CSc 28
Discrete Structures

Chapter 7
ASCII & Integer

Herbert G. Mayer, CSU CSC

Status 2/1/2021

Syllabus

- a2i Conversion
- Account for Sign
- Integer Overflow
- MaxInt
- References

Computers Think?

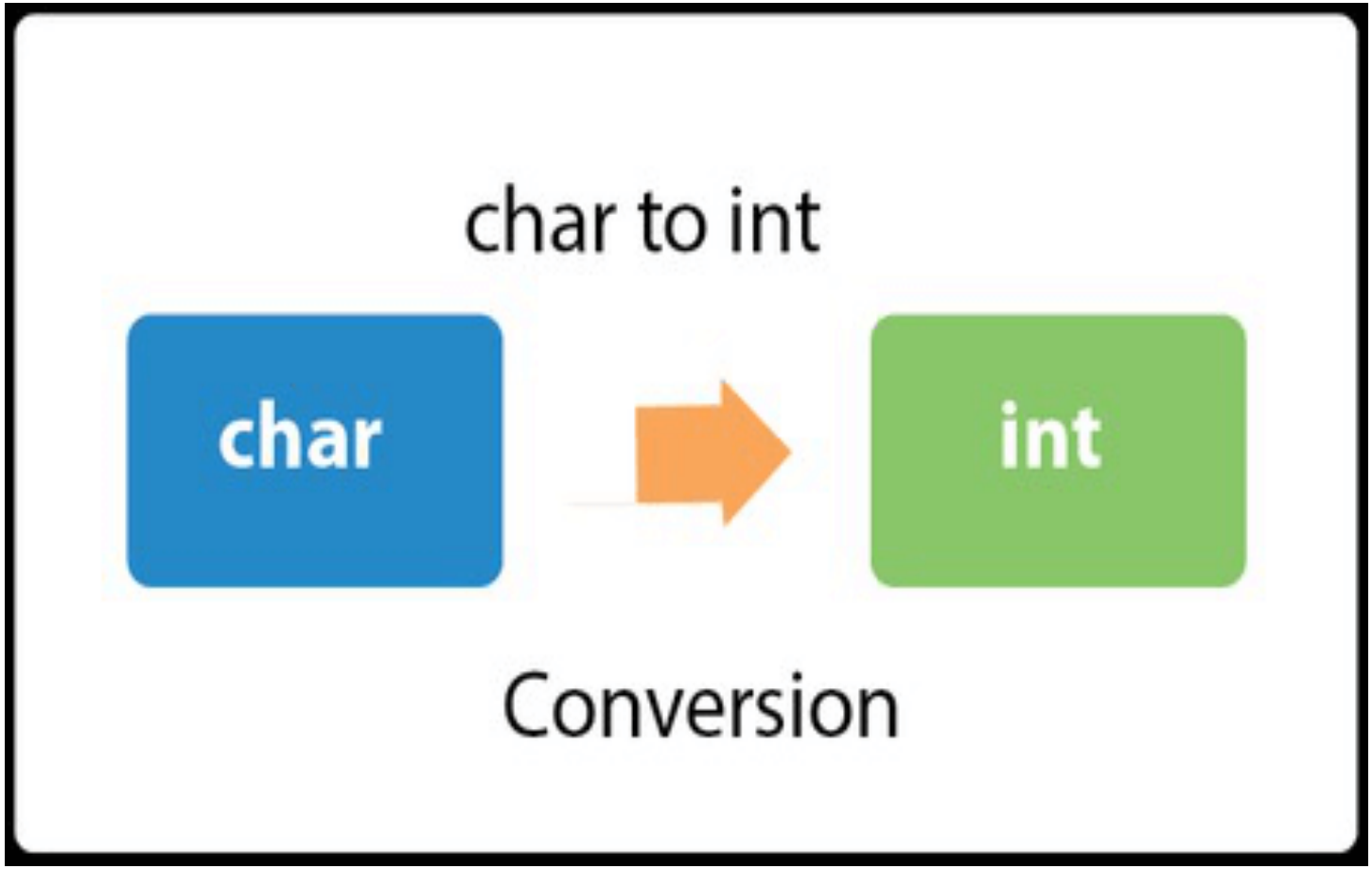
- Computers can . . . . create pictures
  - They cear sound, understand language, and reply
- Computers can compute, draw graphs, read maps
  - Read, write, scan, print, cause trouble . . .
- Are they different in mental capacity from humans?
- Hal in the movie 2001 "thought" so ☺
- But computers really just can store and manipulate bits, hence also characters and numbers
  - And compute new values
  - Yet very swiftly so!
- Here we explore how characters and integers can be used for quite some amazing results

Computers Think?

Computers can’t think.
They only think they can!

Computers Think?

a2i() Conversion



a2i() Conversion

- Reading one input char at a time, assuming and verifying: character is a decimal digit in range '0'..'9'
- Convert this ASCII string of decimal digits to its corresponding integer value
- For example: Incoming string "123" is converted to the integer one-hundred-and-twenty-three, i.e. 123_{10}
- Be aware that this input is indeed a sequence of ASCII characters, not an integer value
- We must convert ASCII character 'digit 5' for example to integer value 5: subtract from character '5' the character '0'; difference will be 5
  - Only in ASCAII, not in EBCDIC!
- Int value represented by character '5' is: '5' – '0' = 5_{10}

a2i() Conversion

```
char c = ' ';          // global char 'c' to craft token
. . .
int a2i( void )        // craft decimal number for chars
{ // a2i               // return int number, at least 0
    int number = 0;    // to be built from digits 0..9
    while( (
      ( c = getchar() ) >= '0' ) && ( c <= '9' ) ) {
        // assumes no overflow condition occurs
        number = number * 10 + ( c - '0' );
    } //end while     // no more digits?
    return number;
} //end a2i

int main()
{ // main
   cout << "Enter a decimal number: " << endl;
   cout << "The number was:" << a2i() << endl;
   return 0;
} //end main
```

a2i() Conversion

- Assume a sequence of decimal digits to be read
- However, if no decimal digit is entered, the correct result of 0 must still be generated
- And it shall be 0, due to initialization of number; see "int number = 0;"
- Crucial test is: `( c = getchar() ) >= '0' ...`
- Here a side-effect happens: A character is read from standard in, and it is tested
- While valid, input is read and processed one digit at a time
- Current number is "shifted left" by 1 decimal position (AKA is multiplied by 10) and new digit's decimal value is added, not the ASCII value

ASCII Characters

Signed a2i() Conversion

- Before reading one decimal digit at a time, do:
  - Check for optional negative sign '-'
  - Or for a redundant positive sign '+'
  - Do not allow multiple signs; disallow + + or + - or - - etc.
- An input character '-' will change the coming value, by inverting the sign, rendering the number negative
- State of '-' has to be remembered; see boolean neg
- Case of leading '+' can be silently skipped
- Once the sign is handled, proceed as before with the decimal digits that constitute actual decimal number
- But caution, the + and - integer ranges are not symmetric on two's complement architecture!

Signed a2i() Conversion

```
bool is_digit( char c )          // this c is local
{ // is_digit                    // formals are local
    return ( c >= '0' ) && ( c <= '9' );
} //end is_digit

int a2i0( void )                 // use global c
{ // a2i0
    int  number =  0;
    bool neg    =  false;        // init. necessary? ☺
    char c = getchar();          // read first char
    if( '-' == c ) {
      neg = true;
      c = getchar();             // skip '-'
    }else if( c == '+' ) {
      c = getchar();             // skip '+'
    } //end if
    while( is_digit( c ) ) {
      // assume: no overflow
      number = number * 10 + ( c - '0' );
      c = getchar();             // skip current digit
    } //end while
   // c is known to be not a digit '0'..'9'
    return neg ? -number : number;  // careful ☺
} //end a2i0
```

Signed a2i() Conversion

- State variable neg is used to determine at the end, whether sign inversion must happen
- Use so-called conditional expression:

```
return neg ? -number : number;
```

- For initial state neg = false, the decimal value computed is returned
- But if neg is true, the sign is inverted, accomplished in single conditional expression
- Note separation of the 2 options via the : and ? operators! Like the then- and else-clauses in an If Statement!
- Inherited from Algol68
- Verified that neg must be initialized?

Overflow in a2i()

Overflow in a2i()

- MaxInt on 16-bit architecture is 32767 AKA +32767
- Delicate programming matter, analyzed by sample of a fictitious 16-bit architecture (did exist in the past!)
- On two's-complement 16-bit architecture, largest negative value MinInt is: -32768
  - MinInt 64-bit architecture is: -9,223,372,036,854,775,808
- Scanning must be handled, without overflow actually occurring: so SW must probe the growing integer value, before next a multiply by 10 will occur!
- Borderline case is MAXby10 = 3276 on 16-bit arch
  - On 64-bit architecture equivalently: 922,337,203,685,477,580
- Also track, whether scanning a negative literal:
  - For negative literal, largest next digit for current 3276 is '8'
  - For positive literal, the largest next/last decimal digit is '7'

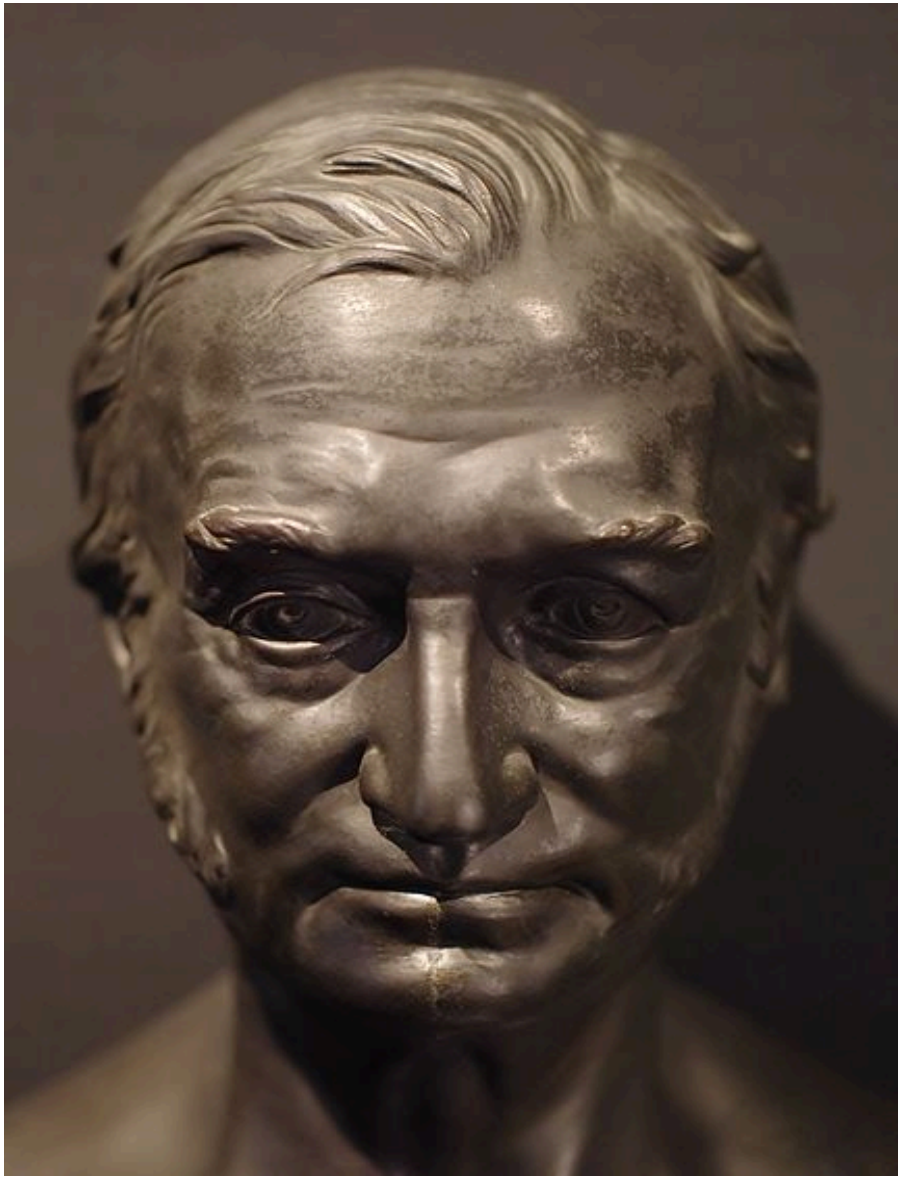
Overflow in a2i()

- **Scanning a positive decimal int literal:**
  - If number scanned so far is < 3276, which is max 32767 / 10, then scanning 1 more digit is safe, so continue:
  - `number = number * 10 + ( c - '0' );`
  - But if number scanned so far is in critical range = 3276, then only some digits are safe, namely ’0’ . . ’7’
  - Else there shall be integer overflow in positive int range!
- **Scanning a negative decimal int literal:**
  - If number scanned so far is less than 3276, which is max 32767 / 10, then scanning 1 more digit is safe, do
  - `number = number * 10 + ( c - '0' );`
  - But if number scanned so far is in critical range 3276, then only some digits are safe, namely ‘0’..’8’
  - Else there shall be overflow

Overflow in a2i()

```
#define MAX      32767
#define MAXby10   3276    // i.e. integer divide 32767 / 10 = 3276
. . .
int a2i1( void )
{ // a2i
    int  number =  0;
    bool neg    =  0;
    if( c == '-' ) {
        neg = true;
        c = getchar();
    }else if( c == '+' ) {
        c = getchar();   // keep neg = 0
    } //end if
    while( is_digit( c ) ) {
        if( number > MAXby10 ) {
            abort( "Overflow 1" );
        }else if( number == MAXby10 ) {
            if( c == '9' ) {
                abort( "Overflow 2" );
            }else if( c == '8' ) {
                if( neg ) {
                    return -32768;   // skip further input digits
                }else{
                    abort( "Overflow 3" );
                } //end if
            }else{
                number = number * 10 + ( c - '0' );
            } //end if
        }else{ // number < MAXby10, so OK to multiply!
            number = number * 10 + ( c - '0' );
       } //end if
       c = getchar();
    } //end while
    return neg ? -number : number;     // is this correct students?
} //end a2i
```

Titan of Arithmetic



Carl Friedrich Gauss 1777 to 1855

Overflow in a2i()

- The critical step: number = number * 10 + next digit is only performed when provably safe:
- For positive state, if the next digit after 3276 is '0'..'7' then one more iteration * 10 is safe; yielding <= 32767
- For negative state, if the next digit after 3276 is '0'..'8' one more iteration * 10 is safe; yielding <= 32768; to be negated
- On a 16-bit arch: signed int 32768 not representable
- Similarly on today's 64-bit computer; the scanner for integer literals needs to be safe, and detect integer overflow before it happens
- Common for computers to ignore integer overflow
- Above checks integer overflow on 16-bit architecture

MaxInt For 32-Bit Precision

```
#include <iostream.h>
#defined POW_EXPO 32

. . .

// compute powers of 2, but then -1
int main( void )
{ // main
   int expo;                        // shift 1 32 times
   unsigned long int pow = 1; // multiply rep. by 2

   for( expo = 1; expo <= POW_EXPO; expo++ ) {
     pow = pow * 2;
     cout << " 2**" << expo << " = " << pow-1 << endl;
   } // end for
   cout << endl;
   . . .
} //end main
```

MaxInt Higher Precisions

```
2**1-1 = 1
2**2-1 = 3
2**3-1 = 7
. . .
2**14-1 = 16,383
2**15-1 = 32,767
2**16-1 = 65,535
2**17-1 = 131,071
2**18-1 = 262,143
2**19-1 = 524,287
2**20-1 = 1,048,575
2**21-1 = 2,097,151
2**22-1 = 4,194,303
2**23-1 = 8,388,607
2**24-1 = 16,777,215
2**25-1 = 33,554,431
2**26-1 = 67,108,863
2**27-1 = 134,217,727
2**28-1 = 268,435,455
2**29-1 = 536,870,911
2**30-1 = 1,073,741,823
2**31-1 = 2,147,483,647
2**32-1 = 4,294,967,295
. . .
2**63-1 = 9,223,372,036,854,775,807
2**64-1 = 18,446,744,073,709,551,615
```

No “Integer” Overflow Here

ASCII Characters

- ASCII stands for American Standard Code for Information Interchange
- Computers only *understand* numbers, so ASCII is a numerical representation for characters such as 'a'
- ASCII was developed long ago; non-printable characters are used, but rarely for their original purpose
- ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure
- Below is the ASCII character table

ASCII Characters

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

References

1. Powers of 2: http://www.tsm-resources.com/alists/pow2.html
2. ASCII Table: https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html
3. Wiki Integer Overflow: https://en.wikipedia.org/wiki/Integer_overflow