CS100
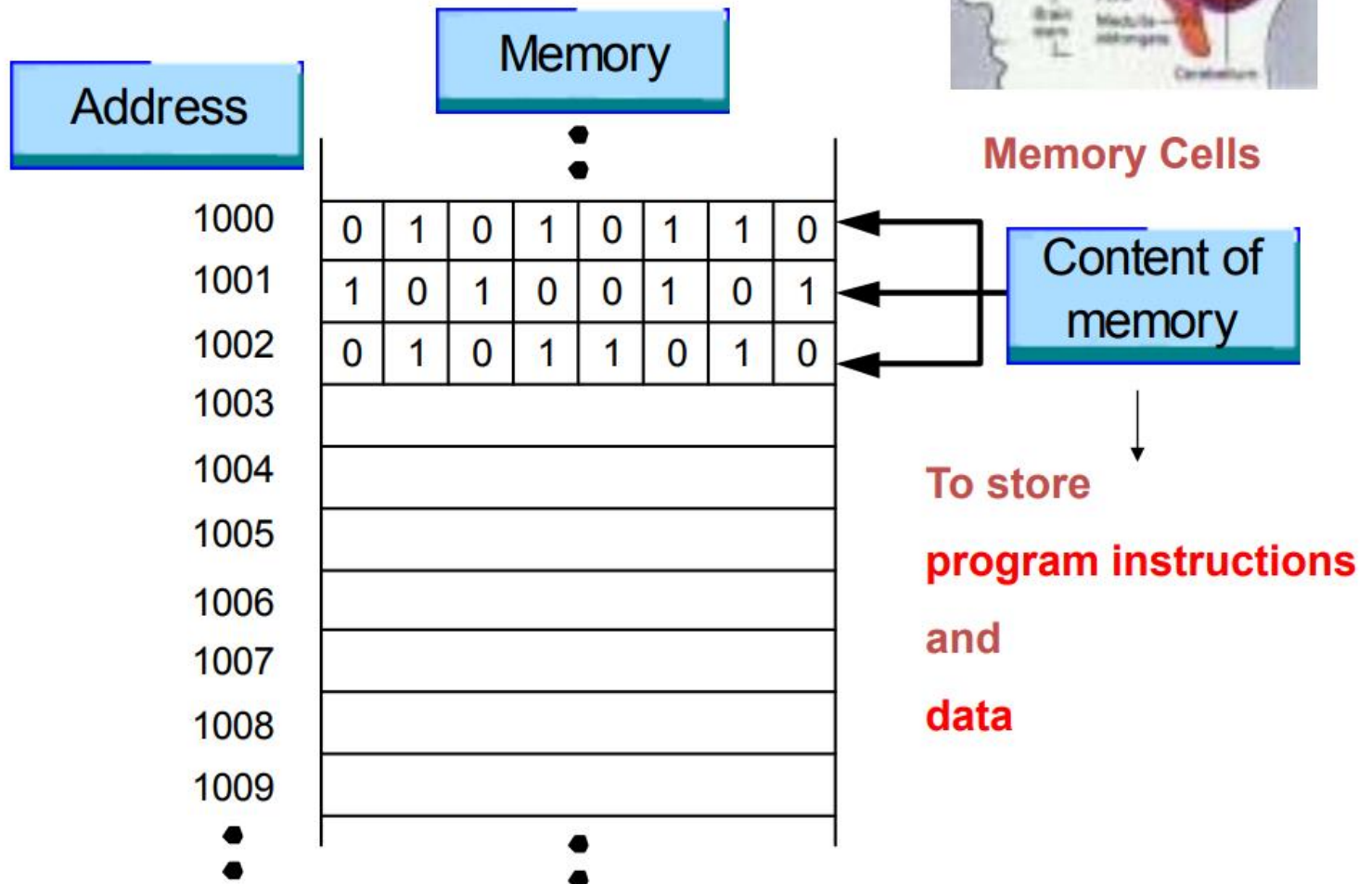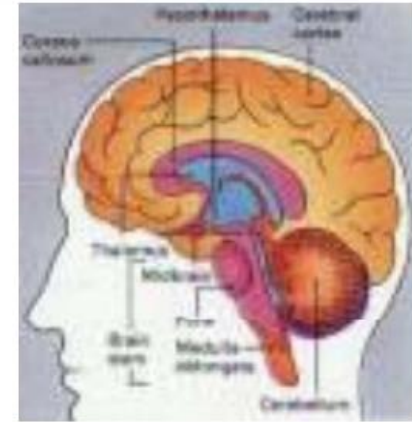# Introduction to Programming
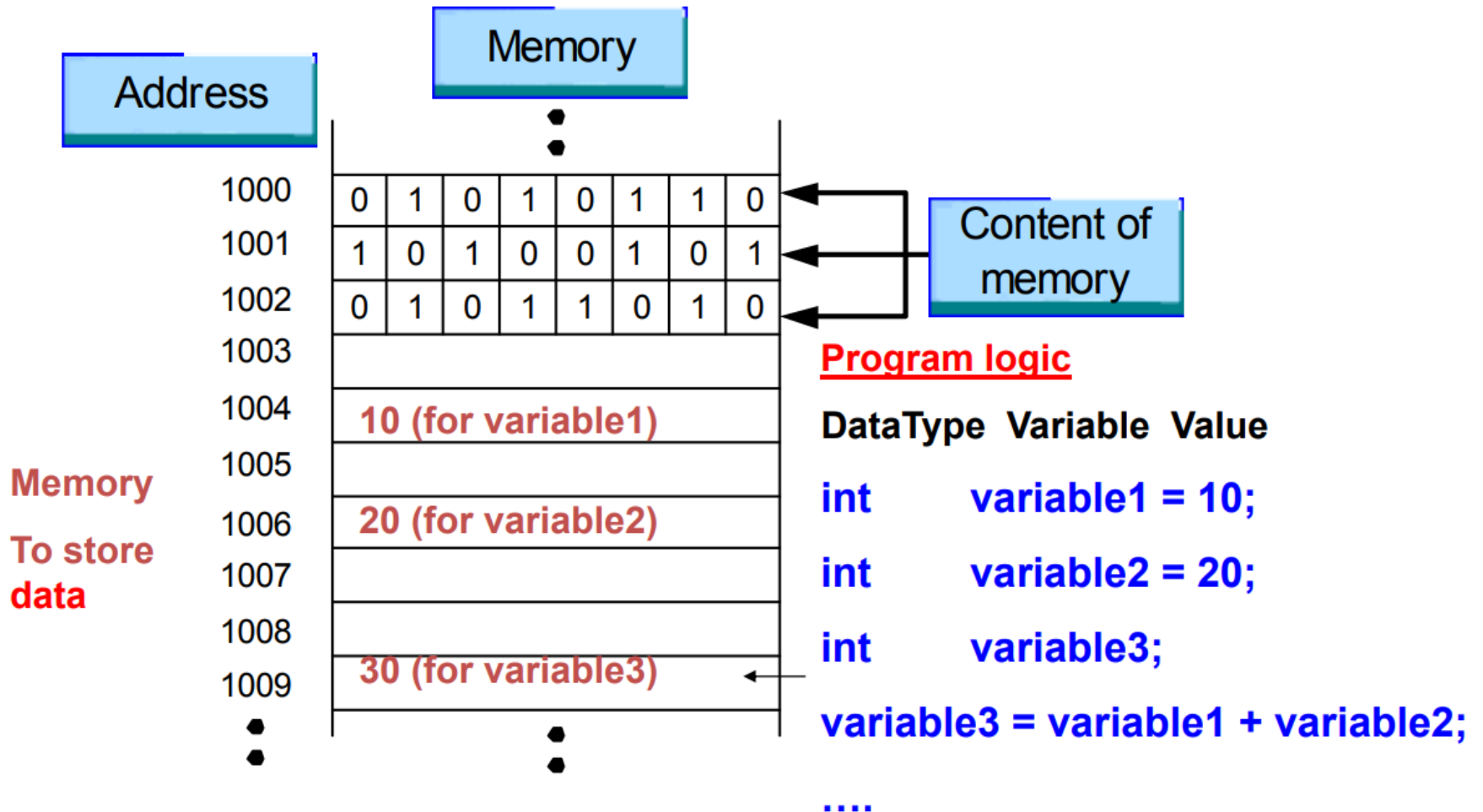
Lecture 2. Data types, operators and expressions

Memory & Data Types

Computer Memory

Address	Memory								Memory Cells
1000	0	1	0	1	0	1	1	0	Content of memory
1001	1	0	1	0	0	1	0	1	
1002	0	1	0	1	1	0	1	0	
1003									To store
1004									program instructions
1005									and
1006									data
1007									
1008									
1009									

Memory and Variables

Data Types

- It determines the kind of data that a variable can hold, how many **memory** cells are reserved for it and the operations that can be performed on it.

- **Integers**
  - short (2 bytes – 16 bits)
  - **int** (2 bytes?)
  - long 32 bits (4 bytes)
  - unsigned (2 bytes)
  - unsigned short (2 bytes)
  - unsigned long 32 bits (4 bytes)
- **Floating Points**
  - **float** (4 byte, or 32 bits)
  - **double** (8 bytes, or 64 bits)

- **Characters**
  - 128 distinct characters in the ASCII character set.
  - Two C character types:
    - **char** (1 byte or 8 bits, range: [–128, 127]
    - unsigned char (1 byte or 8 bits, range: [0, 255]

Data Types

- The amount of **memory** used for objects of these types is machine dependent.

- The **range** of the values allowed for each type depends on the number of bits used

- Choose the type whose range is **just enough** to cover all the possible values of the object, for **space efficiency**.

Literals

- Literals (constant values) are fixed **values** (associated with data type) used in the program.

- Four types of literals:
  - **Integer** literals, e.g. 100, −256
  - **Floating-point** literals, e.g. 2.4, −3.0
  - **Character** literals, e.g. 'a', '+'
  - **String** literals, e.g. "Hello World"

Variables

- A **variable** is a name given to the memory cell(s) where the computer uses to store data.

- A variable's **name** allows the program to refer to the variable.

- It is a good practice to follow the naming convention.

- The following C **keywords** are reserved and cannot be used as variable names

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
struct	switch	typedef	union	sizeof	static
volatile	while	unsigned	void		

Variable Declaration

- To use a variable, you must first declare the variable.
- A variable declaration always contains 2 components:
  - its **data type** (e.g. short, int, long, etc.)
  - its **name** (e.g. count, numOfSeats, etc.)
- Syntax for variable declaration:

  < **data type** > < **name** >

- Below are some examples of variable declarations:

  ```
  int count;
  float temperature, result;
  ```

- Below are some examples of variable initializations:

  ```
  int count = 20;
  float temperature, result;
  temperature = 36.9;
  ```
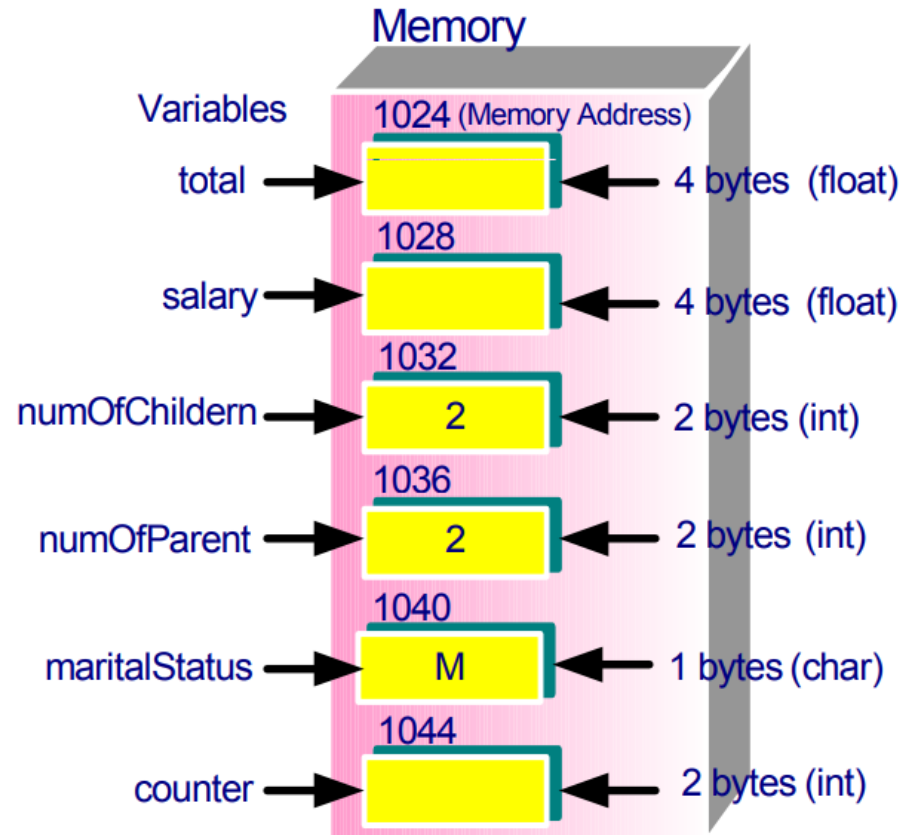
Declaring Variables with Initialization

- **Example**

```
int main()
{
    float total, salary;
    int numOfChildren = 2;
        numOfParents = 2;
    char maritalStatus = 'M';
    int counter;

    ......
    return 0;
}
```


Memory

Variables		
total →	1024 (Memory Address)	← 4 bytes (float)
salary →	1028	← 4 bytes (float)
numOfChildern →	1032 · 2	← 2 bytes (int)
numOfParent →	1036 · 2	← 2 bytes (int)
maritalStatus →	1040 · M	← 1 bytes (char)
counter →	1044	← 2 bytes (int)

- In this example, total and salary are declared without initial values and the other variables are declared with initial values.

Data Types Recall

- It determines the kind of data that a variable can hold, how many **memory** cells are reserved for it and the operations that can be performed on it.

- **Integers**
  - short (2 bytes – 16 bits)
  - **int** (2 bytes?)
  - long 32 bits (4 bytes)
  - unsigned (2 bytes)
  - unsigned short (2 bytes)
  - unsigned long 32 bits (4 bytes)
- **Floating Points**
  - **float** (4 byte, or 32 bits)
  - **double** (8 bytes, or 64 bits)

- **Characters**
  - 128 distinct characters in the ASCII character set.
  - Two C character types:
    - **char** (1 byte or 8 bits, range: [–128, 127]
    - unsigned char (1 byte or 8 bits, range: [0, 255]

ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) Codes (1 byte)

	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Examples of Escape Sequence

- Some useful **non-printable control characters** are referred to by the ***escape sequence*** which is a better alternative, in terms of memorization, than numbers, e.g. `'\n'` the newline (or linefeed) character instead of the number **10**.

'\a'	alarm bell	'\f'	form feed	'\n'	newline
'\t'	horizontal tab	'\"'	double quote	'\v'	vertical tab
'\b'	back space	'\\'	backslash	'\r'	carriage return
'\''	single quote				

Expressions

Operators

- Arithmetic operators: +, −, *, /, %
  - E.g. 7/3 (= 2); 7%3 (= 1); 6.6/2.0 (=3.3); etc.
- Assignment operators:
  - E.g. float amount = 25.50;
- Chained assignment:
  - E.g. a = b = c = 3;
- Arithmetic assignment operators: +=, −=, *=, /=, %=
  - E.g. a += 5 (meaning a = a + 5).
- Relational operators: ==, !=, <, <=, >, >=
  - E.g. 7 >= 5 (this returns TRUE).
- Incremental / decremental operators: ++, −−
  - E.g. a++ (means a = a + 1); b−− (means b = b − 1).

Increment/decrement Operators

- **increment operator**: ++ can be used in two ways, prefix and postfix modes. In both forms, the variable will be incremented by 1.

- In prefix mode: ++varName

  (1) varName is incremented by 1 and

  (2) the value of the expression is the updated value of varName.

- In postfix mode: varName++

  (1) The value of the expression is the current value of varName and

  (2) then varName is incremented by 1.

- The way the **decrement operator** '−−' works is the same as the "++", except that the variable is decremented by 1.

Increment/decrement Operators

```c
#include <stdio.h>
int main(void)
{
    int n = 4, num = 4;
    printf("value of n is %d\n", n);
    printf("value of n++ is %d\n", n++);
    printf("value of n is %d\n", n);
    printf("value of ++n is %d\n", ++n);
    printf("value of n is %d\n\n", n);

    printf("value of num is %d\n", num);
    printf("value of num-- is %d\n", num--);
    printf("value of num is %d\n", num);
    printf("value of --num is %d\n", --num);
    printf("value of num is %d\n", num);
    return 0;
}
```

Output:

value of n is 4

value of n++ is 4

value of n is 5

value of ++n is 6

value of n is 6

value of num is 4

value of num-- is 4

value of num is 3

value of --num is 2

value of num is 2

Constants

- A constant is an object whose value is **unchanged** throughout the life of the program.

- There are **three** ways to define a constant:

1) directly give the value

```
print("p = %f.\n", 3.14159);
/* 3.14159 is a floating point constant */
```
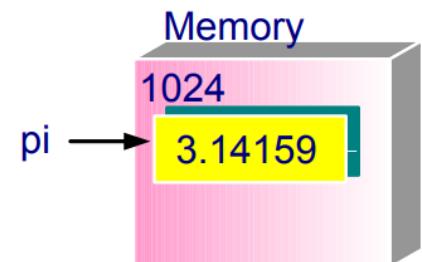
2) define a constant variable

format: **const type varName = value**

where

   type: int, float, char, etc.

   varName: name of the constant variable

```
const float pi = 3.14159;
/* declare a float constant variable pi with value
3.14159 */
printf("p = %f.\n", pi);
```

Memory

1024

pi → 3.14159

Constants

3) use the preprocessor directive #define

Format: **#define constantName value**

where constantName is name of the constant.

(constantName should use *upper* case).

```
#include <studio.h>
#define TAX_RATE 0.12 //define a constant TAXRATE with value 0.12
int main()
{
    float income1, income2, tax;
    tax = income1 * TAX_RATE; //substituted by 0.12
    tax = tax + income2 * TAX_RATE; //substituted by 0.12
    return 0;
}
```

- During compilation, the value of the constant will be **substituted** whenever the name of the constant appears in the program
- By giving a name to a constant,
  - it improves the readability of the program
  - it makes programs easier to be modified

Expressions

- An **expression** is any combination of variables, constants and operators that can be evaluated to yield a result.
  - Examples: a+b; count++; (item1 + item2) * tax_rate; speed = distance/time;

- You can tell the compiler explicitly how you want an expression to be evaluated by using **parentheses (** and **)** .
  - Note: (1 + 2 * 3) is different from ((1 + 2) * 3)

- To make your code easier to read and maintain, you should be explicit and indicate with parentheses whenever possible.

Operator Precedence

- The expression is evaluated according to the priority of the operator

Higher priority

Operator	Meaning	Associativity
()	parentheses	left to right
++, --	increment, decrement	right to left
+,-	unary	right to left
(Type)	type cast	right to left
*, /, %	multiplication, division, modulus	left to right
+,-,+	binary addition, subtraction, String concatenation	left to right
=,+=,-=,*=,/=	assignment	right to left

Lower priority

- Higher priority should be evaluated first

X = a+(a-b*b++)/c

Full List of Operators with Precedence

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- *type*() *type*{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (*type*) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of[note 1] await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
10	== !=	For relational operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= |=	Ternary conditional[note 2] throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR	Right-to-left
17	,	Comma	Left-to-right

Data Type Conversion

Arithmetic operations require two numbers in an expression/assignment are of the same type.

There are three kinds of conversions :

1. Explicit conversion: uses the type casting operators, i.e. (int), (float), …, etc.
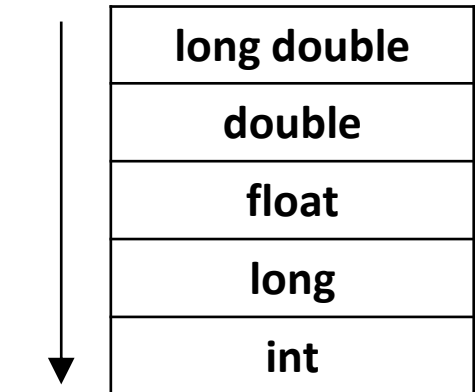
- e.g. `(int)2.7 + (int)3.5`

2. Arithmetic conversion: in mix operation it converts the operands to be type of the **higher** ranking of the two

- e.g. `2 + 3.5; // convert to float`

3. Assignment conversion: converts the type of result of computing the expression to that of the type of the left hand side if they are different:

- e.g. `num = 2.7 + 3.5; // num is int`

High

long double
double
float
long
int

Low

23

Data Type Conversion

```c
#include <stdio.h>
int main(){
    int num;
    /* Explicit Conversion */
    num = (int)2.7 + (int)3.5;
    /* convert 2.7 to 2 and 3.5 to 3
    then do addition */
    printf("num = %d\n", num);

    /* Assignment Conversion */
    num = 2.7 + 3.5;
    /* add 2.7 and 3.5 to get 6.2, then
    convert it to 6 */
    printf("num = %d\n", num);

    /* Arithmetic Conversion */
    /* converts 2 to 2.0 then do
    addition */
    printf("num = %f\n", 2 + 3.5);
    return 0;
}
```
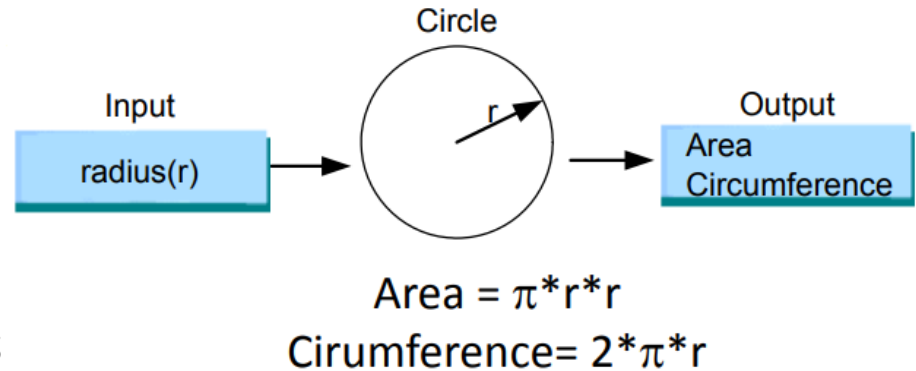
Output
num = 5
num = 6
num = 5.500000

Possible *pitfalls* of data type conversion -

Loss of precision: e.g. from **float** to **int**, the fractional part is lost.

24

A C Program Example

```c
#include <stdio.h>
int main()
{
    const float PI = 3.14;
    float radius, area, circumference;
    // Read the radius of the circle
    printf("Enter the radius: ");
    scanf("%f", &radius);
    // Calculate the area
    area = PI * radius * radius;
    // Calculate the circumference
    circumference = 2 * PI * radius;
    // Print the area and circumference of the circle
    printf("The area is %0.1f\n", area);
    printf("The circumference is %0.1f", circumference);
    return 0;
}
```


Input
radius(r)

Circle
r

Output
Area
Circumference

$Area = \pi * r * r$
$Cirumference = 2 * \pi * r$

In C:
Output function: printf()
Input function: scanf()

Mathematical Library Functions
#include <math.h>

Function	Argument Type	Description	Result Type
ceil(x)	double	Return the smallest **double** larger than or equal to **x** that can be represented as an **int**.	double
floor(x)	double	Return the largest **double** smaller than or equal to **x** that can be represented as an **int**.	double
abs(x)	int	Return the absolute value of **x**, where **x** is an **int**.	int
fabs(x)	double	Return the absolute value of **x**, where **x** is a floating point number.	double
sqrt(x)	double	Return the square root of **x**, where **x** $>= 0$.	double
pow(x,y)	double x, double y	Return x to the y power, x^y.	double
cos(x)	double	Return the cosine of **x**, where **x** is in radians.	double
sin(x)	double	Return the sine of **x**, where **x** is in radians.	double
tan(x)	double	Return the tangent of **x**, where **x** is in radians.	double
exp(x)	double	Return the exponential of **x** with the base e, where e is 2.718282.	double
log(x)	double	Return the natural logarithm of **x**.	double
log10(x)	double	Return the base 10 logarithm of **x**.	double

Executing Programs
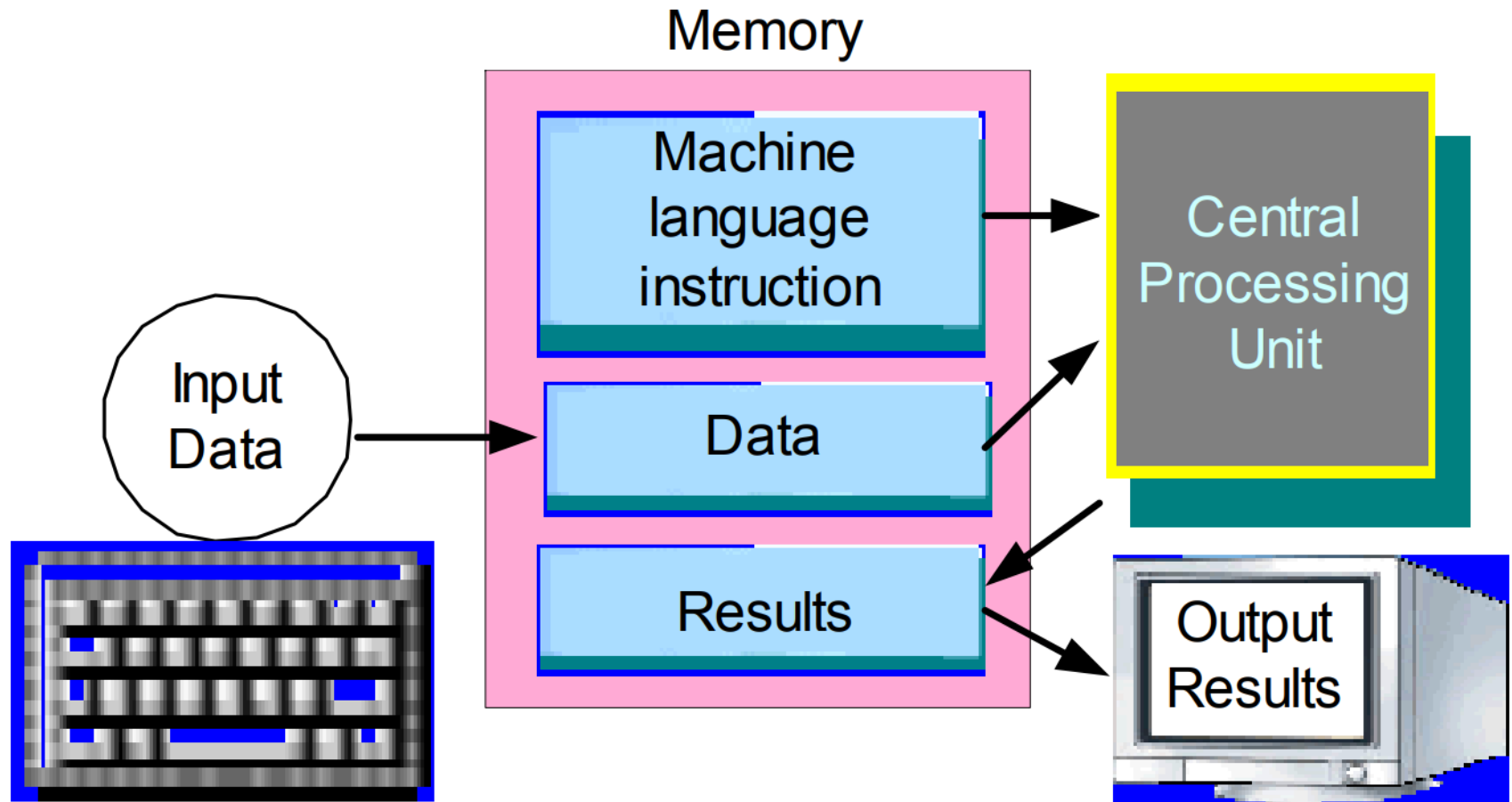
Simple Output: printf()

- The printf() statement has the form:

  printf(**control-string**, **argument-list**);

- The **control-string** is a string constant. It is printed on the screen.
  - %?? is a conversion specification. An item will be substituted for it in the printed output.

- The **argument-list** contains a list of items such as item1, item2, …, etc.
  - Values are to be substituted into places held by the conversion specification in the control string.
  - An item can be a constant, a variable or an expression like num1 + num2.
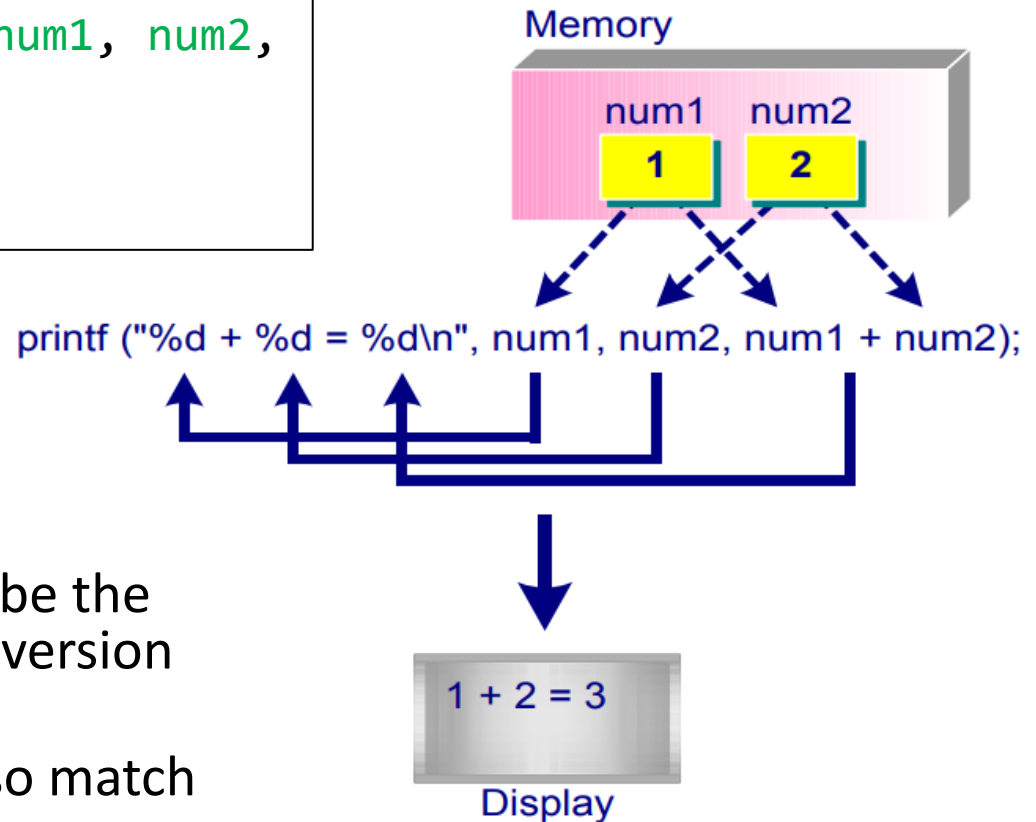
printf() – Example 1

```c
#include <stdio.h>
int main()
{
    int num1 = 1, num2 = 2;
    printf("%d + %d = %d\n", num1, num2,
        num1 + num2);
    return 0;
}
```

Output:

1 + 2 = 3

Note:

- The number of items must be the same as the number of conversion specifiers.
- The types of items must also match the conversion specifiers.

Memory

num1   num2

| 1 | 2 |

printf ("%d + %d = %d\n", num1, num2, num1 + num2);

1 + 2 = 3

Display

printf() – Conversion Specification

Type of *Conversion Specifiers*

d	**signed decimal conversion of int**
o	**unsigned octal conversion of unsigned**
x, X	**unsigned hexadecimal conversion of unsigned**
c	**single character conversion**
f	**signed decimal floating point conversion**
s	**string conversion**

printf() – Example 2

```c
#include <stdio.h>
int main()
{
  int num = 10;
  float i = 10.3;
  double j = 100.0;

  printf("int num = %d\n", num);
  printf("float i = %f\n", i);
  printf("double j = %f\n", j);
   /* by default, 6 digits are
      printed after the decimal
      point */

  return 0;
}
```

Output:
int num = 10
float i = 10.300000
double j = 100.000000

Logical Operations

Relational Operators

Used for **comparison** between **two values**.

Return **Boolean** result: **true** or **false**.

Relational Operators:

operator	example	meaning
==	ch == 'a'	equal to
!=	f != 0.0	not equal to
<	num < 10	less than
<=	num <=10	less than or equal to
>	f > -5.0	greater than
>=	f >= 0.0	greater than or equal to

Logical Operators

- Work on one or more relational expressions to yield a logical value: **true** or **false**.
- Allow testing and combining the results of comparison expressions.

Logical Operators:

operator	example	meaning
!	!(num < 0)	not
&&	(num1 > num2) && (num2 >num3)	and
||	(ch == '\t') || (ch == ' ')	or

	A is true	A is false
!A	false	true

A && B	A is true	A is false
B is true	true	false
B is false	false	false

A || B	A is true	A is false
B is true	true	true
B is false	true	false

34

Precedence of operators

- List of operators of **decreasing precedence**:

!	not
* /	multiply and divide
+ -	add and subtract
< <= > >=	less, less or equal, greater, greater or equal
== !=	equal, not equal
&&	logical and
||	logical or

- **Example**: The expression **!(5 >= 3)\|\|(7 > 3)** is true, where the **logical or operator \|\|** is executed in the end.

Boolean Result

- The result of evaluating an expression involving relational and/or logical operators is
  - either **true** or **false**
  - either **1** or **0**
  - When the result is **true**, it is **1**. Otherwise, it is **0**. That is, the C language uses 0 to represent a false condition.

- In general, **any integer expression whose value is non-zero is considered true**; otherwise it is **false**.

- Examples:

`3`	is true
`0`	is false
`1 || 0`	is true
`!(5 >= 3) || 0`	is false

Recap

- This lecture covers the following concepts:
  - Data Types
  - Literals
  - Constants
  - Variables
  - Operators
  - Expressions
  - Data Type Conversions
- Next:
  - IO, maybe part of Control Flow