Syrian Private University

Introduction to Algorithms and Programming

Instructor: Dr. Mouhib Alnoukari

Problem Solving & Structured Programming

Problem Solving & Program Design

Two phases involved in the design of any program:

1. Problem Solving Phase
   - Define the problem.
   - Outline the solution.
   - Develop the outline into an algorithm.
   - Test the algorithm for correctness.
2. Implementation Phase
   - Code the algorithm using a specific programming language.
   - Run the program on the computer.
   - Document and maintain the program.

Structured Programming

Structured Programming Concept:

- Structured programming techniques assist the programmer in writing effective error free programs.

The elements of structured of programming include:

1. Top-down development
2. Modular design

Structure Programming Theorem

It is possible to write any computer program by using only three (3) basic control structures, namely:

1. Sequential
2. Selection (if-then-else)
3. Repetition (looping, DoWhile)

Algorithm

Algorithm

An algorithm is a sequence of precise instructions for solving a problem in a finite amount of time.

Algorithm Properties

- It must be precise and unambiguous.
- It must give the correct solution in all cases.
- It must eventually end.

Developing an Algorithm

- Understand the problem

  (Do problem by hand. Note the steps)

- Devise a plan

  (look for familiarity and patterns)

- Carry out the plan (trace)

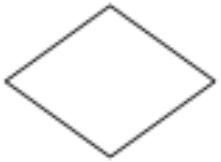
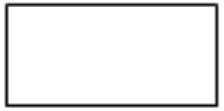
- Review the plan (refinement)

Understanding the Algorithm

Possibly the simplest and easiest method to understand the steps in an algorithm, is by using the flowchart method. This algorithm is composed of block symbols to represent each step in the solution process as well as the directed paths of each step.

Understanding the Algorithm

The most common block symbols are:

Symbol	Representation		Symbol	Representation
	Start/Stop			Decision
	Process			Connector
	Input/Output			Flow Direction

Understanding the Algorithm

Problem Example:

Find the average of a given set of numbers.

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

1. Understanding the problem

   (i) Write down some numbers on paper and find the average manually, noting each step carefully.

   e.g. Given a list say: 5, 3, 25, 0, 9

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

1. Understanding the problem
   (i) Write down some numbers on paper
   (ii) Count numbers | i.e. How many? 5
   (iii) Add them up | i.e. 5 + 3 + 25 + 0 + 9 = 42
   (iv) Divide result by numbers counted | i.e. 42/5 = 8.4

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

2. Devise a plan:

   Make note of NOT what you did in steps (i) through (iv) above, but HOW you did it.

   In doing so, you will begin to develop the algorithm.

Understanding the Algorithm

Problem Example

For Example:

How do we count the numbers?

Starting at 0 we set our COUNTER to 0.

Look at first number and add 1 to COUNTER.

Look at 2nd number and add 1 to COUNTER.

…and so on,

until we reach the end of the list.

Understanding the Algorithm

Problem Example

For Example:

How do we add numbers?

Let SUM be the sum of numbers in list.

i.e. Set SUM to 0

Look at 1st number and add number to SUM.

Look at 2nd number and add number to SUM.

…and so on,

until we reach end of list.

Understanding the Algorithm

Problem Example

For Example:

How do we compute the average?

Let AVE be the average.

$$\text{then AVE} = \frac{\text{total sum of items}}{\text{number of items}}$$

$$= \frac{\text{SUM}}{\text{COUNTER}}$$

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

3. Identify patterns, repetitions and familiar tasks.

Familiarity:	Unknown number of items? i.e. n item
Patterns :	look at each number in the list
Repetitions:	Look at a number Add number to sum Add 1 to counter

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

4. Carry out the plan
   - Check each step
   - Consider special cases
   - Check result
   - Check boundary conditions:
   - e.g. What if the list is empty?
     - Division by 0?
     - Are all data values within specified range?

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

5. Review the plan:

   Can you derive the result differently?

   Can you make the solution more general?

   Can you use the solution or method for another problem?

   e.g. average temperature or average grades

Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

5. Review the plan:

   Can you derive the result differently?

   Can you make the solution more general?

   Can you use the solution or method for another problem?

   e.g. average temperature or average grades

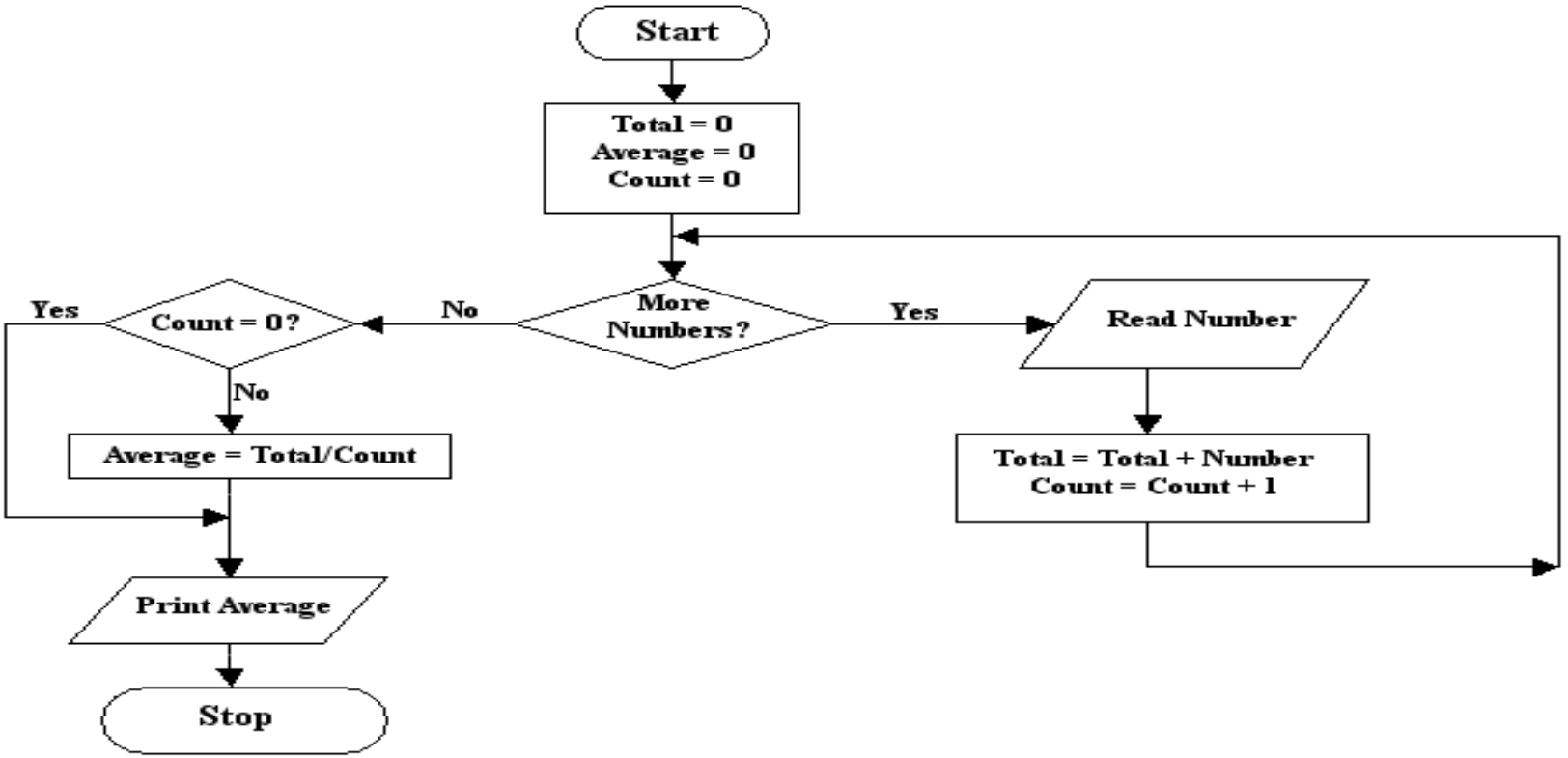
Understanding the Algorithm

Problem Example

Solution Steps - Proceed as follows:

Example

A flowchart representation of the algorithm for the above problem can be as follows:



C Language

Basic Data Type

Using C Programming Language

C Basic Data Types

In C, data type categorized as:

1. Primitive Types : `char`, `short`, `int`, `float`, `double` and `long`.
2. User Defined Types – `struct`, `union`, `enum` and `typedef`.
3. Derived Types – `pointer`, `array` and `function pointer`.

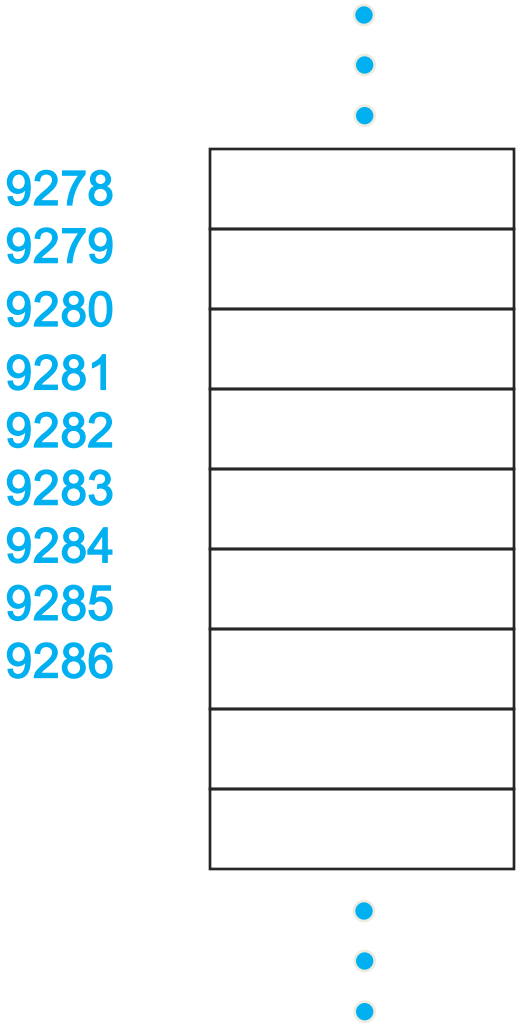
C Programming

Numeric Primitive Data Types

- The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	Storage	Min Value	Max Value
`char`	8 bits	-128	127
`short`	16 bits	-32,768	32,767
`int`	32 bits	-2,147,483,648	2,147,483,647
`long`	64 bits	$< -9 \times 10^{18}$	$> 9 \times 10^{18}$
`float`	32 bits	$+/- 3.4 \times 10^{38}$ with 7 significant digits	
`double`	64 bits	$+/- 1.7 \times 10^{308}$ with 15 significant digits	

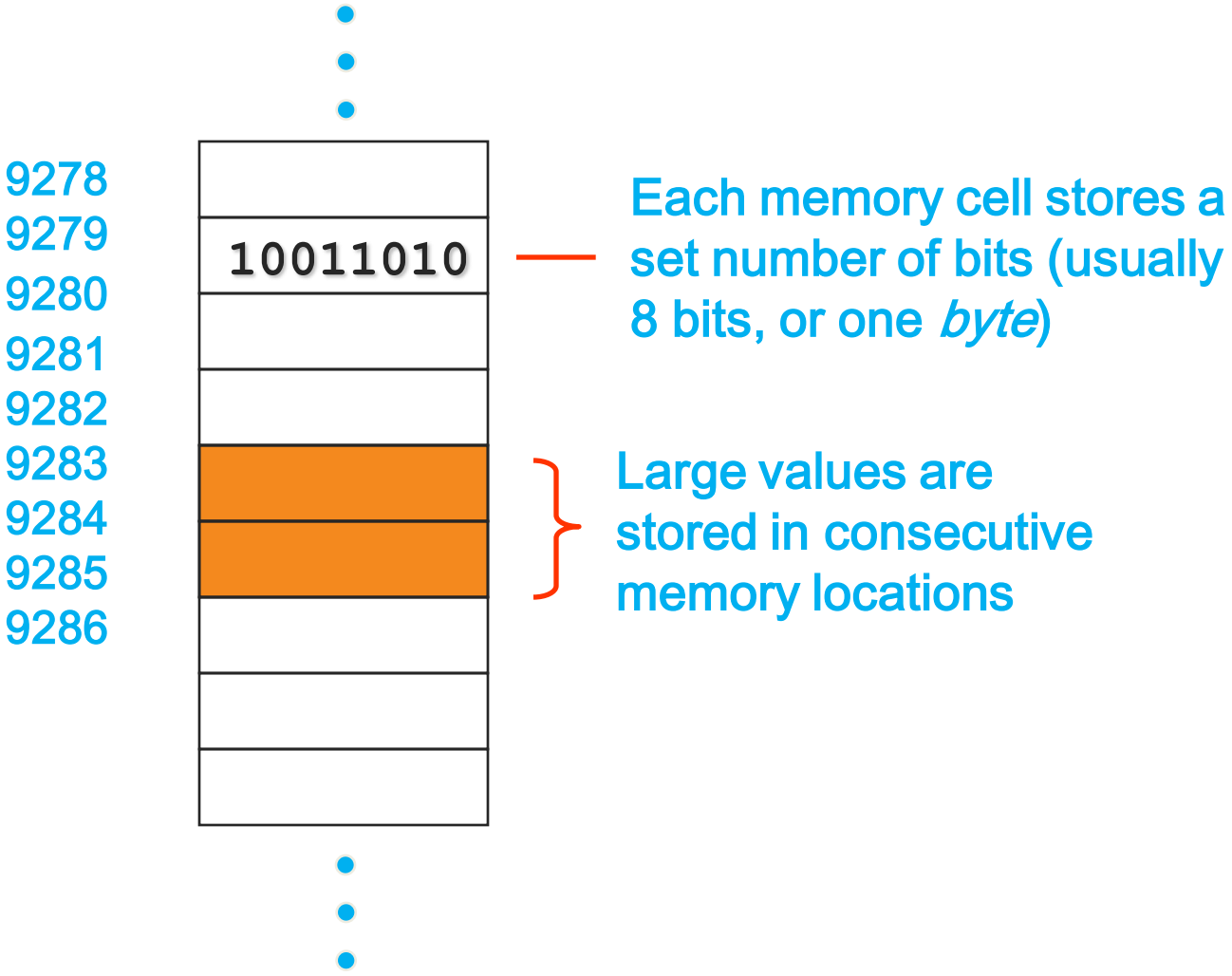
Computer Memory



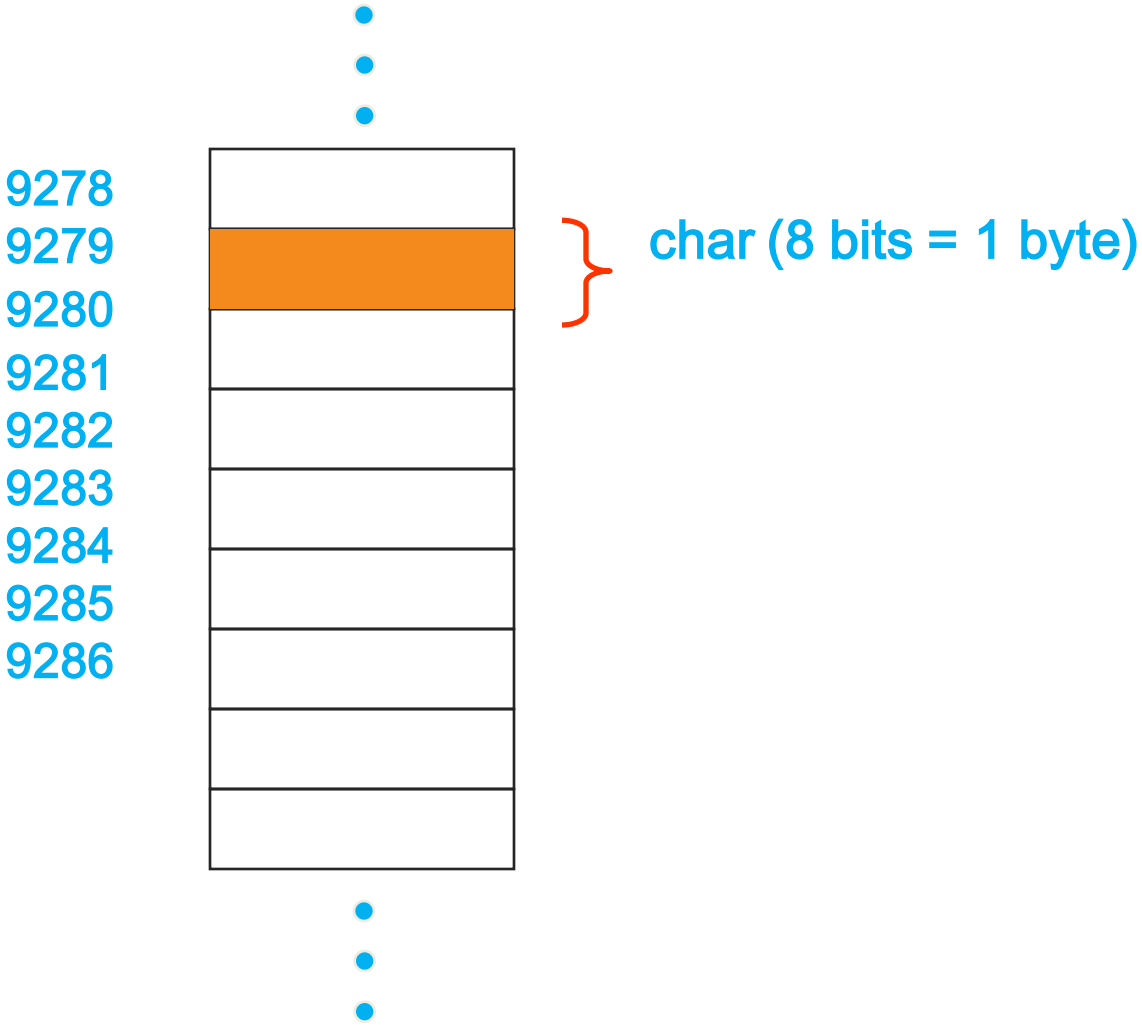
Main memory is divided into many memory locations (or *cells*)

Each memory cell has a numeric *address*, which uniquely identifies it

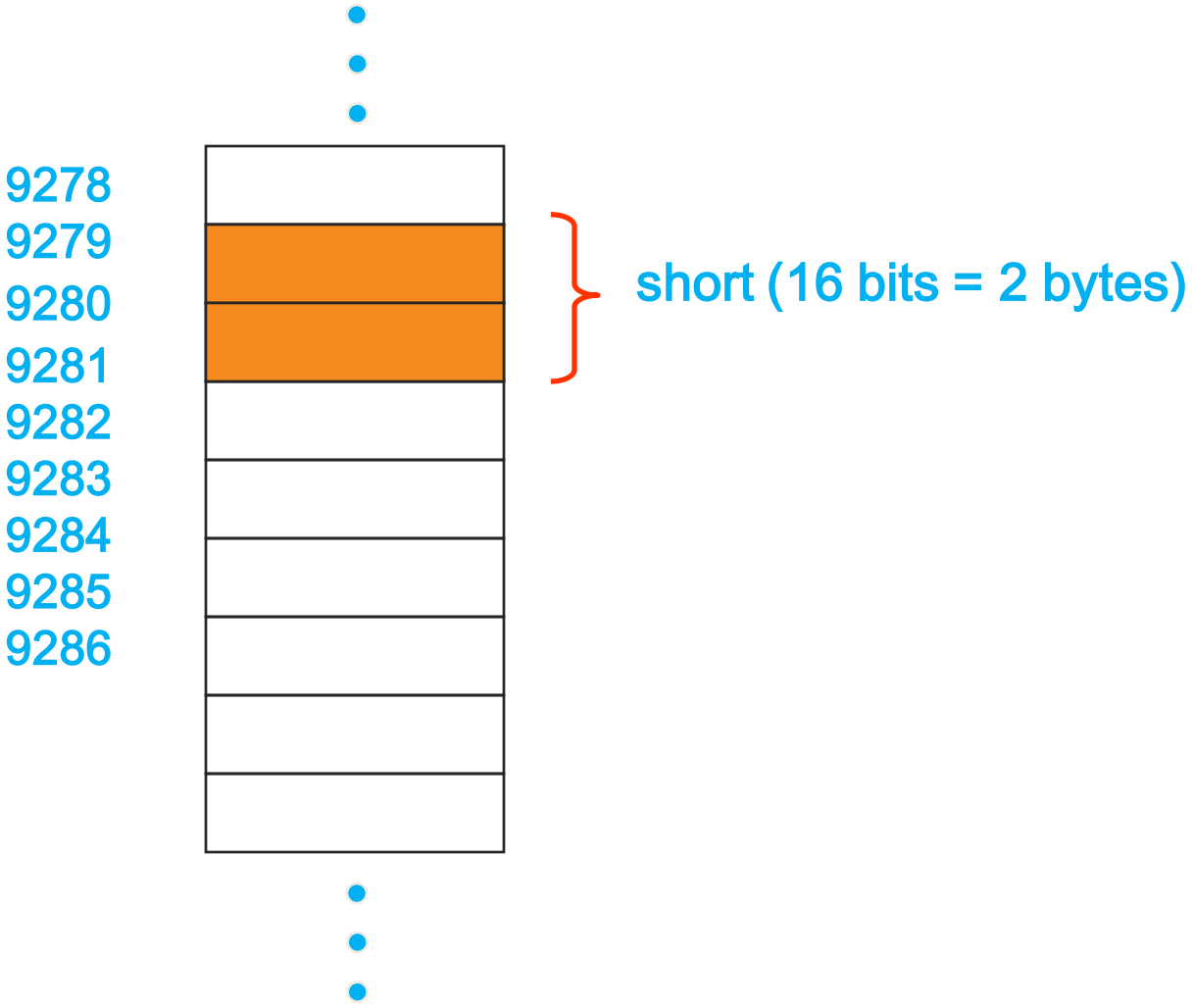
Storing Information



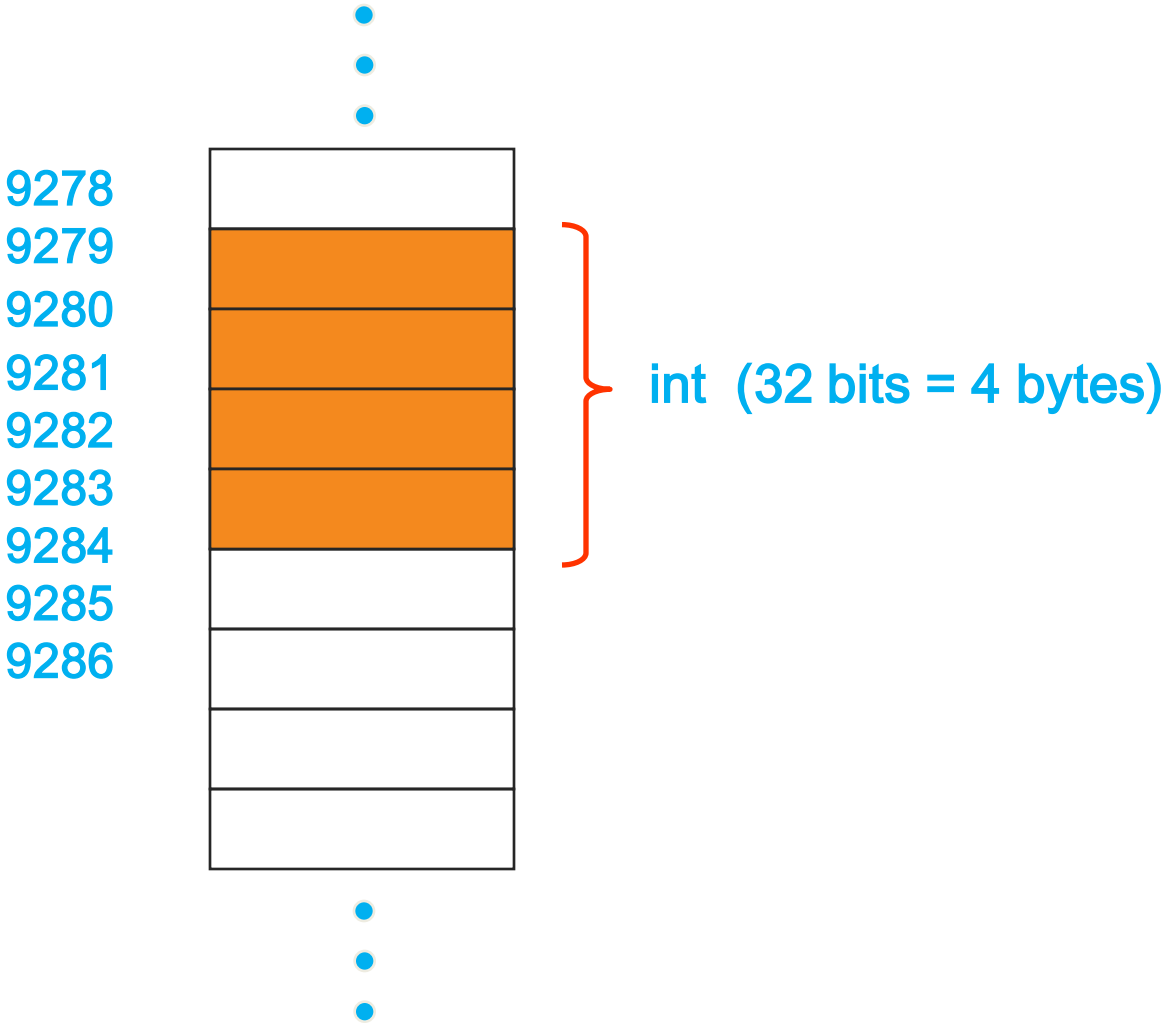
Storing a Char



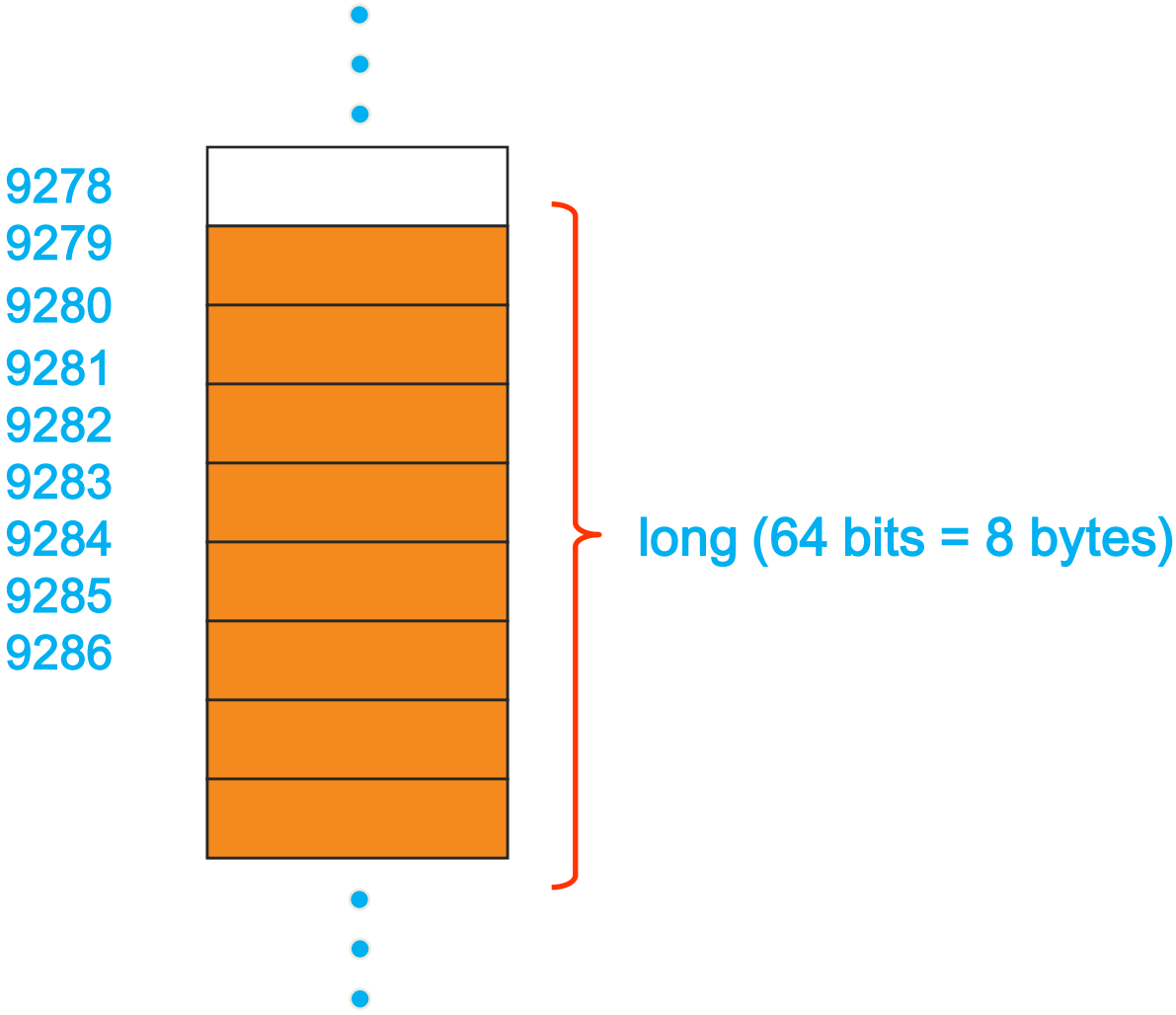
Storing a Short



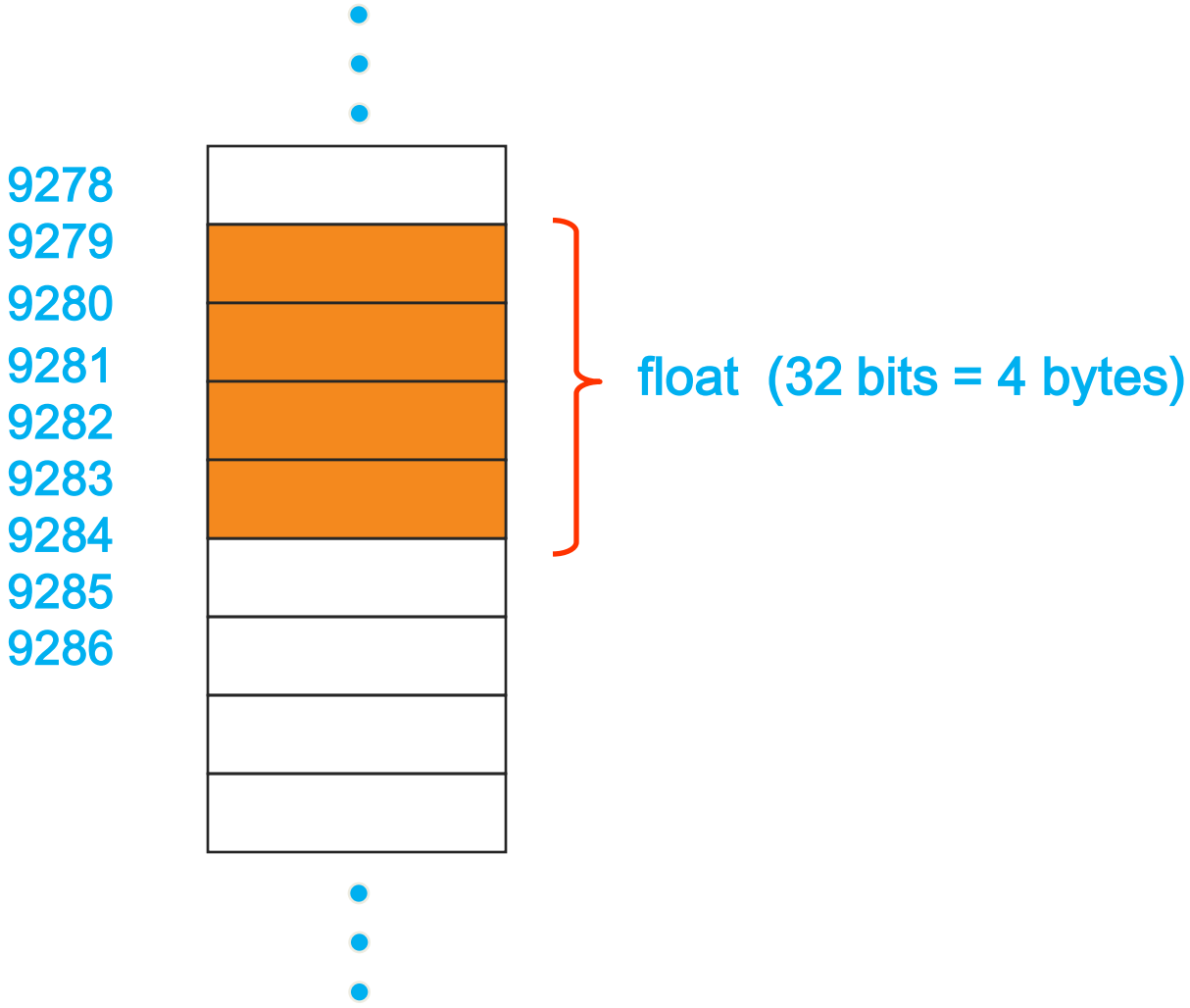
Storing an int



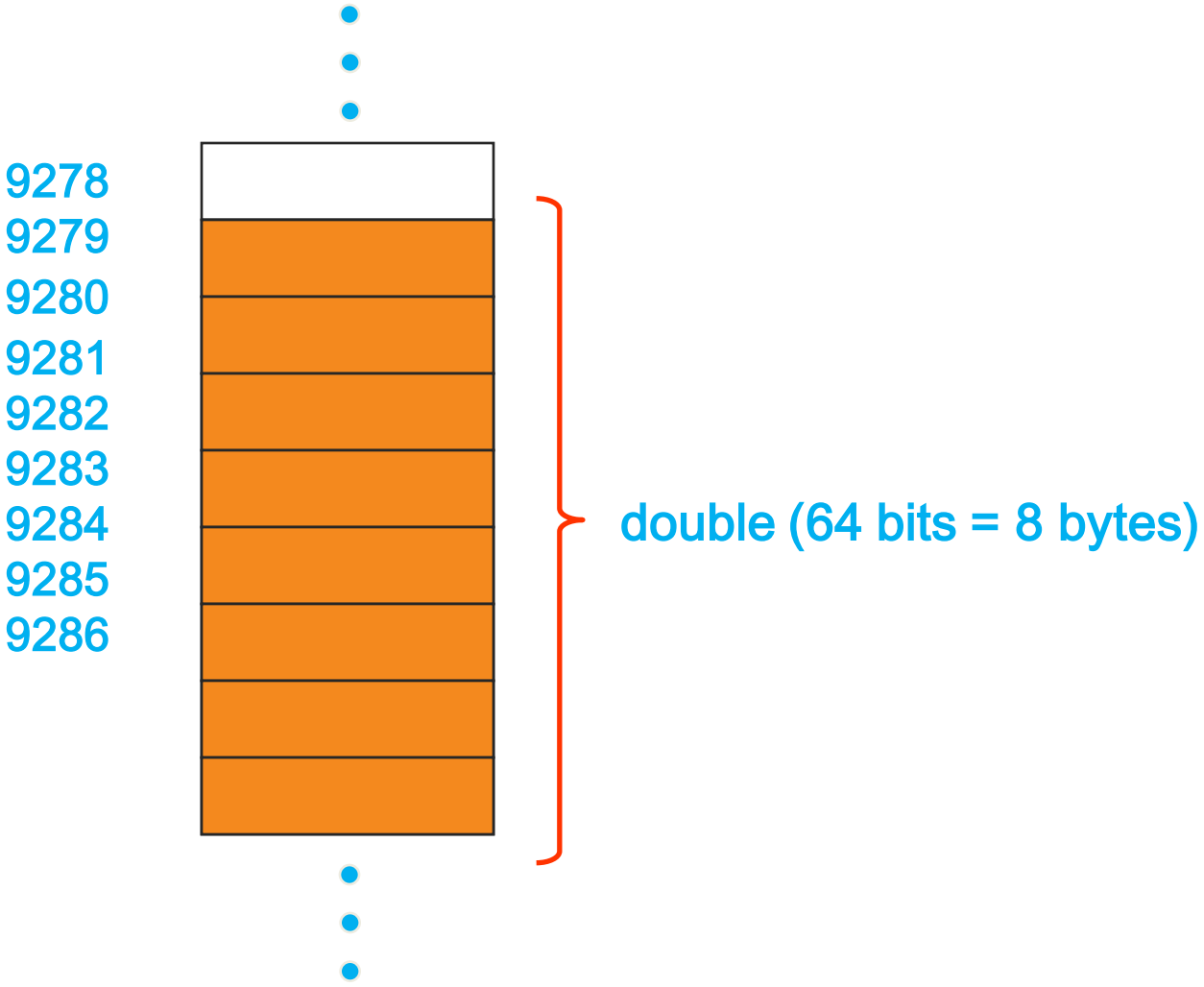
Storing a long



Storing a float



Storing a doable



Characters

- A `char` variable stores a single character
- Character literals are delimited by single quotes:

```
'a'   'X'   '7'   '$'   ','   '\n'
```

- Example declarations:

```
char topGrade = 'A';

char terminator = ';', separator = ' ';
```

Character Strings

- A string of characters can be represented as a *string literal* by putting double quotes around the text:
- Examples:

```
"This is a string literal."
"123 Main Street"
"X"
```

- Note the distinction between a primitive character variable, which holds only one character, and a `String` object, which can hold multiple characters

C Language

Printf() & Scanf() Function

Printf ()

- printf(“format string”, variable1, variable2, …);
- printf(“For int use %d”, myInteger);
- printff(“For float use %f”, myFloat);
- printf(“For double use %lf”, myDouble);
- printf(“For float or double %g”, myF_or_D);
- printf(“int=%d double %lf”, myInteger, myDouble);

Scanf()

- scanf(“format string”, &variable1, &variable2, …);
- scanf(“%d”, &myInteger);
- scanf(“%f”, &myFloat);
- scanf(“%lf”, &myDouble);
- scanf(“%d%f”, &myInteger, &myFloat);

Escape Sequences

- What if we wanted to print a the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

```
printf ("I said "Hello" to you.");
```

- An *escape sequence* is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\)

```
printf ("I said \"Hello\" to you.");
```

Escape Sequences

- Some C escape sequences:

Escape Sequence	Meaning
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\a`	beep
`\"`	double quote
`\'`	single quote
`\\`	backslash

C Language

Exercises

Char Type Exercise

```
#include <stdio.h>

int main(void)
{
        char chVar = 'A';
        signed char chSignedVar = 'B';
        unsigned char chUnSignedVar = 'C';

        printf("char value is %c (%d), its size is %d byte.\n", chVar, chVar, sizeof(char));
        printf("signed char value is %c (%d), its size is %d byte.\n", chSignedVar,
        chSignedVar, sizeof(signed char));
        printf("unsigned char value is %c (%d), its size is %d byte.\n", chUnSignedVar,
        chUnSignedVar, sizeof(unsigned char));

        return 0;
}
```

Long Int Type Exercise

```
#include <stdio.h>
int main(void)
{
        // declare and initialize variables
        long nNumLg = -10000;
        long int nNumLgInt = 200000;
        signed long nNumSignedLg = -3000000;
        signed long int nNumSignedLgInt = 40000000;
        unsigned long nNumUnSignedLg = 500000000;
        float nNumFloat = (float)6.71234;
        double nNumDouble = 789.652341;
        long double nNumLgDouble = 9796.6174;
        // print those values
        printf("long value is %ld with %d bytes in size.\n", nNumLg, sizeof(long));
        printf("long int value is %ld with %d bytes in size.\n", nNumLgInt, sizeof(long int));
        printf("signed long value is %ld with %d bytes in size.\n", nNumSignedLg, sizeof(signed long));
        printf("signed long int value is %ld with %d bytes in size.\n", nNumSignedLgInt, sizeof(signed long int));
        printf("unsigned long value is %lu with %d bytes in size.\n", nNumUnSignedLg, sizeof(unsigned long));
        printf("\nfloat value is %f with %d bytes in size.\n", nNumFloat, sizeof(float));
        printf("double value is %f with %d bytes in size.\n", nNumDouble, sizeof(double));
        printf("long double value is %lf with %d bytes in size.\n", nNumLgDouble, sizeof(long double));
        return 0;
}
```

Ptr Diff Exercise

```
#include <stdio.h>

int main(void)
{
        // declare two pointer variables
        int *pPointA = NULL, *pPointB = NULL;
        // two integer variables
        int nNumA, nNumB;
        // and variable to hold the result of subtracting
        // two pointers
        ptrdiff_t ptDiffVar;

        // assign integers to variables
        nNumA = 24;
        nNumB = 45;

        // let those pointers point to those variables
        pPointA = &nNumA;
        pPointB = &nNumB;

        // subtract those pointers and store at ptDiffVar
        ptDiffVar = pPointA - pPointB;
>>>>
```

Ptr Diff Exercise

```
    // print some info
    printf("Value of  of pPointA is %i and pPointB is %i\n", *pPointA, *pPointB);
    printf("Address of pPointA is %X and pPointB is %X\n", pPointA, pPointB);
    printf("The size of ptrdiff_t is %d bytes\n", sizeof(ptDiffVar));
    printf("The difference between two pointers is %X (%d)\n", ptDiffVar, ptDiffVar);

    ptDiffVar = pPointB - pPointA;
    printf("The difference between two pointers is %X (%d)\n", ptDiffVar, ptDiffVar);

    return 0;
}
```

Common Bugs (printf, scanf)

- Using & in a printf function call.
  printf("For int use %d", &myInteger); // wrong

- Using the wrong string in printf
  printf("This is a float %d", myFloat); // use %f not %d

- Not using & in a scanf() function call.
  scanf("%d", myInteger); // Wrong

- Using the wrong string in scanf()
  scanf("%d", &myFloat); // wrong; use %f instead of %d