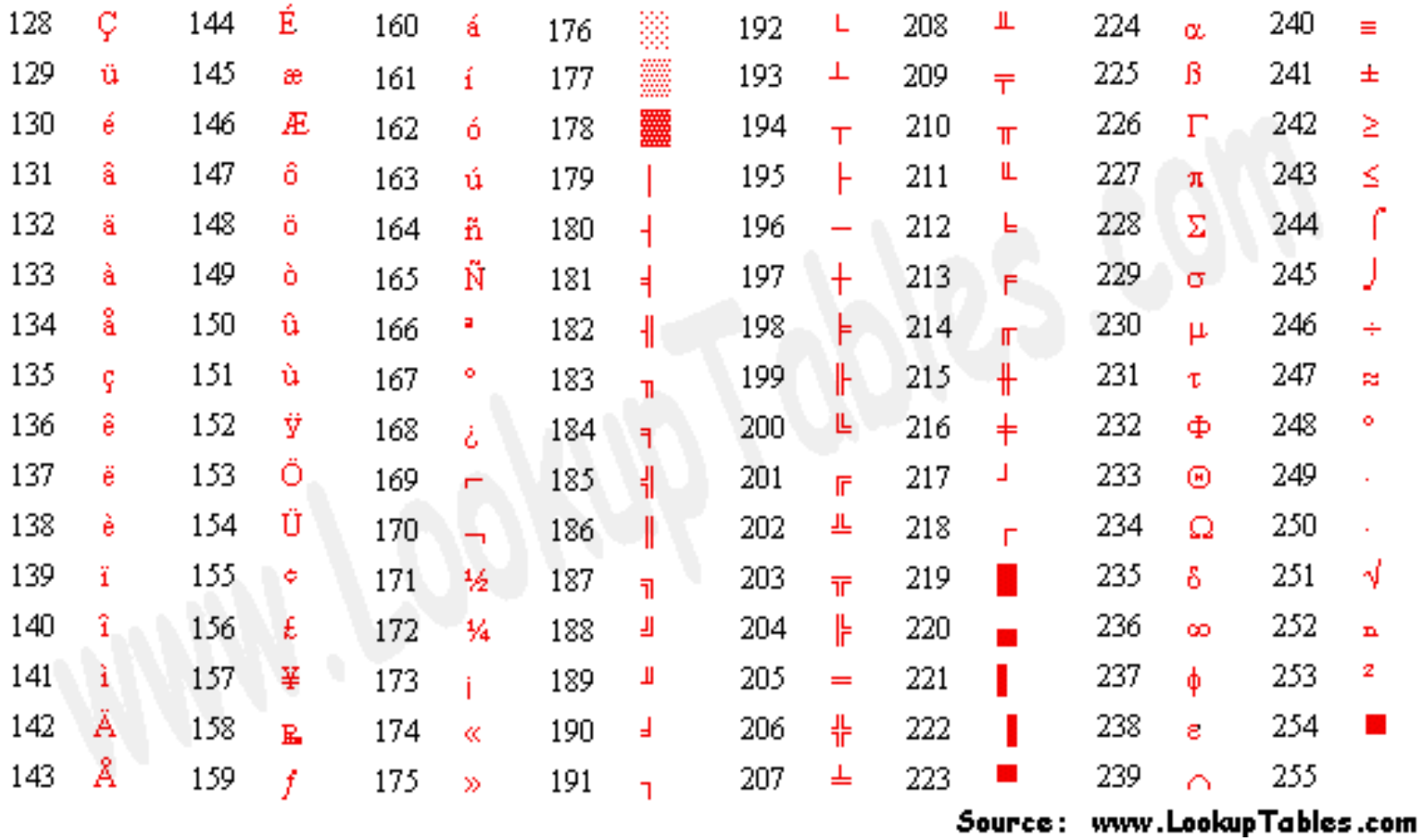

Figure 2: Extended ASCII Character Set

Formatted Output I

- `printf` displays contents of a string known as the format string
- Format string may contain both ordinary characters and *conversion specifications* that begin with the % character
- The information after % character specifies how the value is converted from its internal (binary) form to printed form (characters)
- %d specifies `printf` to convert an `int` value from binary to a string of decimal digits; %f does the same for a `float` value.
- `gcc` does not check whether number of conversion specifications in a format string matches the number of output items
- `gcc` does not check whether a conversion specification is appropriate for the item being printed

Formatted Output II

- General form of conversion specification is `%m.pX` or `%-m.pX`, where `m` and `p` are integer constants and `X` is a letter.
- `m` represents minimum field width
  - If the value to be printed requires fewer than `m` characters, the value is right justified within the field
  - If the value to be printed requires more than `m` characters, the field width automatically expands to the necessary size
  - Putting a minus sign in front of `m` causes left justification
- Meaning of `p` or precision is dependent on the choice of `X`
- If `printf` encounters two consecutive % characters in a format string, it prints a single % character

Sample Program

```
#include<stdio.h>
int main(void){
int i;
float x;
i = 40;
x = 839.21;
printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
/* Output: |40|   40|40   |  040| */
printf("|%10.3f|%10.3e|%-10g|", x, x, x);
/* Output: |   839.210| 8.392e+02|839.21    | */
return(0);
}
```

Escape Sequence

- Enable strings to contain characters that would otherwise cause problems for the compiler
- An escape character is a character which invokes an alternative interpretation on subsequent characters in a character sequence
- Alert (bell) \a
  Backspace \b
  New line \n
  Horizontal tab \t
  etc.

Formatted Input I

- `scanf` library function reads input according to a particular format
- A `scanf` format string may contain both ordinary characters and conversion specifiers
- Programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable
- & symbol must precede each variable in a `scanf` call $\Rightarrow$ produces unpredictable results if omitted
- `scanf` "peeks" at the final new-line character without actually reading it. This new-line will be the first character read by the next call of `scanf`.
- A white-space character in a format string matches any number of white-space characters in the input, including none.

Formatted Input II

- When `scanf` encounters a non white-space character in a format string, `scanf` compares it with the next input character. If the two characters match, `scanf` discards the input character and continues processing the format string. If the characters don't match, `scanf` puts the offending character back into the input then aborts without further processing the format string or reading characters from the input.
- Example

```
#include <stdio.h>
int main (void){
  int num1, denom1, num2, denom2, result_num, result_denom;
  printf("Enter first fraction: ");
  scanf("%d/%d". &num1, &denom1);
  printf("Enter second fraction: ");
  scanf ("%d/%d", &num2, &denom2);
  result_num =   num1 * denom2 + num2 * denom1;
  result_denom = denom1 * denom2;
```

Formatted Input III

```
  printf("The sum is %d/\d\n", result_num, result_denom);
  return 0;
}
```

- In a `scanf` format string %d can only match an integer written in decimal (base 10) while %i can match an integer expressed in octal (base 8), decimal, or hexadecimal numbers (base 16).
- If an input number has a 0 prefix (as in 056), %i treats it as an octal number; if it has a 0x or 0X prefix (as in Ox56) %i treat it as a hex number.
- When reading or writing an unsigned integer, use the letter u, o, or x instead of d in the conversion specification.

Formatted Input IV

```
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 */
```

- 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);
```

Reading and Writing Characters using getchar and putchar

- `putchar` writes a single character:
- `getchar` reads one character, which it returns

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

- much simpler than `scanf` and printf

Type Conversion I

- 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
- C allows the basic types to be mixed in expressions
- If we add a 16-bit `short` and a 32-bit `int`, the compiler will convert the `short` value to 32-bit `int`
- If we add an `int` and a `float`, the compiler will convert the `int` to `float` format
- **Implicit conversion**: C compiler handles conversions automatically
- **Explicit conversion**: programmer performs conversion using cast operators
- Example

Type Conversion II

```
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*/
```

Type Conversion III

- Conversion During Assignment: expression on the right side of the assignment is converted to the type of the variable on the left side
- Explicit Type Casting: `( type_name ) expression`

The sizeof Operator I

- Returns how much memory (in bytes) is required to store values of a particular type
- Format: `sizeof ( type_name )`
- Example:

```
printf("Size of char = %d\n", sizeof(char));
printf("Size of int = %d\n", sizeof(int));
printf("Size of long = %d\n", sizeof(long));
printf("Size of long long = %d\n", sizeof(long long));
printf("Size of float = %d\n", sizeof(float));
printf("Size of double = %d\n", sizeof(double));
printf("Size of long double = %d\n", sizeof(long double));
```

Output: In a 32 bit machine

The sizeof Operator II

```
Size of char = 1
Size of int = 4
Size of long = 4
Size of long long = 8
Size of float = 4
Size of double = 8
Size of long double = 12
```

Output: In a 64 bit machine

```
Size of char = 1
Size of int = 4
Size of long = 8
Size of long long = 8
Size of float = 4
Size of double = 8
Size of long double = 16
```