# C++ Programming Essentials

## Unit 1: Sequential Programming

CHAPTER 2: C++ SYNTAX AND SEMANTICS, AND THE PROGRAM DEVELOPMENT PROCESS

DR. ERIC CHOU

IEEE SENIOR MEMBER

LECTURE 1

# C++ Program Structure

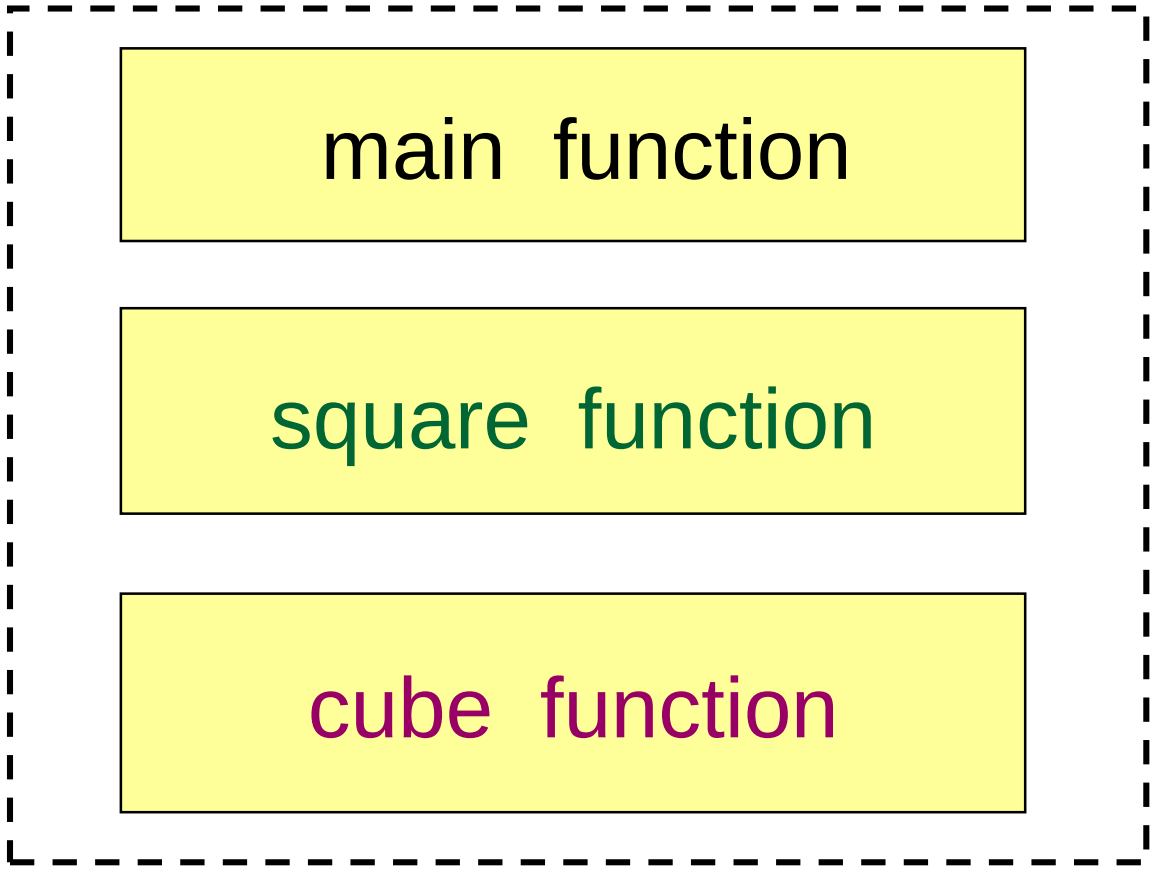
# Chapter 2 Topics

- Programs Composed of Several Functions
- Syntax Templates
- Legal C++ Identifiers
- Assigning Values to Variables
- Declaring Named Constants
- String Concatenation
- Output Statements
- C++ Program Comments

# A C++ program is a collection of one or more functions

- there must be a function called main( )
- execution always begins with the first statement in function main( )
- any other functions in your program are subprograms and are not executed until they are called

# Program With Several Functions



# Program With Three Functions

## Demo Program: **square.cpp**

```
#include <iostream>

int Square( int );       // declares these two
int Cube( int );        // value-returning functions

using namespace std ;

int main( )
{
    cout << "The square of 27 is "
      << Square(27) << endl;  // function call

    cout << "The cube of 27 is "
         << Cube(27) << endl;        // function call
    return 0;
}
```

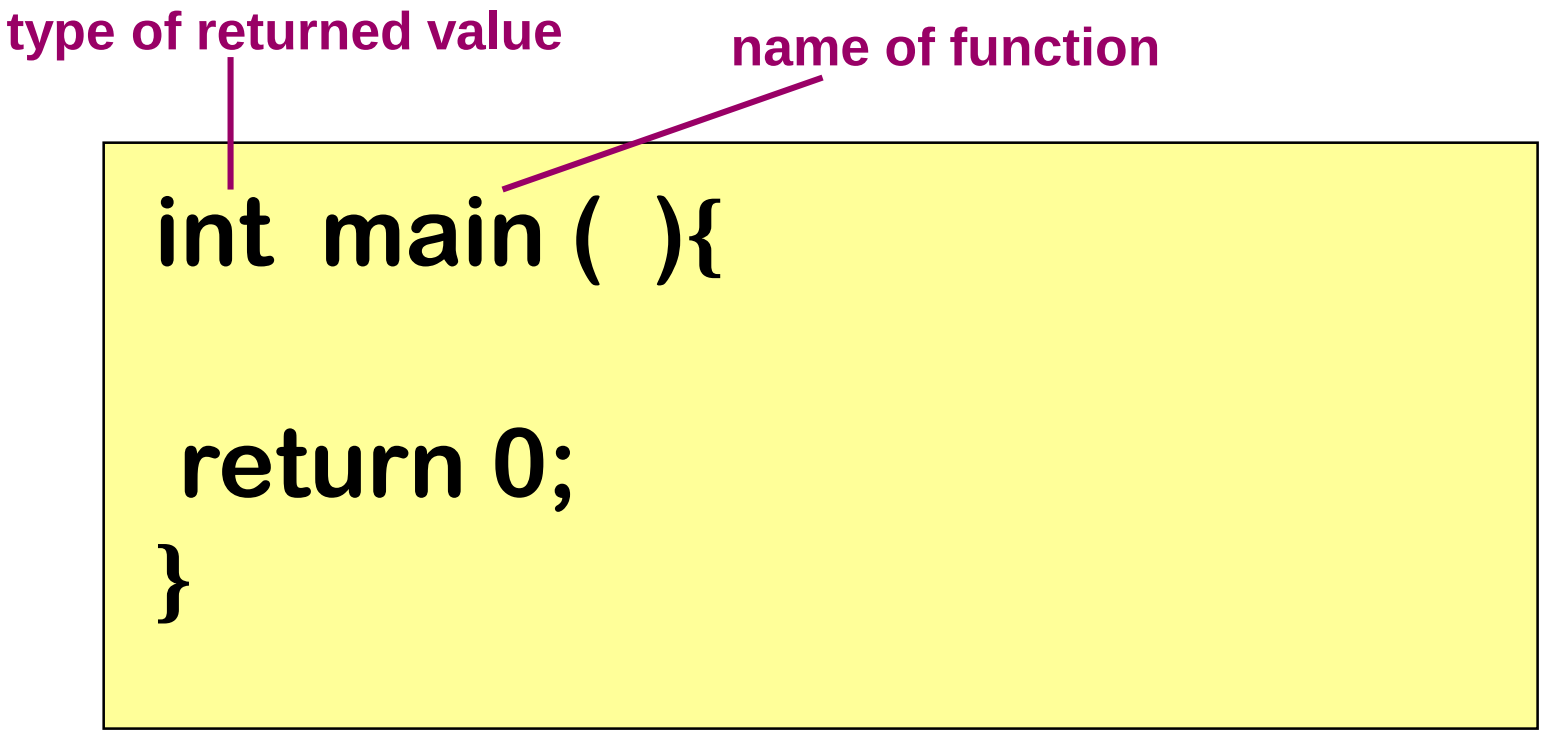
# Rest of Program

```
int Square( int n )
{
      return n * n;
}


int Cube( int n )
{
     return n * n * n;
}
```

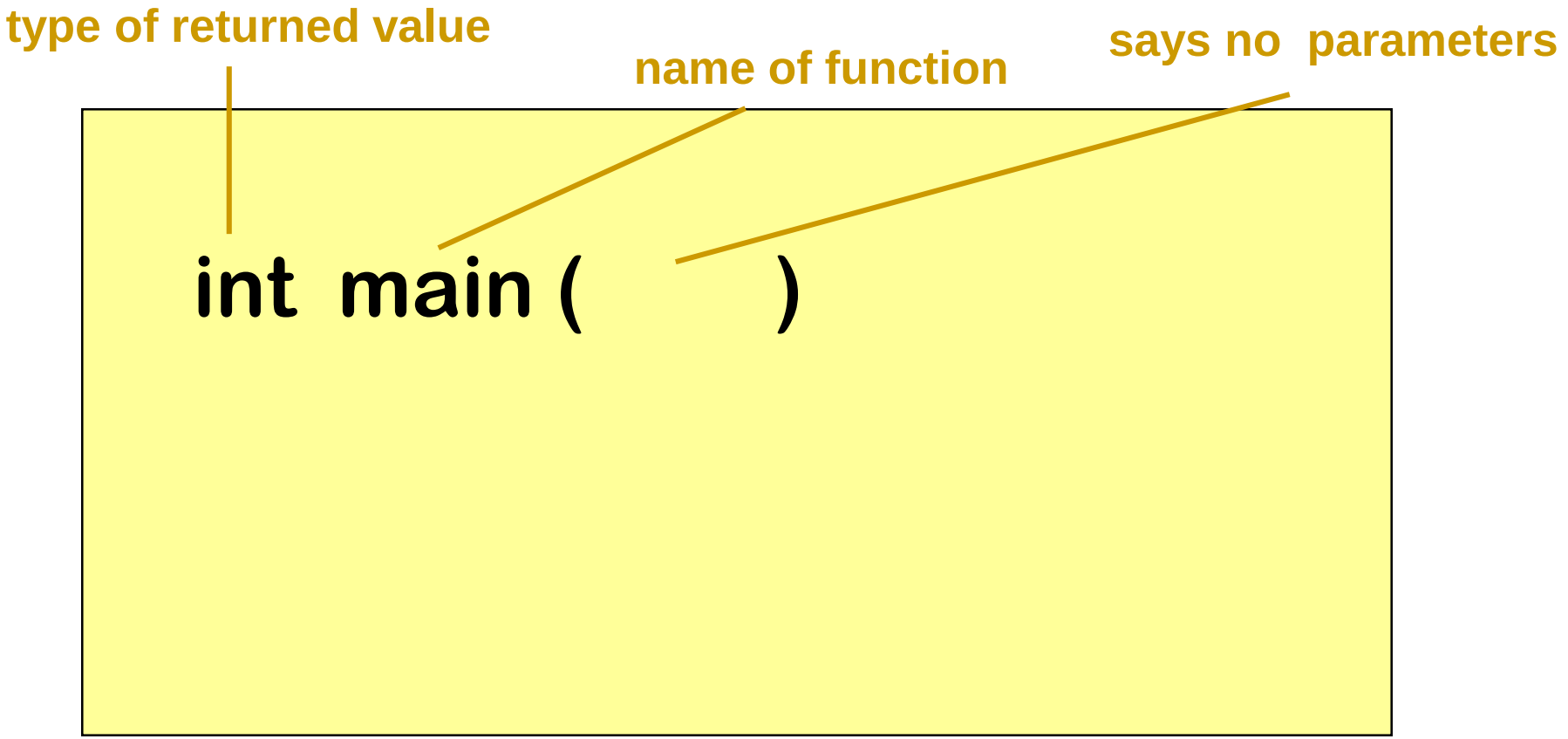
# Output of program

```
The square of 27 is 729
The cube of 27 is 19683
```

# Shortest C++ Program



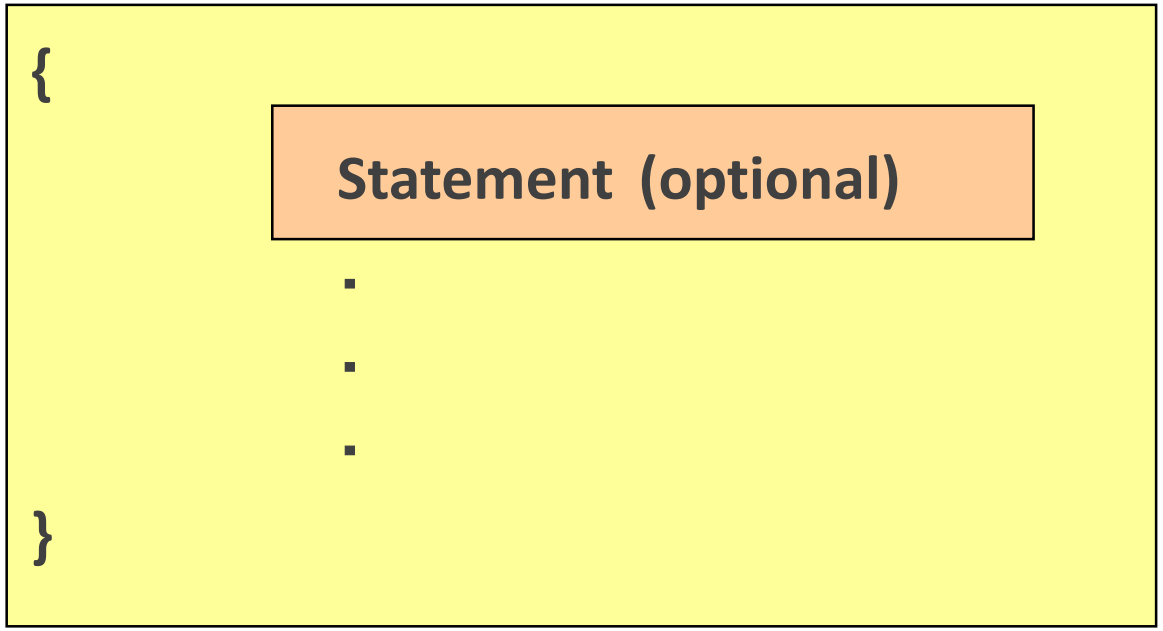
# What is in a heading?



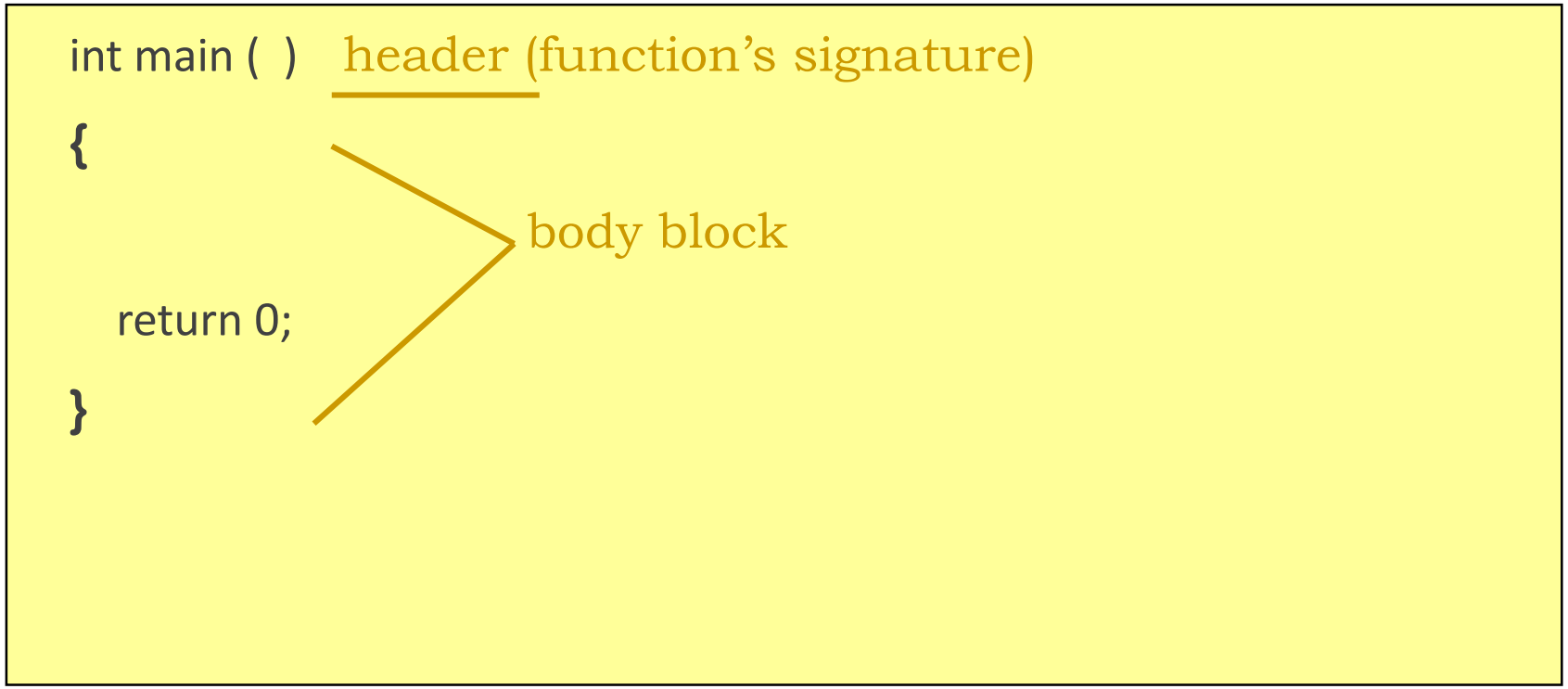
# Block (Compound Statement)

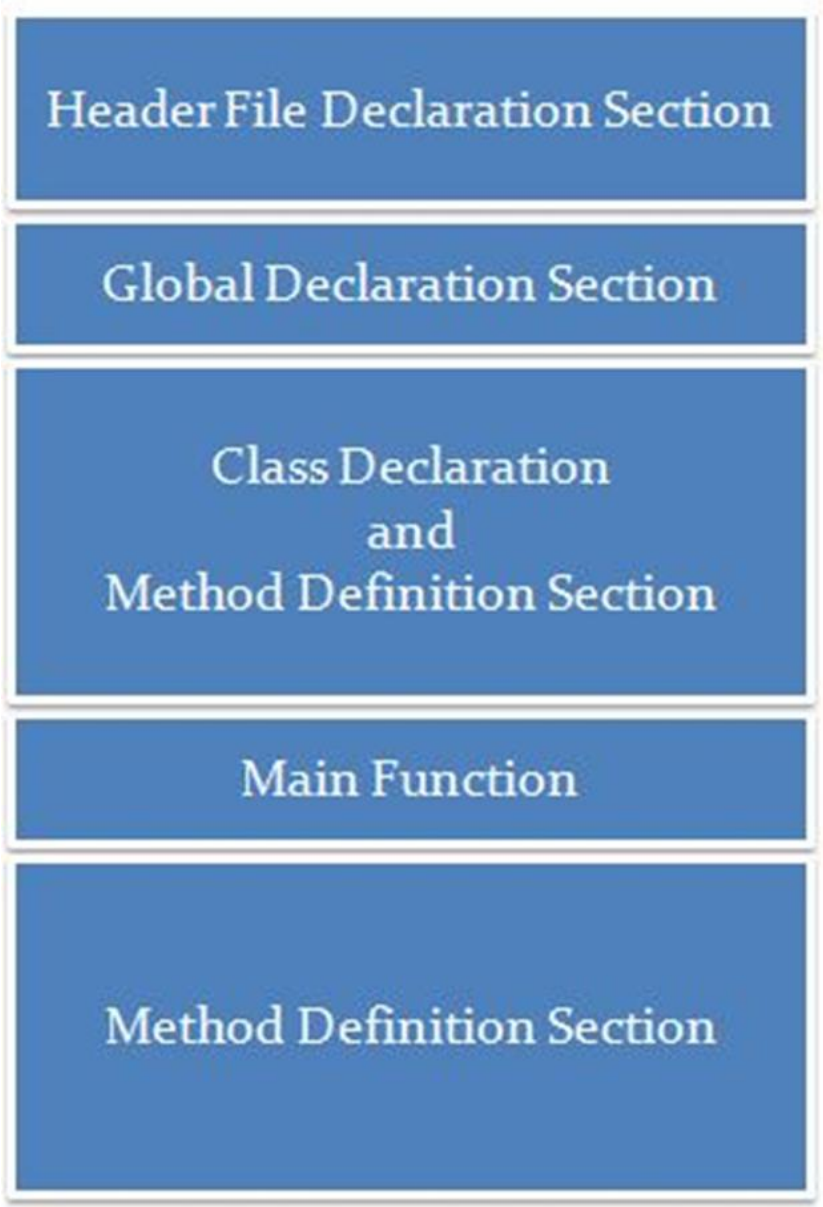
- a block is a sequence of zero or more statements enclosed by a pair of curly braces { }

**SYNTAX**



# Every C++ function has 2 parts




Header File Declaration Section
Global Declaration Section
Class Declaration
and
Method Definition Section
Main Function
Method Definition Section

LECTURE 2

# C++ Identifiers

# What is an Identifier?

- An identifier is the name used for a data object (a variable or a constant), or for a function,  in a C++ program.
- C++ is a case-sensitive language.
- using meaningful identifiers is a good programming practice

# Identifiers

- an identifier must start with a letter or underscore, and be followed by zero or more letters

  (A-Z, a-z), digits (0-9), or underscores

**VALID**

**age_of_dog** **taxRateY2K**

**PrintHeading** **ageOfHorse**

**NOT VALID (Why?)**

**age#** **2000TaxRate** **Age-Of-Cat**

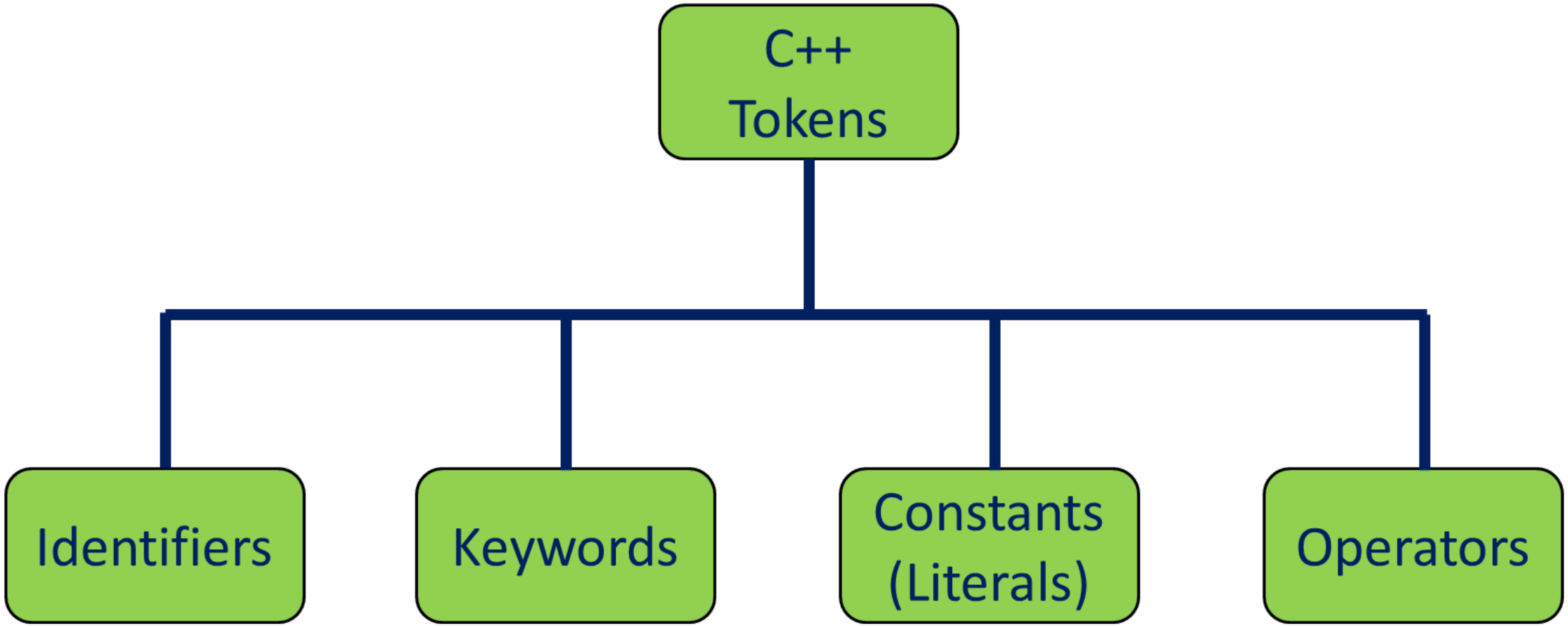
# More About Identifiers

- some C++ compilers recognize only the first 32 characters of an identifier as significant
- then these identifiers are considered the same:

```
age_Of_This_Old_Rhinoceros_At_My_Zoo
age_Of_This_Old_Rhinoceros_At_My_Safari
```

- consider these:

```
Age_Of_This_Old_Rhinoceros_At_My_Zoo
age_Of_This_Old_Rhinoceros_At_My_Zoo
```


C++
Tokens
Identifiers
Keywords
Constants
(Literals)
Operators

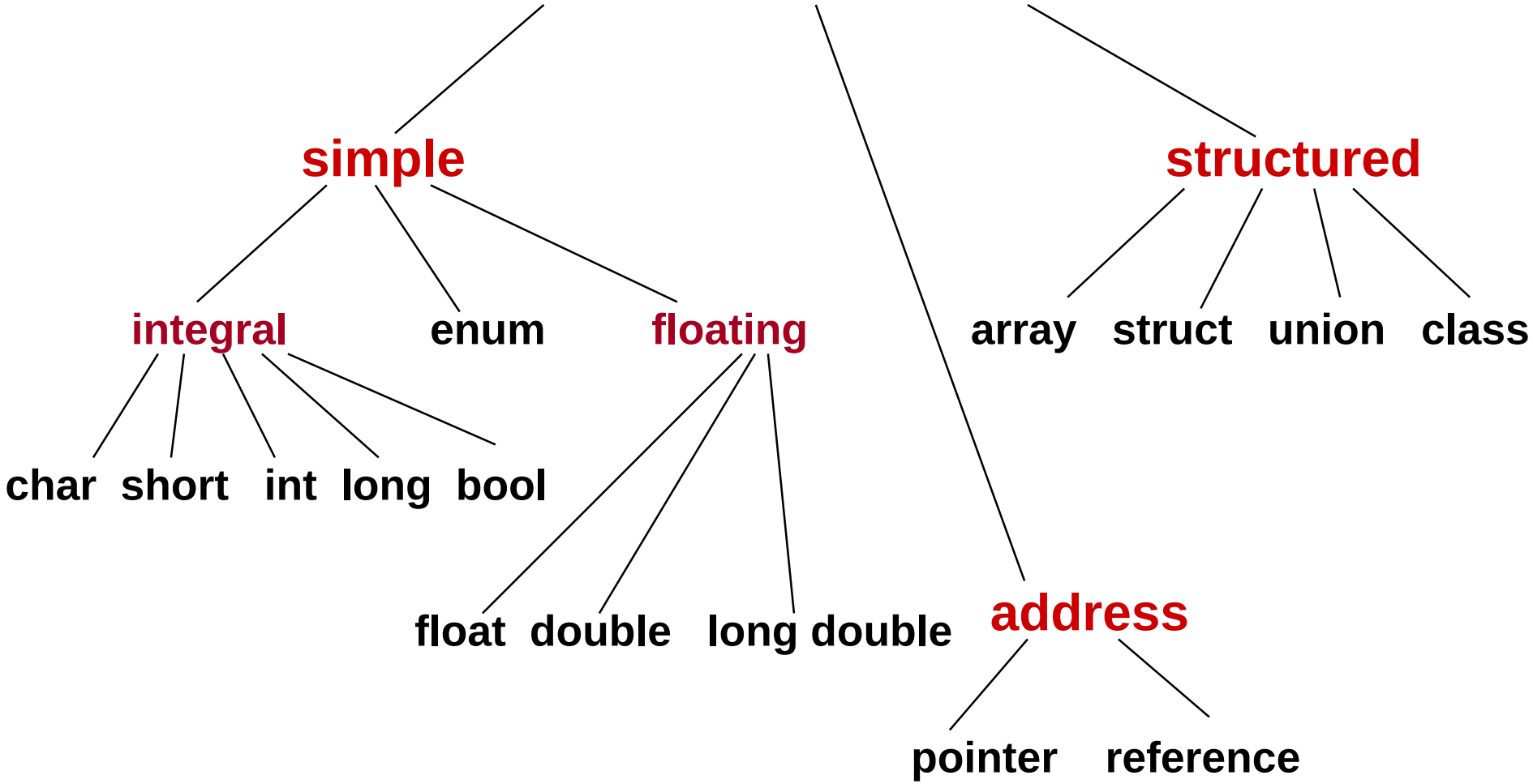
# C++ Reserved Keywords

| | | | |
|---|---|---|---|
| alignas | decltype | namespace | struct |
| alignof | default | new | switch |
| and | delete | noexcept | template |
| and_eq | double | not | this |
| asm | do | not_eq | thread_local |
| auto | dynamic_cast | nullptr | throw |
| bitand | else | operator | true |
| bitor | enum | or | try |
| bool | explicit | or_eq | typedef |
| break | export | private | typeid |
| case | extern | protected | typename |
| catch | false | public | union |
| char | float | register | unsigned |
| char16_t | for | reinterpret_cast | using |
| char32_t | friend | return | virtual |
| class | goto | short | void |
| compl | if | signed | volatile |
| const | inline | sizeof | wchar_t |
| constexpr | int | static | while |
| const_cast | long | static_assert | xor |
| continue | mutable | static_cast | xor_eq |

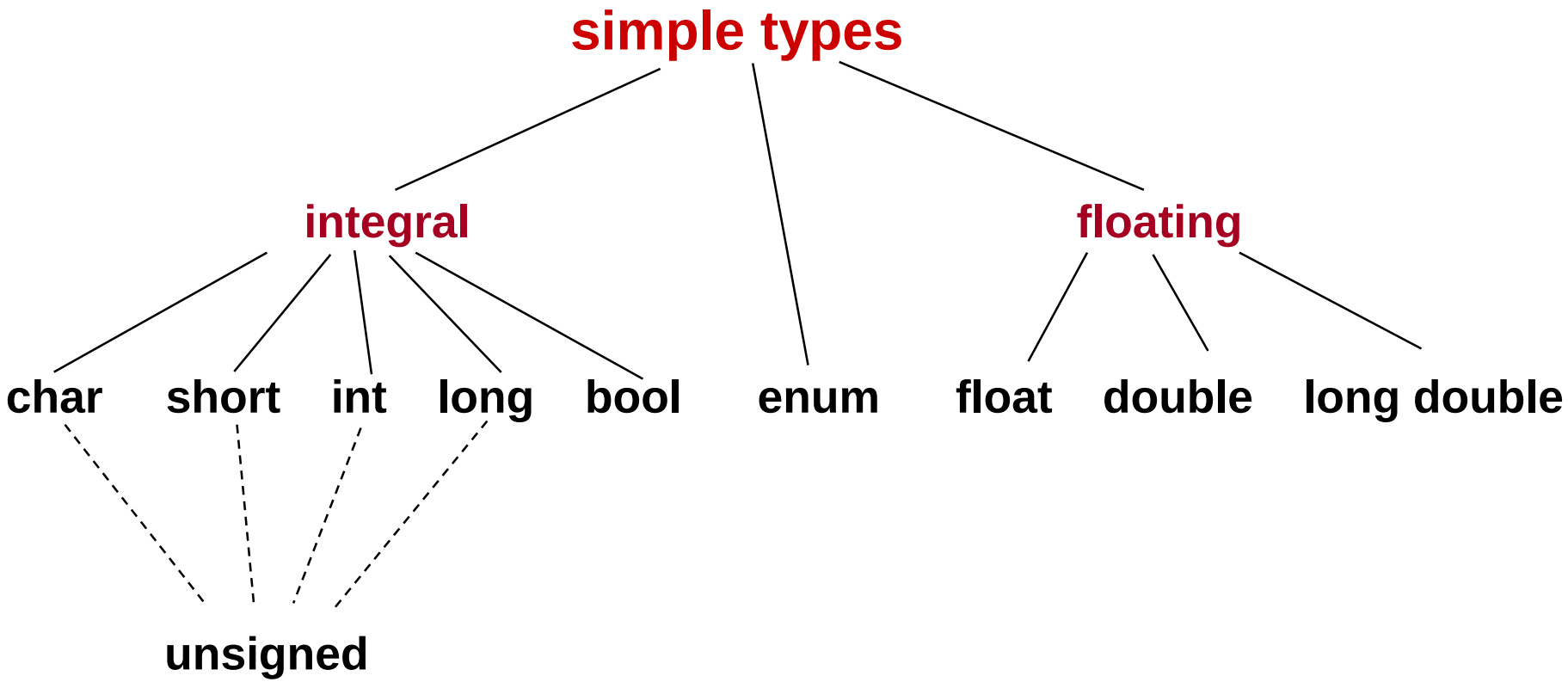
LECTURE 3

# C++ Variables

# C++ Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

# C++ Simple Data Types



# Standard Data Types in C++

Integral Types

- represent whole numbers and their negatives
- declared as int,  short, or long

Floating Types

- represent real numbers with a decimal point
- declared as float, or double

Character Types

- represent single characters
- declared as char

header

# <cstdint> (stdint.h)

### Integer types

This header defines a set of integral type aliases with specific width requirements, along with macros specifying their limits and macro functions to create values of these types.

## Types

The following are typedefs of *fundamental integral types* or *extended integral types*.

| signed type | unsigned type | description |
|---|---|---|
| `intmax_t` | `uintmax_t` | Integer type with the maximum width supported. |
| `int8_t` | `uint8_t` | Integer type with a width of exactly 8, 16, 32, or 64 bits.<br>For signed types, negative values are represented using 2's complement.<br>No padding bits.<br>**Optional:** These typedefs are not defined if no types with such characteristics exist.* |
| `int16_t` | `uint16_t` | |
| `int32_t` | `uint32_t` | |
| `int64_t` | `uint64_t` | |
| `int_least8_t` | `uint_least8_t` | Integer type with a minimum of 8, 16, 32, or 64 bits.<br>No other integer type exists with lesser size and at least the specified width. |
| `int_least16_t` | `uint_least16_t` | |
| `int_least32_t` | `uint_least32_t` | |
| `int_least64_t` | `uint_least64_t` | |
| `int_fast8_t` | `uint_fast8_t` | Integer type with a minimum of 8, 16, 32, or 64 bits.<br>At least as fast as any other integer type with at least the specified width. |
| `int_fast16_t` | `uint_fast16_t` | |
| `int_fast32_t` | `uint_fast32_t` | |
| `int_fast64_t` | `uint_fast64_t` | |
| `intptr_t` | `uintptr_t` | Integer type capable of holding a value converted from a `void` pointer and then be converted back to that type with a value that compares equal to the original pointer.<br>**Optional:** These typedefs may not be defined in some library implementations.* |

# Samples of C++ Data Values

- int sample values

  **4578 -4578 0**

- float sample values

  **95.274 95. .265**

- char sample values

  **‘B’ ‘d’ ‘4’ ‘?’ ‘*’**

# What is a Variable?

- A variable is a location in memory which we can refer to by an identifier, and in which a data value that can be changed is stored.

- declaring a variable means specifying both its name and its data type

# What Does a Variable Declaration Do?

```
int   ageOfDog;
float taxRateY2K;
char  middleInitial;
```

- A declaration tells the compiler to allocate enough memory to hold a value of this data type, and to associate the identifier with this location.

**4 bytes for taxRateY2K**

**1 byte for middleInitial**

LECTURE 4

# C++ String Data Type

# C++ Data Type String

- a string is a sequence of characters enclosed in double quotes

- string sample values

  “Hello” “Year 2000” “1234”

- the empty string (null string) contains no characters and is written as “”

# More About Type String

- string is not a built-in (standard) type
  - it is a programmer-defined data type
  - it is provided in the C++ standard library

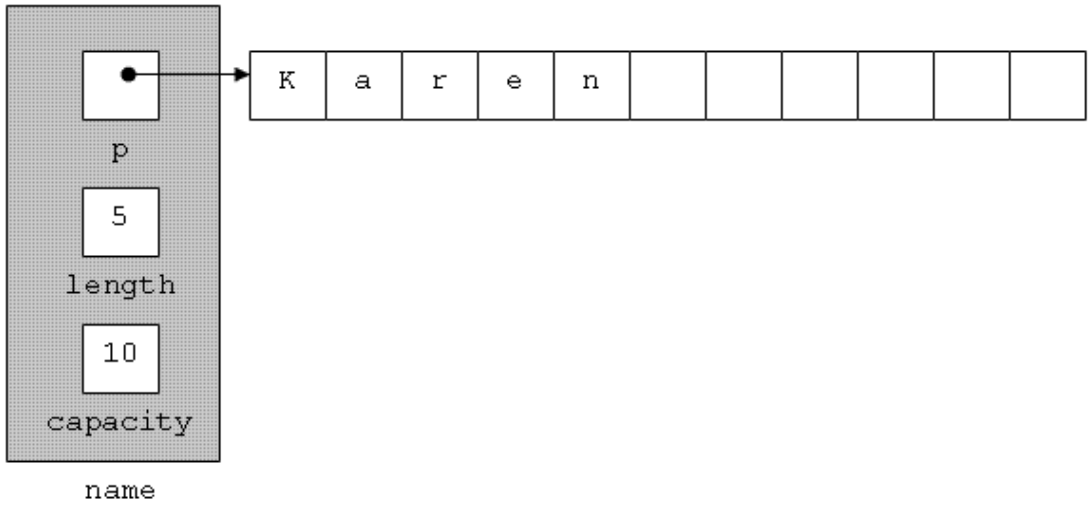
- string operations include
  - comparing 2 string values
  - searching a string for a particular character
  - joining one string to another

# C++ Representation in Memory

## #include <string>  // works a string object

Here is another example of declaring a C++ string:

string name = "Karen";




### C++ string

- **name** is a string object with several data members.
- The data member p is a pointer to (contains the address of) the first character in a dynamically-allocated array of characters.
- The data member length contains the length of the **string**.
- The data member capacity contains the number of valid characters that may currently be stored in the array.

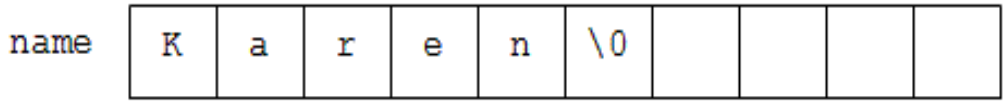
# C Representation in Memory

## Works as array of characters.

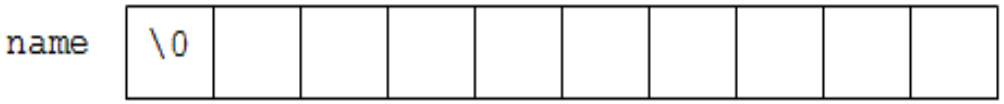
Here is another example of declaring a C string:

```
char name[10] = "Karen";
```

**Array of Characters (String) in C language**




**Null String in C language**

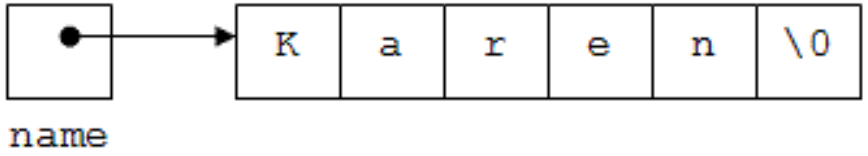



What about a C string declared as a char pointer?

```
char* name = "Karen";
```

This declaration creates an unnamed character array just large enough to hold the string "Karen" (including room for the null character) and places the address of the first element of the array in the char pointer name:

**Pointer-based C string**



LECTURE 4

# C++ Constants

# What is a Named Constant?

A **named constant** is a location in memory that we can refer to by an identifier, and in which a **data value that cannot be changed** is stored.

**VALID CONSTANT DECLARATIONS**

```
const   string   STARS = "****";
const   float    NORMAL_TEMP = 98.6;
const   char     BLANK = ' ';
const   int      VOTING_AGE = 18;
const   float    MAX_HOURS = 40.0;
```

# Standard Constants

Implementation Limits

**#include <climits>**

**INT_MIN  INT_MAX**

**LONG_MIN  LONG_MAX**

**#include <cfloat>**

**FLT_MIN  FLT_MAX**

**DBL_MIN  DBL_MAX**

# Integer literals

- An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or **0X** for hexadecimal, **0** for octal, and **nothing** for decimal.
- An integer literal can also have a suffix that is a combination of **U** and **L**, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         // Legal
215u        // Legal
0xFeeL      // Legal
078         // Illegal: 8 is not an octal digit
032UU       // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals:

```
85          // decimal
0213        // octal
0x4b        // hexadecimal
30          // int
30u         // unsigned int
30l         // long
30ul        // unsigned long
```

# Floating-point literals

- A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.
- While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159       // Legal
314159E-5L    // Legal
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

# Boolean literals

- There are two Boolean literals and they are part of standard C++ keywords:
  - A value of true representing true.
  - A value of false representing false.
- You should not consider the value of true equal to 1 and value of false equal to 0.

# Character literals

- Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in wchar_t type of variable . Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of char type.
- A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

| Escape sequence | Meaning |
| --- | --- |
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \ooo | Octal number of one to three digits |
| \xhh . . . | Hexadecimal number of one or more digits |

# Escape Characters

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

# String literals

- String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
- You can break a long line into multiple lines using string literals and separate them using whitespaces.
- Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear“
```

```
"hello, " "d" "ear"
```

```
"hello, \
dear"
```


# Defining Constants

There are two simple ways in C++ to define constants:

- Using **#define** preprocessor.
- Using **const** keyword.

# Demo Program

## constant1.cpp

Go Dev C++!!!

```
1   #include <iostream>
2   using namespace std;
3
4   #define LENGTH 10
5   #define WIDTH  5
6   #define NEWLINE '\n'
7
8   int main() {
9
10     int area;
11
12     area = LENGTH * WIDTH;
13     cout << area;
14     cout << NEWLINE;
15     return 0;
16  }
```

# Demo Program:

## constant2.cpp

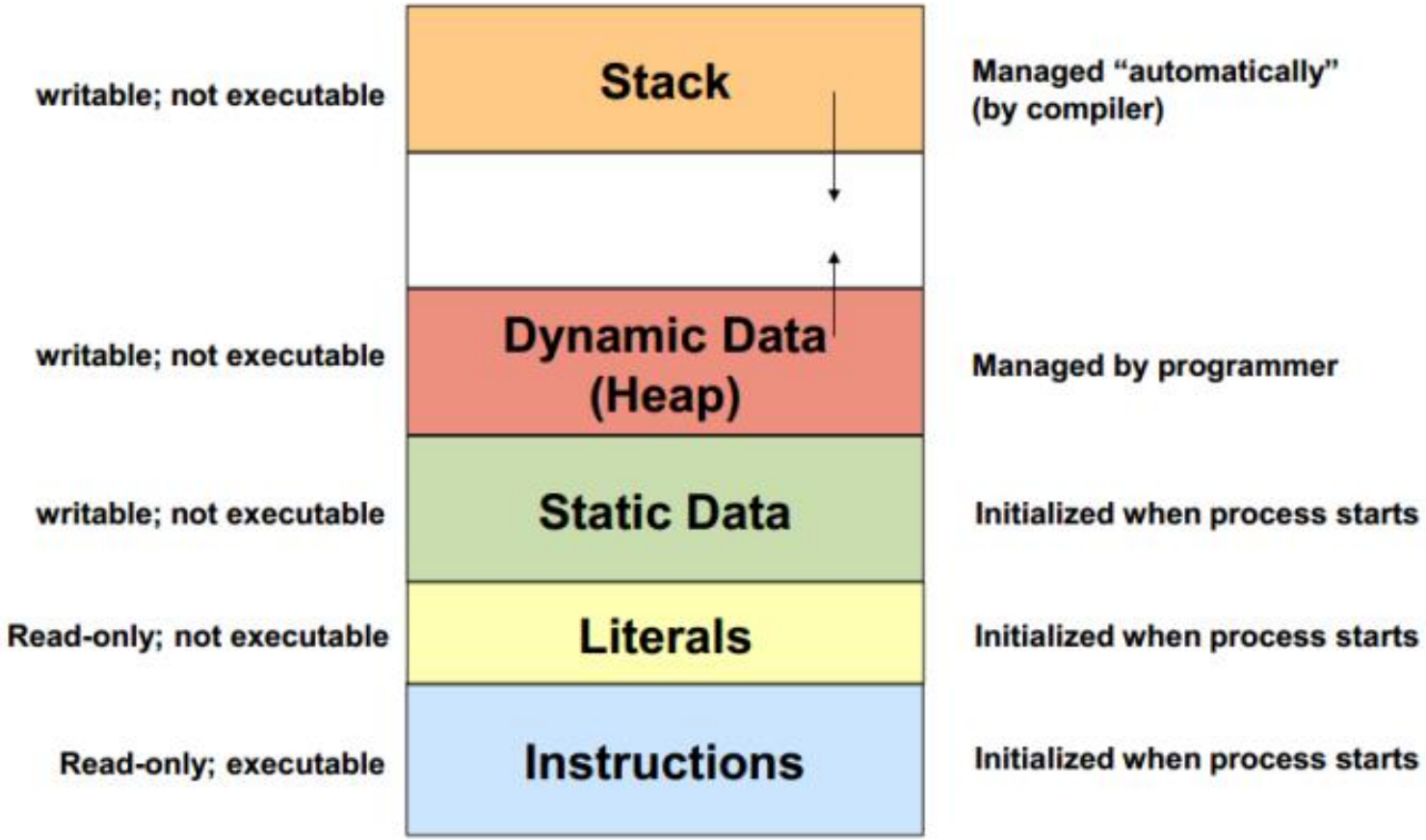
Go Dev C++!!!

```cpp
#include <iostream>
using namespace std;

int main() {
    const int  LENGTH = 10;
    const int  WIDTH  = 5;
    const char NEWLINE = '\n';
    int area;

    area = LENGTH * WIDTH;
    cout << area;
    cout << NEWLINE;
    return 0;
}
```

LECTURE 5

# C++ Memory Model

# C++ Memory Model




**Static and global variables:** Static Data
**Local variables:** Stack
**Objects, structs, arrays, vectors, strings:** Heap


# C11/C++11 memory model

A memory model describes the interactions of threads through memory and their shared use of the data.

The memory model sets the rules that different threads needs to follow when sharing data in memory. Unshared data act as before.

Key here is interaction of threads. No memory model was needed with Single-Thread programs.

# C11/C++11 memory object

The standard defines an ‚object', here called memory object to avoid confusion with normal C++ objects.

A memory object is a region of memory with a specific type. (think memory of int, double, char etc)
Elements of an aggregate (arrays, structs etc) are separate memory objects. (adjacent bitfields can share memory objects)

The standard gurantee that read / writes to different memory objects by different threads are independent. (The compiler must make that true).

Two different threads can read/write
e.g. a[0] and a[1] without any issues.
MyStruct has 5 memory objects in it.

```
struct MyStruct
{
  char a[4];
  int b;
};
```

LECTURE 6

# C++ Brief Expressions and Variable Assignments

# Giving a Value to a Variable

**You can assign (give) a value to a variable by using the assignment operator =**

**VARIABLE DECLARATIONS**

```
string   firstName ;
char     middleInitial ;
char     letter ;
int      ageOfDog;
```

**VALID ASSIGNMENT STATEMENTS**

```
firstName  =  “Fido” ;
middleInitial  =  ‘X’ ;
letter  =  middleInitial ;
ageOfDog  =  12 ;
```

# What is an Expression in C++?

- An expression is a valid arrangement of variables, constants, and operators.

- in C++ each expression can be evaluated to compute a value of a given type

- the value of the expression

  9 + 5 is 14

# Assignment Operator Syntax

```
Variable = Expression
```

First, Expression on right is evaluated.

Then the resulting value is stored in the memory location of Variable on left.

NOTE: An automatic type conversion occurs after evaluation but before the value is stored if the types differ for Expression and Variable


# Assignment Operator Syntax

Examples

- Y = 3;
- X = X + 1;
- Total = (Total + 1) / Count;

# String Concatenation (+)

- concatenation is a binary operation that uses the + operator

- at least one of the operands must be a string variable or named constant--the other operand can be string type or char type

# Concatenation Example

```
const   string  WHEN = “Tomorrow” ;
const   char     EXCLAMATION = ‘!’ ;
string   message1 ;
string  message2 ;


 message1 = “Yesterday “ ;
 message2 = “and “ ;
 message1 = message1 + message2 +
                                        WHEN + EXCLAMATION ;
```

# Insertion Operator ( << )

- The command cout is predefined to denote an output stream that goes to the standard output device (display screen)

- the insertion operator << called "put to" takes 2 operands

- the left operand is a stream expression, such as cout. The right operand is an expression of simple type or a string constant

# Output Statements

**SYNTAX**

```
cout << Expression << Expression . . . ;
```

**These examples yield the same output:**

```
cout << "The answer is " ;
cout << 3 * 4 ;
```

```
cout << "The answer is " << 3 * 4 ;
```

LECTURE 7

# C++ Program Libraries

# Is compilation the first step?

**No. Before your source program is compiled, it is first examined by the preprocessor to**

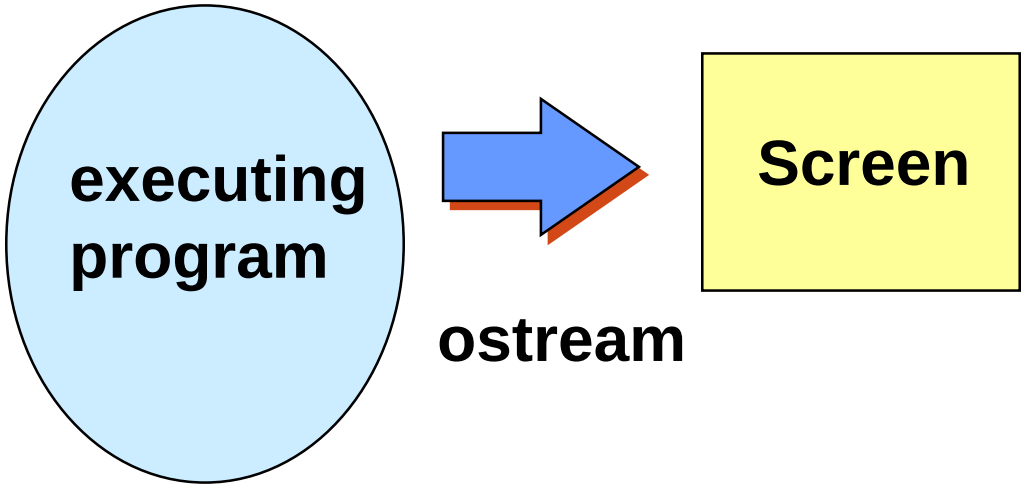
- **remove all comments from source code**
- **handle all preprocessor directives--they begin with the # character such as**

  **#include <iostream>**

**Note: tells preprocessor to look in the standard include directory for the header file called iostream and insert its contents into your source code**

# No I/O is built into C++

Instead, a library provides an output stream

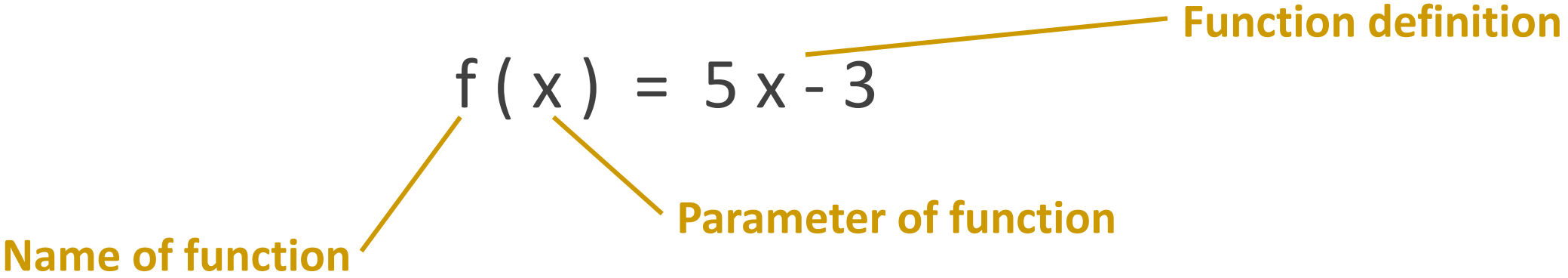


# Using Libraries

- A library has 2 parts
  - Interface (stored in a header file) tells what items are in the library and how to use them.
  - Implementation (stored in another file) contains the definitions of the items in the library.

- #include <iostream>
  - Refers to the header file for the *iostream* library needed for use of cout and endl.

# Function Concept in Math




When x = 1, f ( x ) = 2 is the returned value.

When x = 4, f ( x ) = 17 is the returned value.

Returned value is determined by the function definition and by the values of any parameters.

# Demo Program:

## PrintName.cpp

```
// ********************************************************
//  PrintName program
//  This program prints a name in two different formats
// ********************************************************

#include <iostream>     // for cout and endl
#include <string>       // for data type string

using namespace std;

const  string  FIRST = "Herman";  // Person's first name
const  string  LAST =  "Smith";   // Person's last name
const  char    MIDDLE = 'G';      // Person's middle initial
```

# Demo Program:

## PrintName.cpp

```
int  main( )
{
     string    firstLast;     //  Name in first-last format
     string    lastFirst;     //  Name in last-first format

     firstLast = FIRST + " " + LAST ;
     cout  << "Name in first-last format is "  << endl
       << firstLast  <<  endl;

     lastFirst = LAST + ", " + FIRST + ' ' ;
     cout  << "Name in first-last format is "  << endl
       << lastFirst  <<  MIDDLE  <<  '.'  <<  endl;

     return  0;
}
```

# Output of Program

```
Name in first-last format is Herman Smith
Name in last-first-initial format is Smith, Herman G.
```