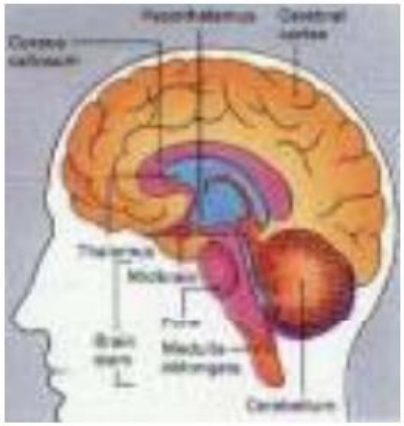
CS100
Introduction to Programming

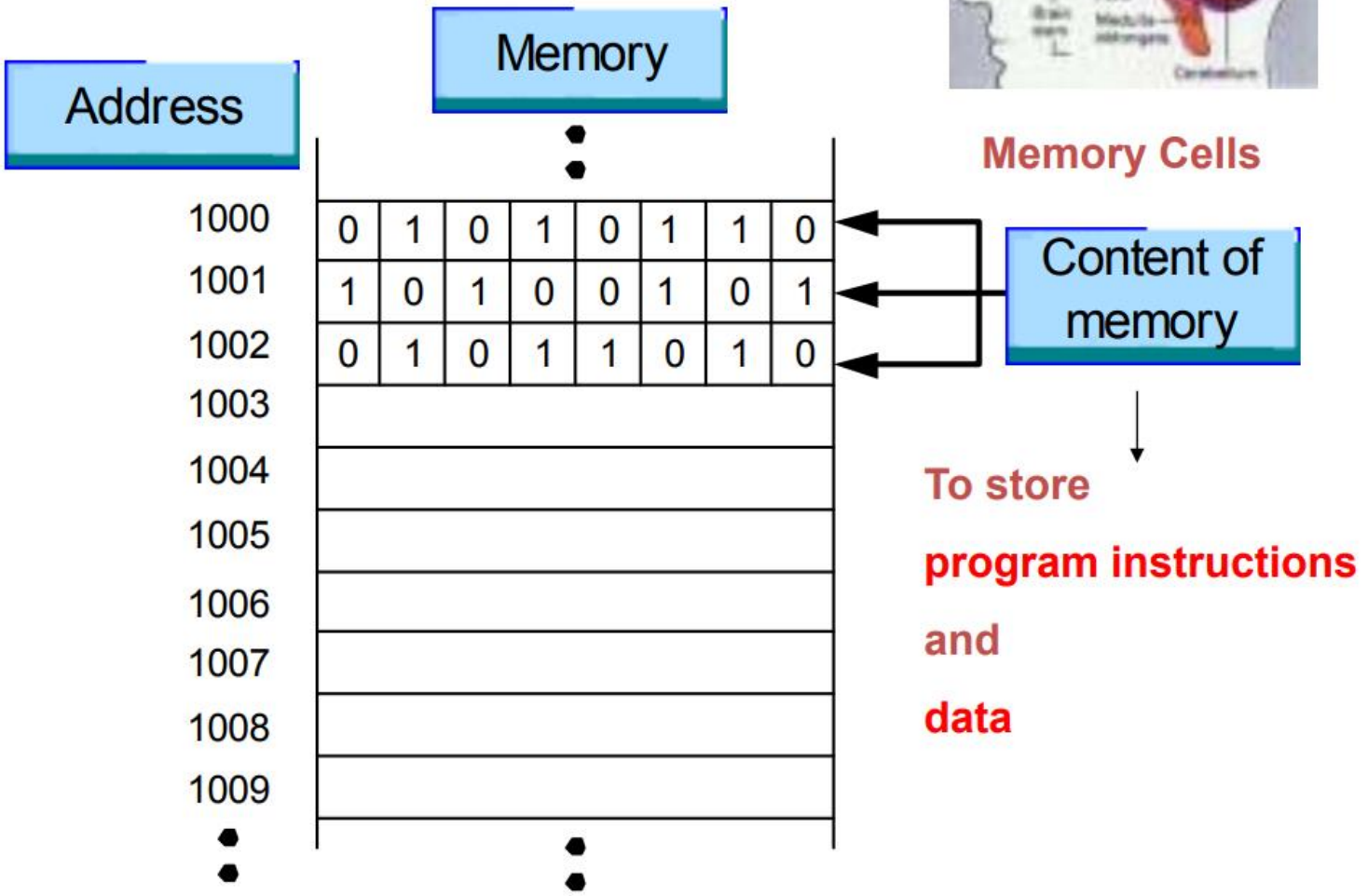
Lecture 2. Data Types and Operators

Learning Objectives

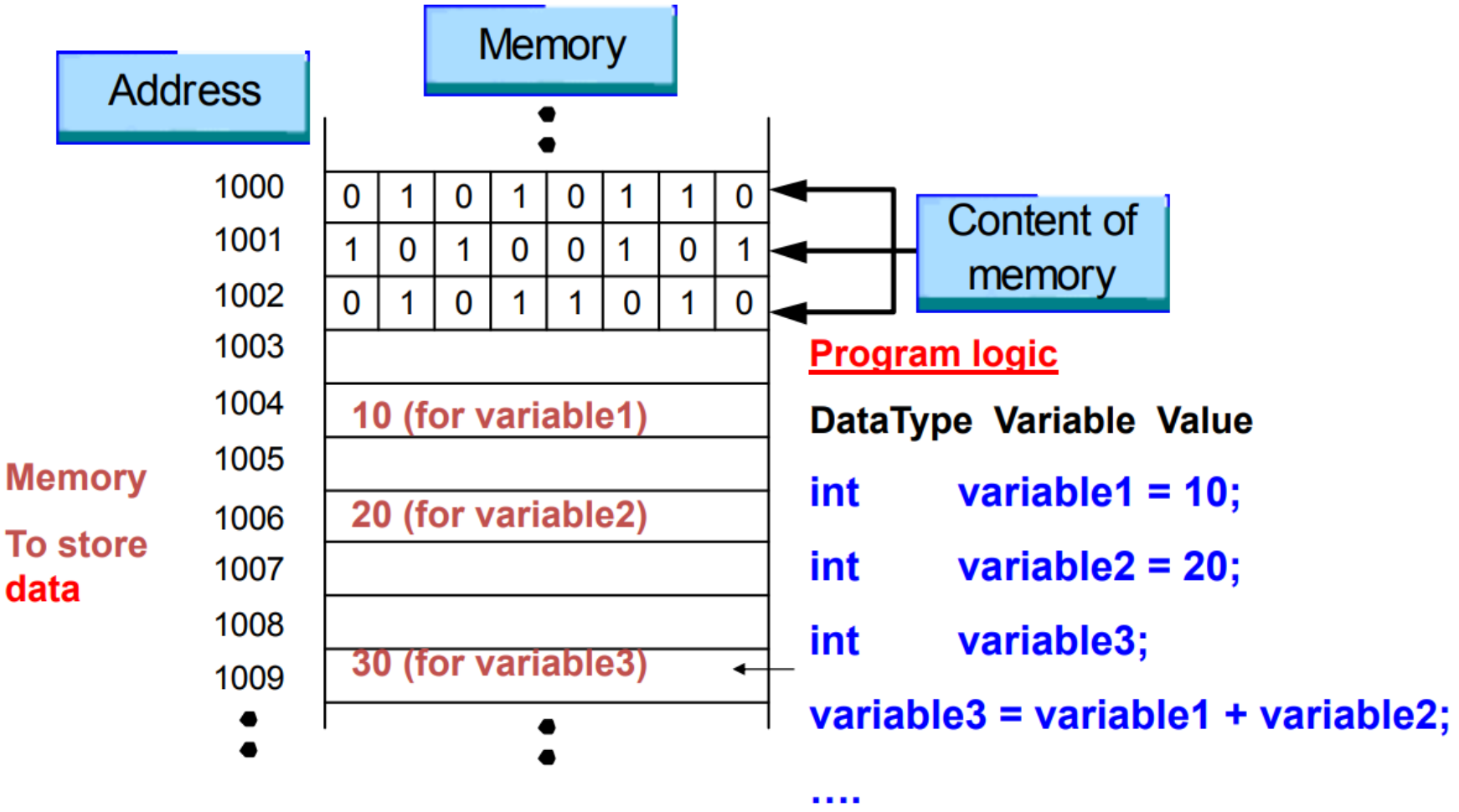
- At the end of this lecture, you will be able to understand:
  - Data Types
  - Constants
  - Variables
  - Operators
  - Expressions
  - Data Type Conversions

Computer Memory





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 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”

ASCII (American Standard Code for Information Interchange) 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**				

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 */
```

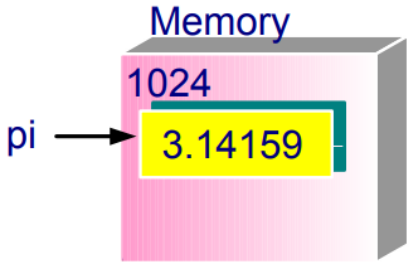
2) define a constant variable

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);
```

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

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:

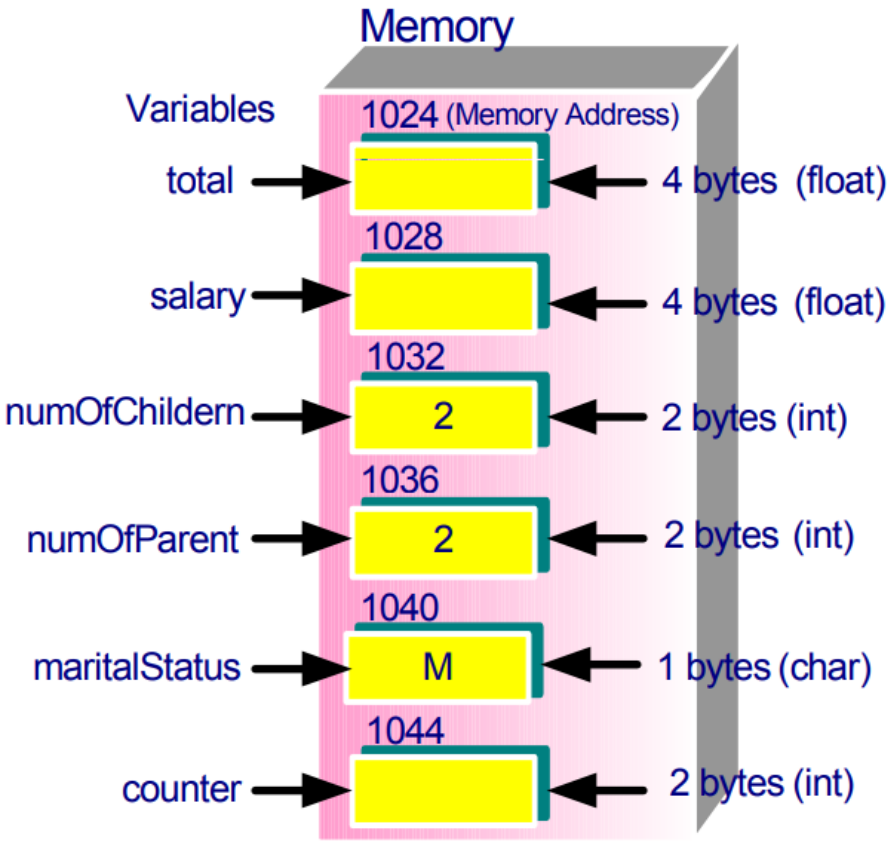
```
int count = 20;
float temperature, result;
temperature = 36.9;
```

Declaring Variables with Initialization

- **Example**

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

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



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

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

```
#include <stdio.h>
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

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.

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.

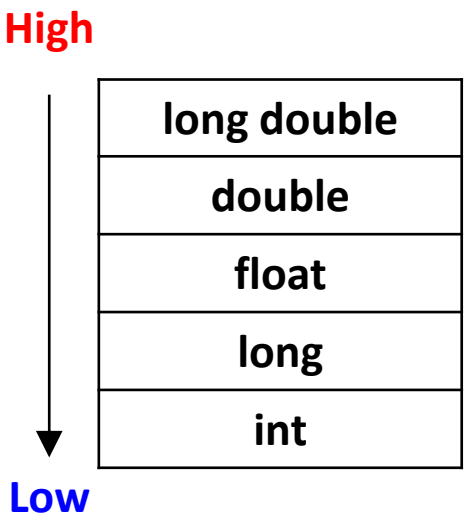
- e.g. `(int)2.7 + (int)3.5`

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`



Data Type Conversion

```
#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.

Precedence of Operators

- This decides the **order** of evaluation for arithmetic expressions containing several operands.
- The list of operators with **decreasing 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

decreasing

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`

Recap

- This lecture covers the following concepts:
  - Data Types
  - Literals (ASCII codes)
  - Constants
  - Variables
  - Operators
  - Expressions
  - Data Type Conversions