CEng-140
Week 2

Data Types and Expressions

Data Types in C

- char, int, float, double
- **char**: a character in C character set

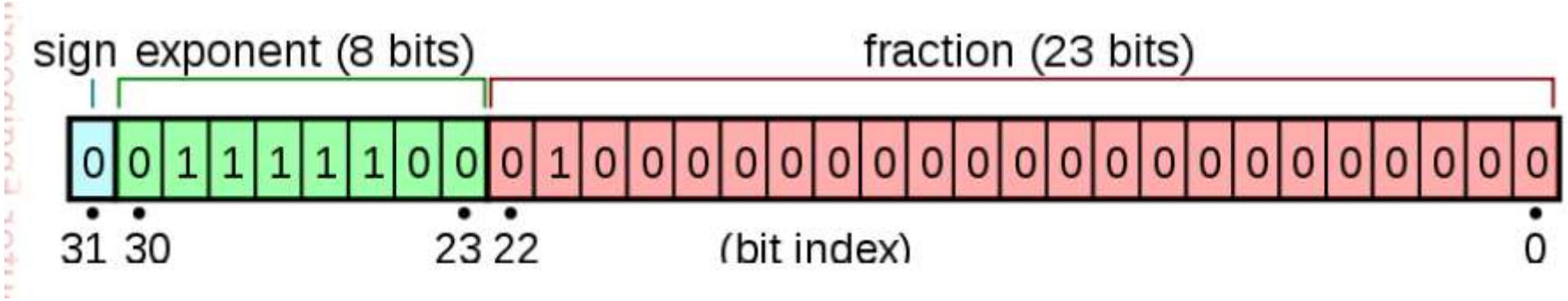
Character	Meaning	Character	Meaning
0,1,. . .,9	Decimal digits	:	Colon
A,B,. . .,Z	Uppercase letters	;	Semicolon
a,b,. . .,z	Lowercase letters	<	Less than
!	Exclamation point	=	Equal to
"	Double quotation mark	>	Greater than
#	Number/pound sign	?	Question mark
$	Dollar sign	@	"At" sign
%	Percent sign	[	Left bracket
&	Ampersand sign	\	Backslash
'	Apostrophe/single quotation mark	]	Right bracket
		^	Caret/circumflex
(	Left parenthesis	_	Underscore
)	Right parenthesis	`	Accent grave/back quotation mark
*	Asterisk		
+	Plus	{	Left brace
,	Comma	|	Vertical bar
-	Minus/hyphen	}	Right brace
.	Period	~	Tilde
/	Slash	b̸	Blank/space

The set of characters that may appear in a legal C program:
graphic (letters, decimal digits, …)
or non-graphic (backslash + letter)

Character	Meaning
\b	Back space
\n	New line
\t	Horizontal tab

Data Types in C

- **float:** single-precision floating point number
- **double:** double-precision floating point number
- IEEE 754 standard for single-precision (32 bit) floating point number



- Be careful with the precision loss!

Precision loss

Consider 33554433

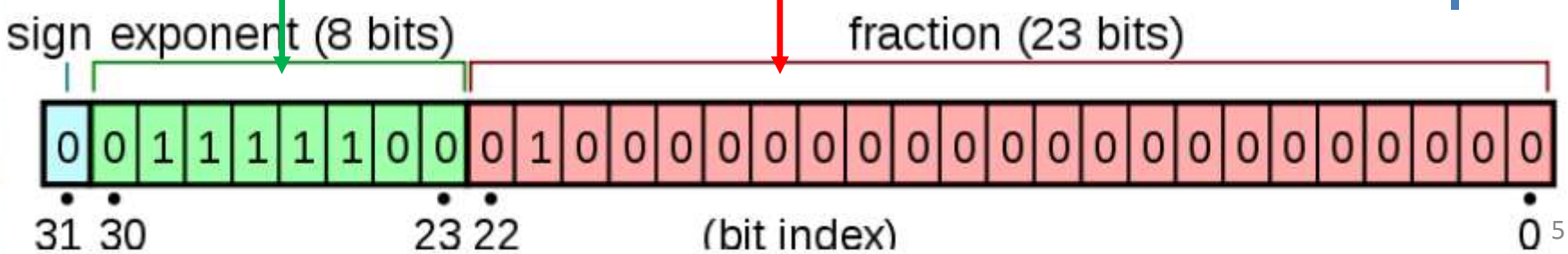
* Binary representation (blanks are for easy reading):

10 0000 0000 0000 0000 0000 0001

- Scientific format:

1. 0 0000 0000 0000 0000 0000 0001 x 2^{25}

2^{25} x 1. **0 0000 0000 0000 0000 0000 0001**



Representing 33554433

- The number I could store is:

2^{25} x 1. **0 0000 0000 0000 0000 0000 0001**

What is it in decimal?

Data Types in C

- **int**
  - We can apply qualifiers **short** and **long** to int
  - We can apply qualifiers **signed** and **unsigned** to all **int** types (default is signed) and to **char** type

short int (abbreviated as short)

unsigned short int

int

unsigned int

long int (abbreviated as long)

unsigned long int

sizeof(int) >= sizeof(short)
sizeof(long)>=sizeof(int)

Data Types in C

- Note that long qualifier can also apply to double to represent extended precision floating point
  - long double

integer and char types	→ integral type
float, double, long double	→ floating-point type

Altogether these types are called **Arithmetic type**

Classes of Data

- A C program manipulates
  - Variables
  - Constants

Variables

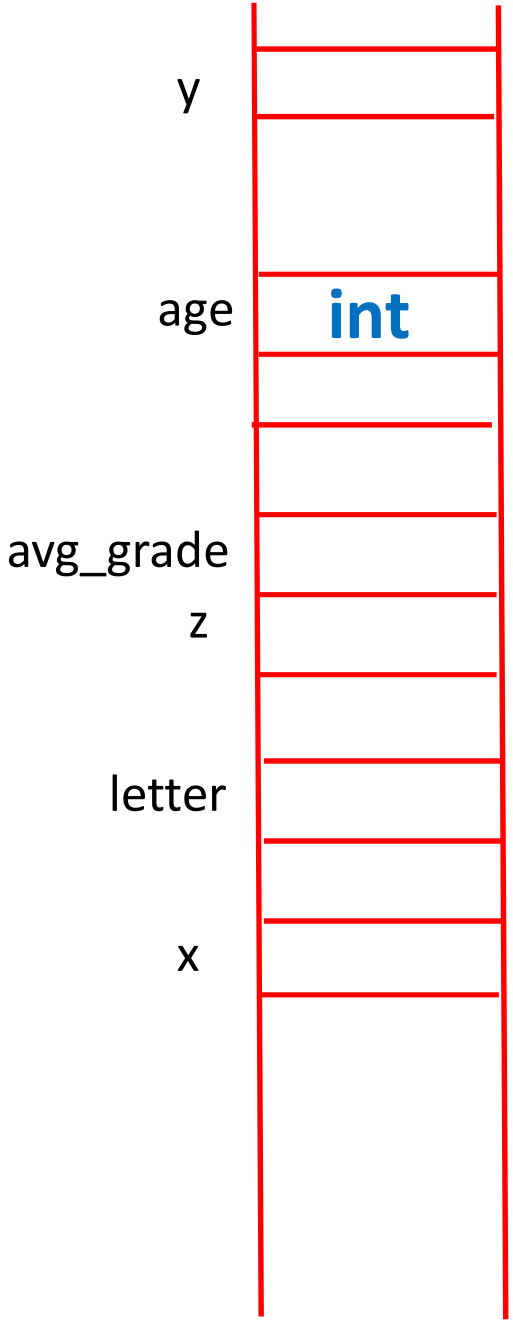
- A **variable** is an entity to store values used in the program
- Every variable
  - must have a **type** and
  - must be **declared** before it is used
- Variable declaration:
  - A type name followed by list of variable names (separated by commas)

```
int age;

float avg_grade;

char letter;

int x, y, z;
```



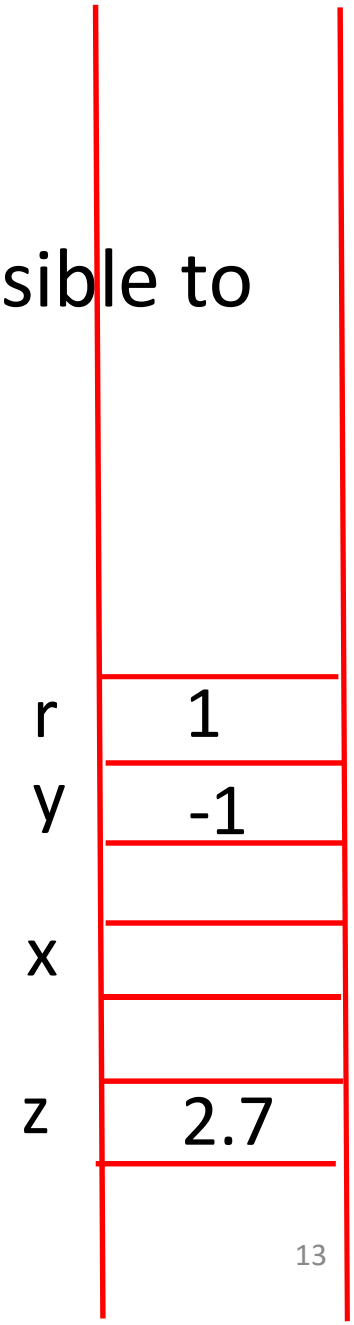
Variables

- A variable name is an **identifier**
  - identifier: (letter | _) (letter | digit | _)*
  - letter: a |…| z | A | …|Z
  - digit: 0 | 1 | 2… |9
- So, how about these:
  - 4everX
  - x2.5
  - ansi c

Variables

- **Initialization**: while declaring it is possible to assign an initial value
- All var declarations are usually at the <u>beginning</u> of a function, before the executable statements

```
int main(void)
{ int r=1;
  float x, y= -1.0, z=2.7; }
```



Variables

- Variable names are **case sensitive**

int Revenue, revenue;

- C keywords (like break, do, while, if, else, int, char, …) cannot be used as variable names

Constants

- A **constant** is an entity whose value does not change during the program execution
- We will consider 5 types of constants
  - integer constants
  - floating-point constants
  - character constants
  - string constants
  - enumeration constants

Integer Constants

- An integer constant is a number that has an integer value and normally, of type **int**
  - can be explicitly long or unsigned, etc: 1234L, 1234l, 123U, 123u

- A decimal integer constant
  - is a sequence of decimal digits (0 to 9)
  - First digit can’t be 0 (except 0 itself)

Integer Constants

- An octal integer constant
  - starts with 0 and followed by octal digits (0 to 7)
- A hexadecimal integer constant
  - starts with 0x (or, 0X) and followed by hexadecimal digits (0, …, 9, A, …F)
- Commas and spaces are not allowed in integer constants

Floating-point Constants

- An floating-point constant is a number that has a real value and normally, of type **double**
  - May explicitly specify float (0.5f, 0.5F), long double (1e-5L)
  - Written with a decimal point
    - 1.0, 1., .1, 0., .0 …
  - Scientific nottaion supported
    - 1.1e-8

Character constants

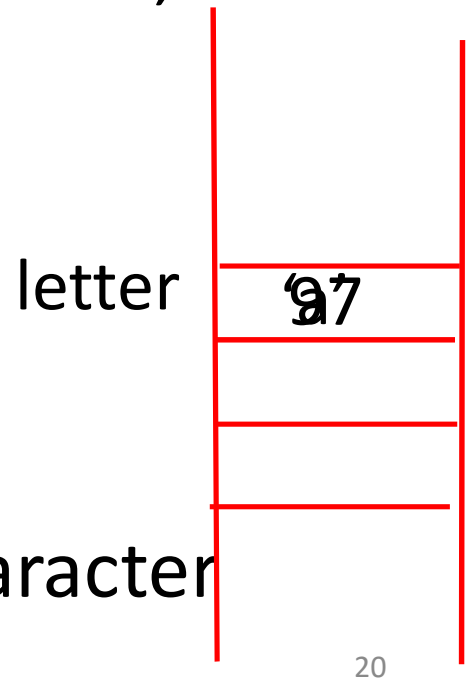
- Consists of a single character in ‘ ‘ (apostrophes)
  - Grpahich characters: ‘0’, ‘a’, ‘?’, ‘Z’,
  - Non-graphic: ‘\n’, ‘\t’, ‘\v’
  - Special case for apostrophe of backslash: ‘\’’ ‘\\’

Character constants

- Character constants are of type **int**
- Value of a character constant is the numeric value of the character in machine's character set (e.g., ASCII)

char letter = 'a';

Character Constant	ASCII
'0'	48
'a'	97
'Z'	90
'?'	63
'%'	37



- Special case: \0 represents the character with value 0, i.e, **NULL character**

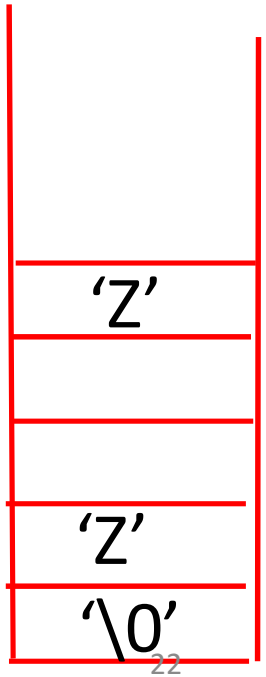
Character constants

- Character constants are of type **int**
- Hence,we can also use an arbitrary bit pattern (of byte size) to specify a character constant
  - '\ooo' (up to three octal digits)
  - '\xhh' (up to two hexadecimal digits)

Character Constant		Decimal Value	ASCII Character
Octal	**Hexadecimal**		
'\100'	'\x40'	64	'@'
'\135'	'\x5d'	93	']'
'\176'	'\x7e'	126	'~'

String constants

- Zero or more characters enclosed in quotation marks
  - “ “ are delimiters but not part of the string
- Non-graphic characters may be used in string constants
  - “Hi“, “\tC is fun!\n“
- C automatically adds the NULL character at the end of each string
  - So physically stored in one more byte than the number of chars in the string
  - ‘Z’ is not the same as “Z“



CEng-140

Data Types and Expressions

Arithmetic Operators

- An **operator** is a symbol that causes **mathematical** or **logical** manipulations to be performed.
- Arithmetic operators:
  - **Binary**: + , -, * , / , remainder(%),
  - **Unary:** + , -, increment (++), decrement (--)
  - All (except %) operate on any arithmetic operands
  - % operates on any integral operands

When **/** applied to ints, result is **truncated** to int

- 12 / 7 = 1 but 12. / 7. = 1.71

Expressions

- A combination of **constants** and **variables** with **operators** is called an <u>expression</u>.
  - Balanced parentheses may be used in combining constants & vars with operators.
  - Constants and variables by themselves are also considered as expressions.

Arithmetic Expressions

- Arithmetic expression: char, int, floating-point data can participate as operands.
  - 0.12
  - 12.3/45.3
  - –(i+1)
  - i%j
  - i
  - –x
  - ‘\n’

Expression Evaluation

Every expression has a <u>value</u> that can be determined

by first <u>binding</u> the <u>operands</u> to <u>operators</u>

and then evaluating the expression.

Let's consider the example 32 + 2 * x

Expression Evaluation

In order to specify the order in which operands are bound to operators, C uses

- A precedence & associativity rule
- And, a parentheses rule

Higher precedence

Operator	Type	Associativity
+ -	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right

L to R associativity: if more than one op. with equal precedence, operator to the left has higher precedence

Expression Evaluation

- Parentheses can alter the order of the predence, i.e., force higher precedence levels
- The operation will be performed in the innermost set of parentheses (using precendence & associativity rule when appropriate), then outer parentheses, until all are consumed
- 32 + 2 * x vs. (32 + 2) * x

Assignment Operator & Statement

- An assignment expression is of the form:

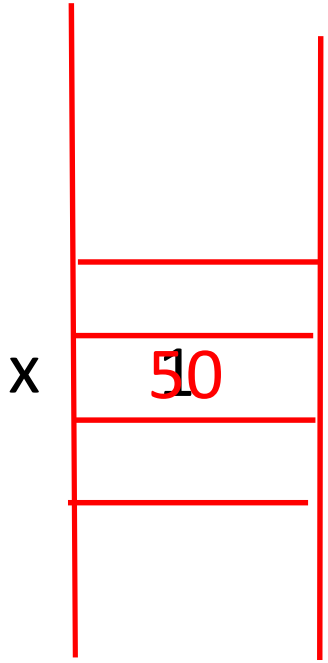
variable = *expression*

- When this is followed by a ; it becomes an assignment statement.
- The value of the assignment expr is the value of the *expression*, furthermore, = op assigns the value of the RHS (*expression*) to the LHS of the operator (and destroys the original value of LHS)
- x = y; vs. y = x;

The value of the assignment expr is the value of the *expression,* furthermore, = op assigns the value of the RHS (*expression*) to the LHS of the operator (and destroys the original value of LHS)

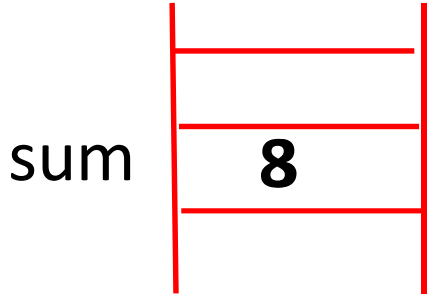
```
int x = 1;
```

$$\underbrace{x = \overbrace{25 * 2}^{50};}_{50}$$



Assignment Operator & Statement

- Precedence of the = operator is lower than the arithmetic operators.
  - sum = sum + 5;
  - Left operand of = must be an Lvalue
  - Lvalue is an object that can be examined and altered (e.g., a variable name)
  - Rvalue is an expression that permits examination but **not** alteration



15 =n; **?**

sum + 1.0 = 2.0; **?**

Increment & Decrement Operators

Prefix form: ++*var*

Postfix form: *var*++

Both increment var by one

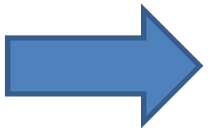
Prefix form: First increment, then use (inc before use)

Postfix form: First use, then increment (inc after use)

Operator	Type	Associativity
+ - ++ --	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right

Example

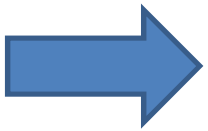
```
i = 1;
n = ++i;
```



```
i= i +1;
n = i;
```

Afterwards, i and n have the values 2.

```
i = 1;
n = i++;
```



```
n = i;
i= i +1;
```

Afterwards, i has the value 2, n has value 1

Compound Assignment Operators

- *var* op= *expr* → *var = var op expr*
- += -= *= /= %=

sum += 5; → sum = sum + 5;

Compound Assignment Operators

- Compound assignment ops have the **same precedence with =** and hence, lower than artihmetic operators.

Operator	Type	Associativity
+ - ++ --	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right
= += -= /= *= %=	Binary	Right to left

i *= a+1; means what?

i = (i * a) + 1; OR i = i * (a+1)

Nested assignments

- C allows multiple assigments in one statement, they are said to be nested.
- Assign op. is **right-associative**!

i = j = k = 0; → i = (j = (k = 0));

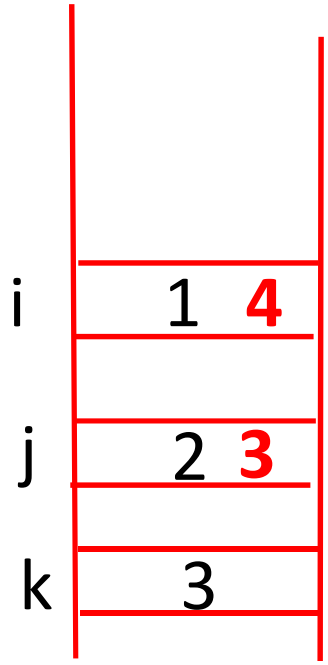
int i=1, j=2, k=3;

i += j = k;

j becomes 3, i becomes 4

i = j += k;

j becomes 5, i becomes 5

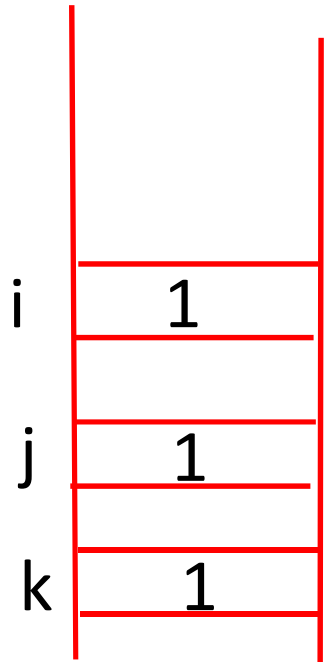


Example

```
int i , j, k;
i = j = k = 1;
i -= -j-- - --k;
```

What is i, j, k afterwards?

```
i -= -j-- - (--k);
i -= -(j--) - (--k);
i -= (-(j--)) - (--k);
i -= ((-(j--)) - (--k));
```



Example

0

i -= -j-- - (--k);

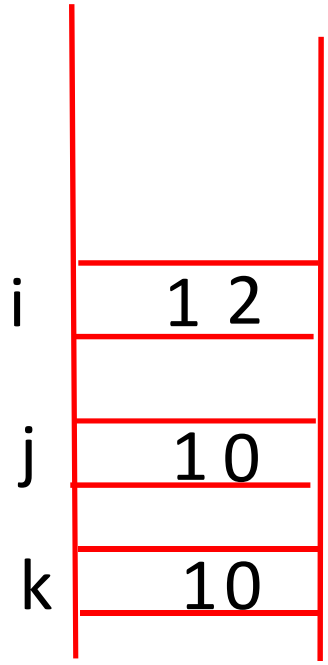
1 0

i -= -(j--) - (--k);

-1

-1

i -= -1; which is i = i - -1;



CEng-140

Data Types and Expressions

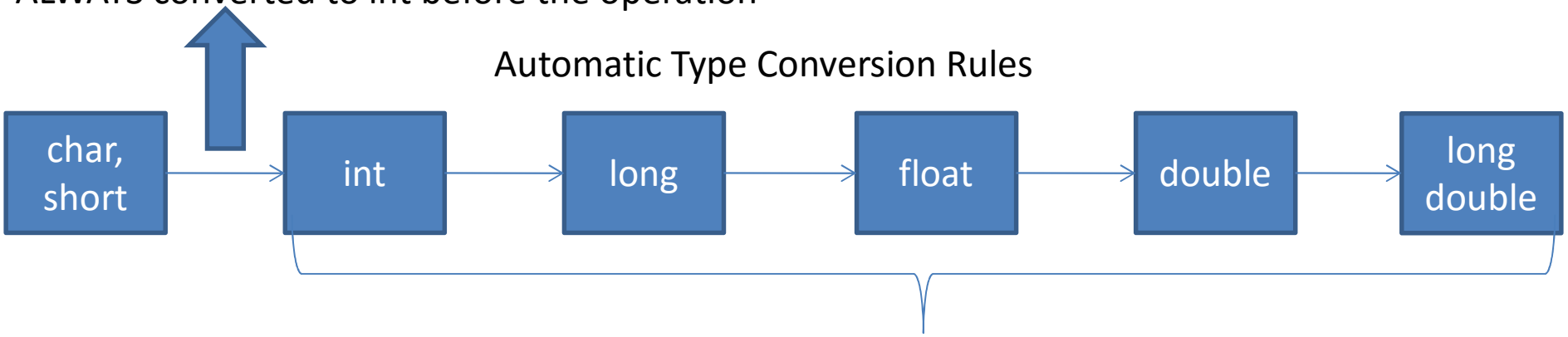
* Advice: Avoid automatic type conversion!

Type Conversions

1) **Automatic:** C performs all arith. ops with 6 data types: int, unsigned int, long, float, double, long double

Any operand of type char or short is
ALWAYS converted to int before the operation

Automatic Type Conversion Rules



Idea: If a binary operator has operands of two different types, lower type is **promoted to** higher type before the op. proceeds, and the **result** is also **higher type**.

Type Conversions

2) Explicit: (cast_type) *exp*

(int) 12.8 * 3.1 → 12 * 3.1 → 37.2

(int) (12.8 * 3.1)

Commonly used in divison:

```
int sum, n;
(float) sum / n ;
```

Type Conversions

3) Type converison in assignment:

When vars off different type are mixed in assignment exp, the type of the value of the exp on the RHS is automatically converted to the type of the variable on LHS.

- Conversion from lower to higher: changes form, but not more precision!
- Conversion from higher to lower: truncation and loss of info!

Type Conversions

3) Type converison in assignment:

- Conversion from **lower to higher**: changes form, but not more precision!

```
int x = 1;
float y = 7.92;
y = x; /* What is the value in y? */
```

- Conversion from **higher to lower**: **truncation** and loss of info!

```
int x = 1;
float y = 7.92;
x = y; /* What is the value in x? What if y = x; afterwards? */
```

Example

```
int x = 5, z;
float y;
x /= y = z = 1 +1.5;
```

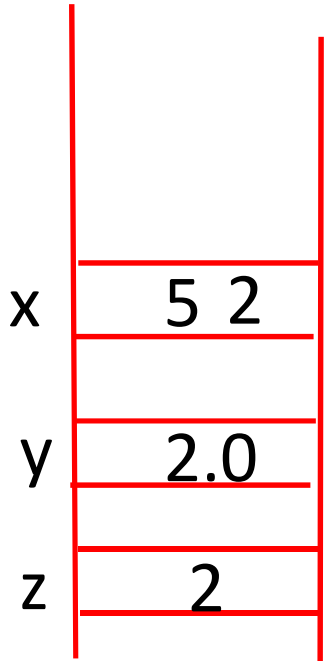
2.5

2

2.0

5 / 2.0 = 2.5

2



C Statements

- if, if else, while, for….

- Expression statements
  - x * 3 + y / 2 ;
  - x = y; /* assignment statement */
  - sum = sum + 5; /* assignment statement */
  - ++x -1;

More to come…

Remark: Order of Evaluation

While precedence and associativity rules (with parentheses) tell you what is grouped with what and these rules are well-defined,

the order in which these groupings will be evaluated is <u>mostly unspecified</u> (except for the operators && and ||).

Example: a *b + c * d or 231 * 1234 + 3 * 2

Which side is computed first ?

Does it matter while we are coding? When does it?

Side effects & Exp Eval Order

Problem: When exp has functions with side-effects or operations with side-effects on a variable that appears more than once!

int i=1, j =1;

i * (j=3) → OK

i * (i=3) → NOT OK

i * j++ → OK

i * i++ → NOT OK

AVOID SUCH EXPRESSIONS!!!

Output

- **printf**(control string, arg1, arg2, …)
  - Control string contains conversion specifications (after % symbol):
    - d,i: integers
    - f: float, double
    - e: float, double in exponential notation
    - c: character
    - s: string

```
int x=1; float y =23.4; char c='W';
printf("Here %d %f %c\n", x, y, c,);
```

Input

- **scanf**(control string, arg1, arg2, …)
  - arg1, arg2, ..: addresses of memory locations!
  - Control string contains:
    - d,i: integers
    - f: float, double
    - e: float, double in exponential notation
    - c: character
    - s: string

scanf(control string, arg1, arg2, …)

The control string can be viewed as a picture of the expected input and the scanf makes a matching between the control string and input stream. Control string may have:

- Whitespace: any sequence of consecutive whitespace characters in the control string matches any sequence of whitespace in the input stream.
- Conversion specifications. Begins with % and… (read p. 345)
- Ordinary characters

scanf(control string, arg1, arg2, …)

Processing whitespace:

- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces +NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read is converted to type appropriate for conversion control char (if cannot be done, behaviour is undefined)

- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces +NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
int i;
float f1, f2;
char c1, c2;
scanf("%d  %f  %e  %c  %c", &i, &f1, &f2, &c1, &c2);
```

10 1.0e1 10.0 a b

- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces+NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
int i;
float f1, f2;
char c1, c2;
scanf("%d %f  %e %c %c", &i, &f1, &f2, &c1, &c2);
```

10 1.0e1 10.0ab

- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces +NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
int i;
float f1, f2;
char c1, c2;
scanf("%d %f  %e %c%c", &i, &f1, &f2, &c1, &c2);
```

10 1.0e1 10.0a b

- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces +NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
int i;
float f1, f2;
char c1, c2;
scanf("%d %f", &i, &f1);
```

-123.456

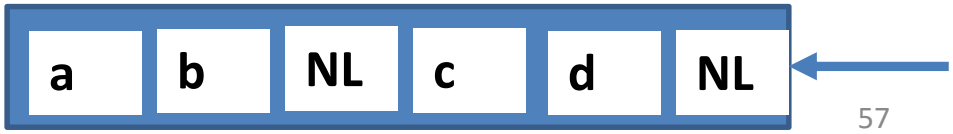
- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces+NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
char c1, c2;
scanf("%c %c", &c1, &c2);
printf("%c %c\n", c1, c2);
scanf("%c %c", &c1, &c2);
printf("%c %c\n", c1, c2);
```

Suppose my input is:
a b HIT ENTER
c d HIT ENTER



- Whitespace in CS is matched by reading input up to first non-whitespace characters (remains unread) and matches to largest whitespace sequence in the input (possibly of length 0)

Processing conversion specification:

- Input whitespaces+NL are skipped unless conv char has c ,n ,[
- The matching item is read from input per conv char; the first char after the matching input remains "unread"
- The read item is converted to type appropriate for conv control char (if cannot be done, behaviour is undefined)

```
scanf("%c %d", &c, &i);
printf("%c %d\n", c, i);
scanf("%f %lf", &f, &d);
printf("%f %f\n", f, d);
```

Suppose I first write: a 10 HIT ENTER
And then write 12.4 56.7 HIT ENTER

Suppose I first write: a b HIT ENTER
What happens?

Simple Macros

- Basic usage:

#define **macro_name** **sequence_of_tokens**

macro body

Pre-processor simply replaces the macro_name with the body of the macro!

- Not recognized in comments or string constants

Simple Macros

- For long and/or frequent constants:
  - #**define** PI 3.14159265
- For long and/or frequent calculations:
  - #**define** Area(Radius) (4*PI*Radius*Radius)
  - … a = 10.0 + Area(2.0);

Be careful while using macros!

- ```
  #define square(x) x*x
  main()
  {
  int i;
  i = 64/square(4);
  printf("%d",i);
  }
  ```

What is the output of the above program?
  ```