Basic elements of C++

Following the book:

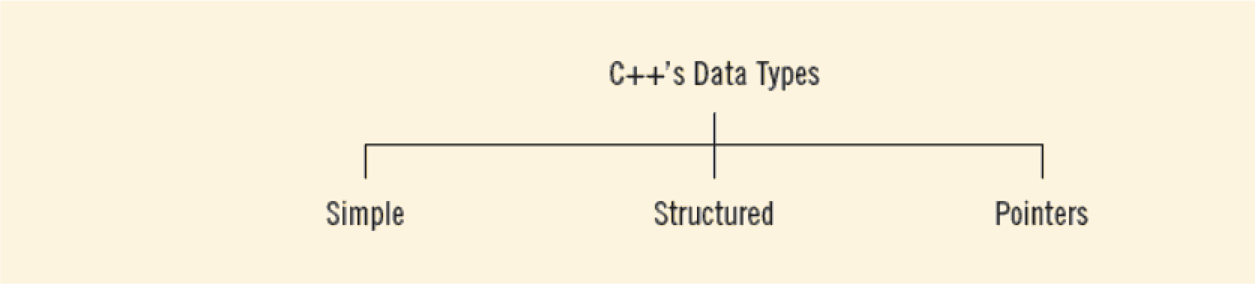
- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*
- Useful documentation: http://www.cplusplus.com/

Contents

- Data types
- Operators
- Flow control
- User defined functions
- Arrays
- Classes
- Pointers
- Standard library
- Reading/writing files

Data types

- Data type: set of values together with a set of operations



C++'s Data Types

Simple

Structured

Pointers

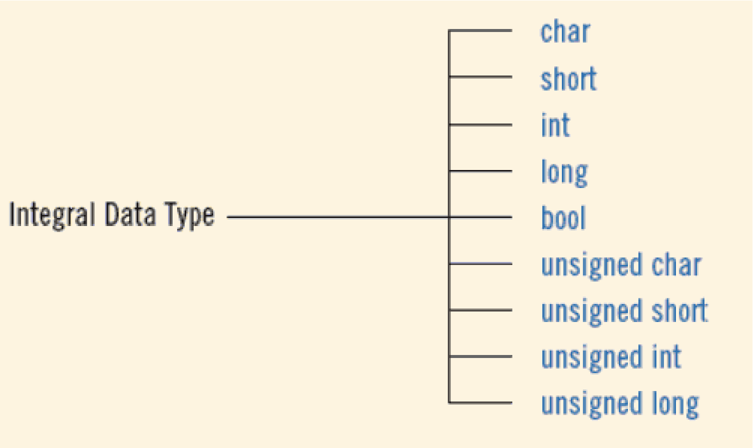
FIGURE 2-1 C++ data types

- Three categories of simple data

  Integers

  Floating-point: real numbers

  Enumeration type: user-defined data type



Integral Data Type

char

short

int

long

bool

unsigned char

unsigned short

unsigned int

unsigned long

Data types

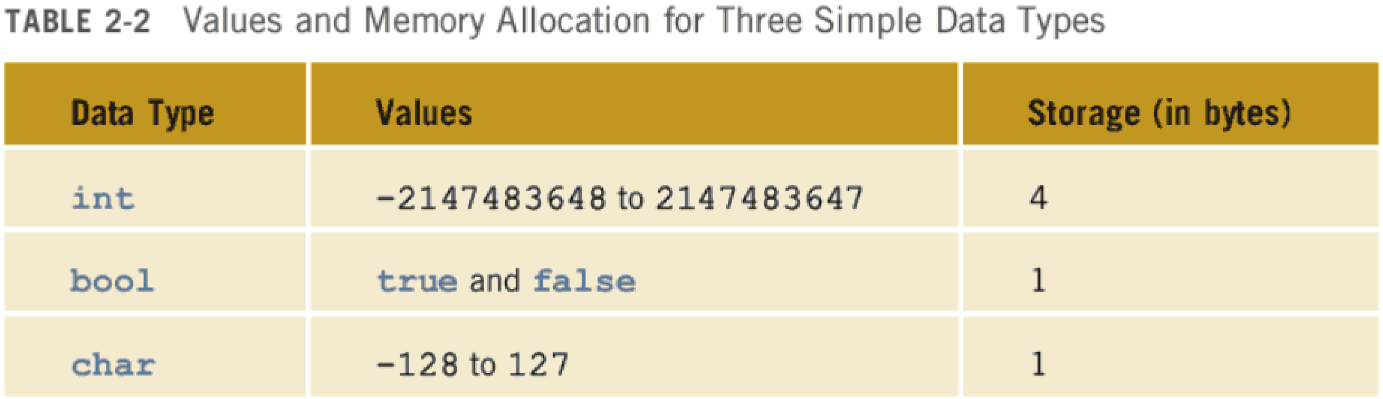


TABLE 2-2 Values and Memory Allocation for Three Simple Data Types

Data Type	Values	Storage (in bytes)
`int`	-2147483648 to 2147483647	4
`bool`	`true` and `false`	1
`char`	-128 to 127	1

- bool type: used to manipulate logical (Boolean) expressions
  - Two possible values: true, false
  - True, false: reserved words
- char: used for characters (smallest type)

```
'A', 'a', '0', '*', '+', '$', '&'
```

Data types

- Represent real numbers

  Range: from -3.4E+38 to +3.4E+38 (four bytes)

  Maximum number of significant digits: 6 or 7

- double: floating point of double precision

  Range: -1.7E+308 to 1.7E+308 (eight bytes)

  Maximum number of significant digits: 15

Operators

- Operators

  Binary or unary

  Act on an expression to give another expression

+ addition

- subtraction

* multiplication

/ division

% modulus operator

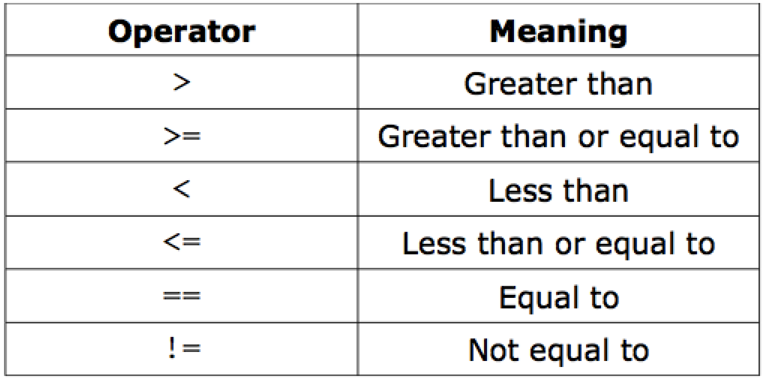
- All operations inside of () are evaluated first
- *, /, and % are at the same level of precedence and are evaluated next
- + and - have the same level of precedence and are evaluated last
- When operators are on the same level – Performed from left to right (associativity)

`3 * 7 - 6 + 2 * 5 / 4 + 6` means

`(((3 * 7) - 6) + ((2 * 5) / 4 )) + 6`

Relational, logical, increment operators

Relational operators



Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Increment/decrement operators

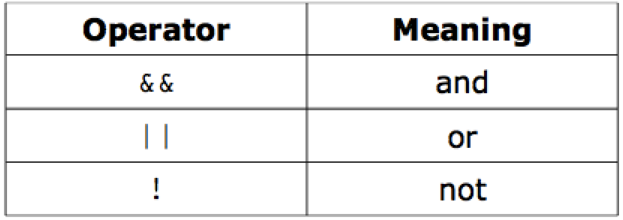
++variable, variable++

--variable, variable--

```
x = 5;
y = ++x;
```

```
x = 5;
y = x++;
```

Logical operators



Operator	Meaning
&&	and
||	or
!	not

Examples

Assume x=6, y=2:

```
!(x > 2) → false
(x > y) && (y > 0) → true
(x < y) && (y > 0) → false
(x < y) || (y > 0) → true
```

Ternary operator ?:

- Example: `result = a > b ? x : y;`

- Equivalent to:

```
if(a > b)
      result = x;
else
      result = y;
```

Expressions

- Statement: unit of code that does something – a basic building block of a program.
- Expression: a statement that has a value
  - If all operands are integer: integer expressions
  - If all operands are float: floating point expression
  - If mixed:
    - Integer is changed to floating-point
    - Operator is evaluated
    - Result is floating-point

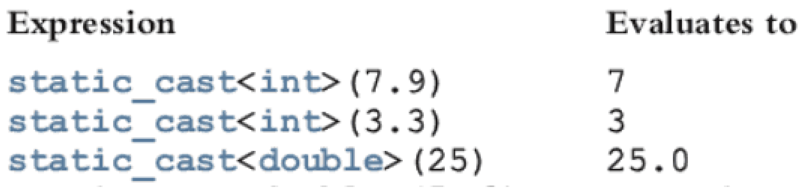
Example of implicit type conversion

Type conversions & cast

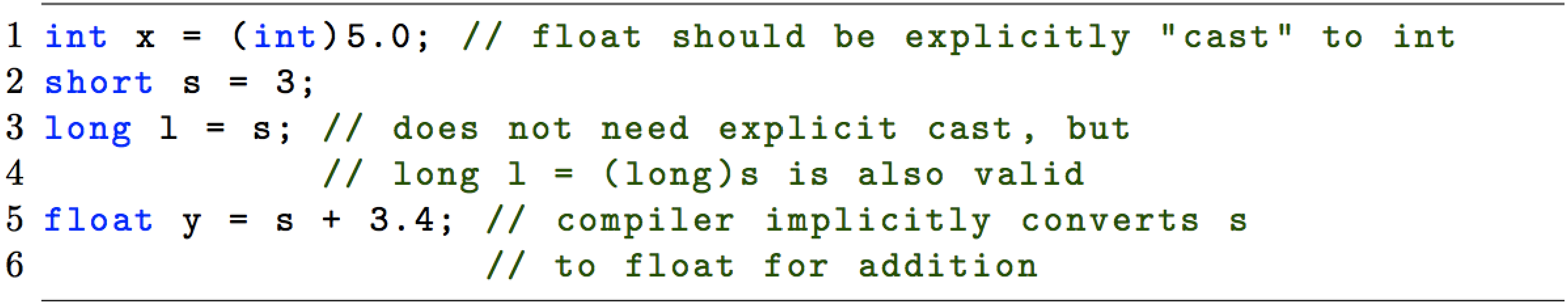
- Implicit type conversion:

  When changing from smaller to larger types

- Explicit type conversion: `static_cast<dataTypeName>(expression)`



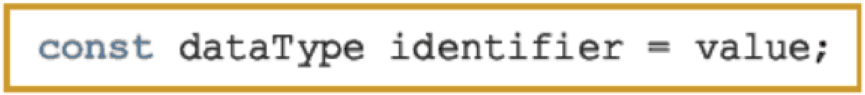
Expression	Evaluates to
`static_cast<int>(7.9)`	7
`static_cast<int>(3.3)`	3
`static_cast<double>(25)`	25.0



```
int x = (int)5.0; // float should be explicitly "cast" to int
short s = 3;
long l = s; // does not need explicit cast, but
            // long l = (long)s is also valid
float y = s + 3.4; // compiler implicitly converts s
                   // to float for addition
```

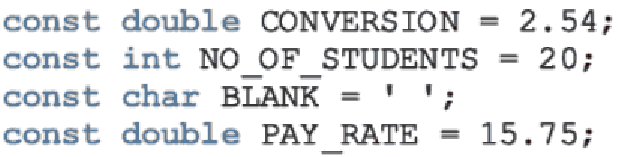
Variables and constants

- Named constant: memory location whose content can't change during execution



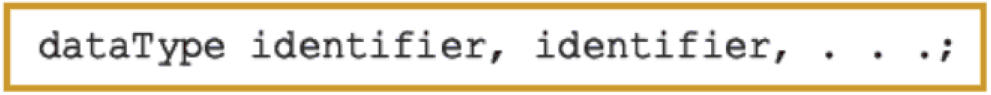
```
const dataType identifier = value;
```

Examples

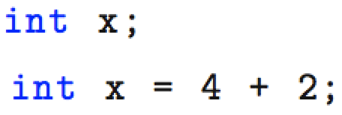


```
const double CONVERSION = 2.54;
const int NO_OF_STUDENTS = 20;
const char BLANK = ' ';
const double PAY_RATE = 15.75;
```

- Variable: memory location whose content may change during execution



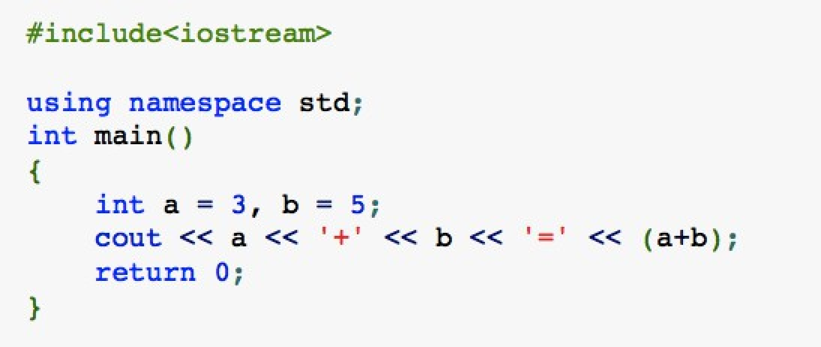
```
dataType identifier, identifier, . . .;
```



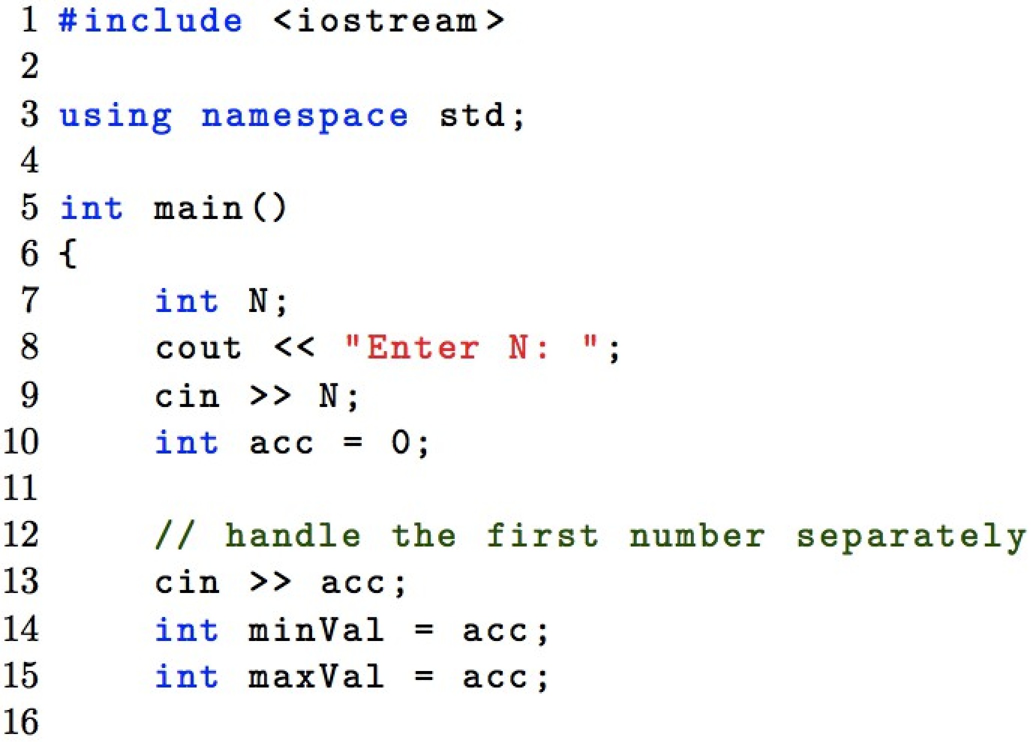
```
int x;

int x = 4 + 2;
```

} All variables must be initialized before using them, but not necessarily during declaration

Examples

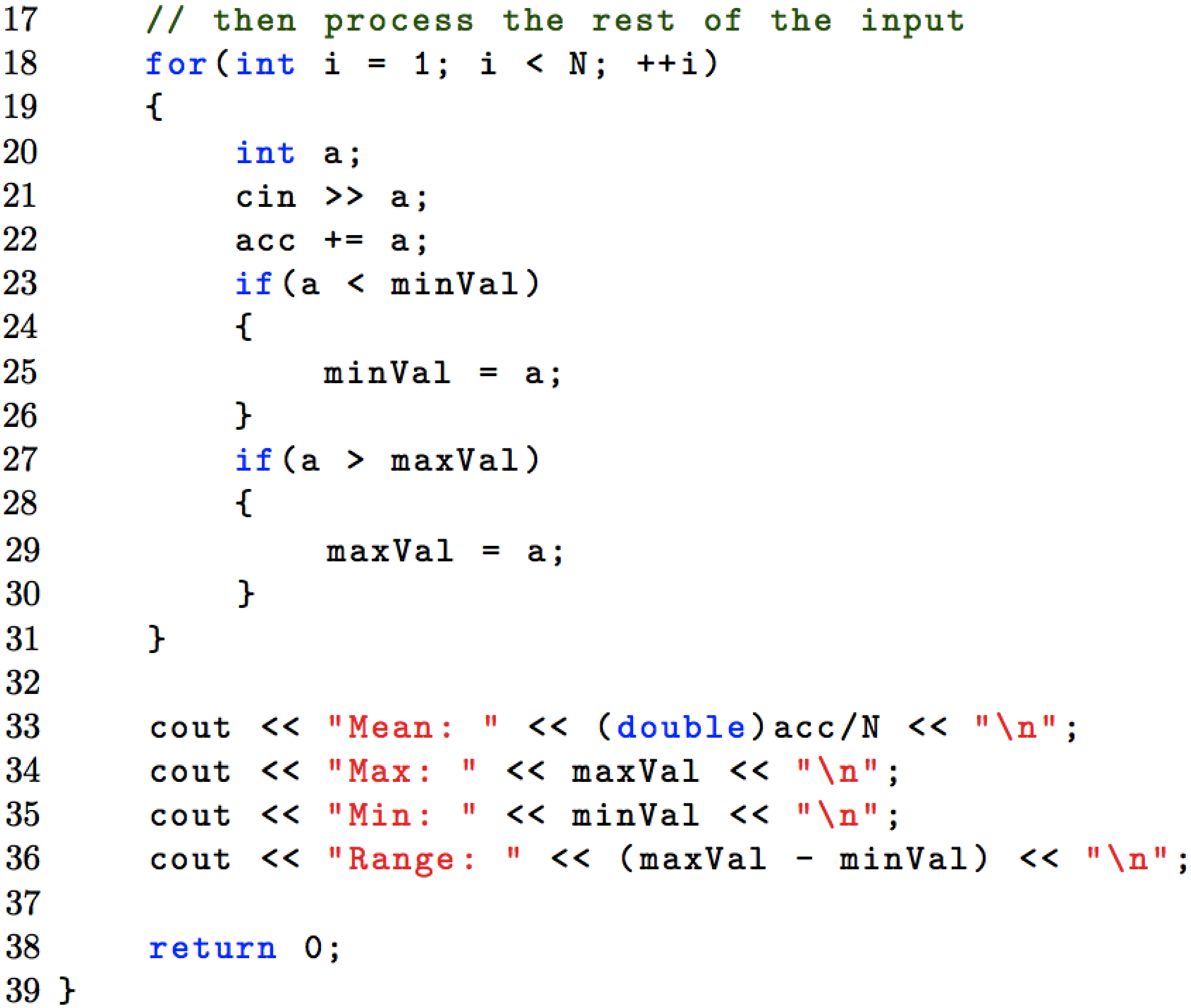


```
#include<iostream>

using namespace std;
int main()
{
    int a = 3, b = 5;
    cout << a << '+' << b << '=' << (a+b);
    return 0;
}
```



```
#include <iostream>

using namespace std;

int main()
{
    int N;
    cout << "Enter N: ";
    cin >> N;
    int acc = 0;

    // handle the first number separately
    cin >> acc;
    int minVal = acc;
    int maxVal = acc;

```

Examples II



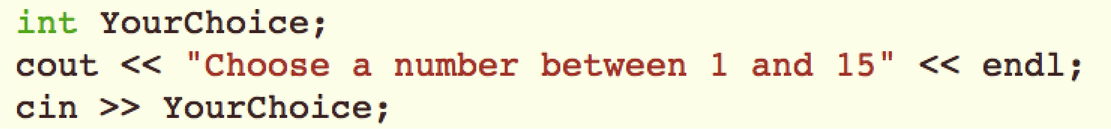
```cpp
    // then process the rest of the input
    for(int i = 1; i < N; ++i)
    {
        int a;
        cin >> a;
        acc += a;
        if(a < minVal)
        {
            minVal = a;
        }
        if(a > maxVal)
        {
            maxVal = a;
        }
    }

    cout << "Mean: " << (double)acc/N << "\n";
    cout << "Max: " << maxVal << "\n";
    cout << "Min: " << minVal << "\n";
    cout << "Range: " << (maxVal - minVal) << "\n";

    return 0;
}
```

Input/Output statements

- Output: cout

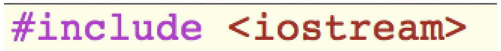
  Ex.: cout << " The factorial of 5 is " << Factorial(5) << endl;
- The stream insertion operator is <<
- The expression is evaluated and its value is printed at the current cursor position on the screen
- Input:

  cin >> x;



```
int YourChoice;
cout << "Choose a number between 1 and 15" << endl;
cin >> YourChoice;
```

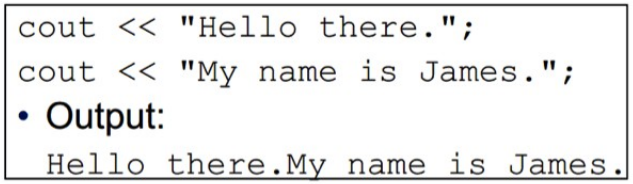
- Include file:



```
#include <iostream>
```

Input/Output statements

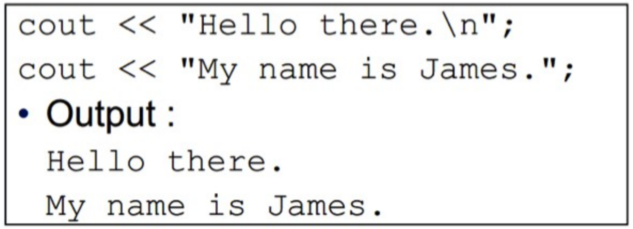
- Modifiers to change the format of the output



```
cout << "Hello there.";
cout << "My name is James.";
```

- Output:

```
Hello there.My name is James.
```



```
cout << "Hello there.\n";
cout << "My name is James.";
```

- Output :

```
Hello there.
My name is James.
```

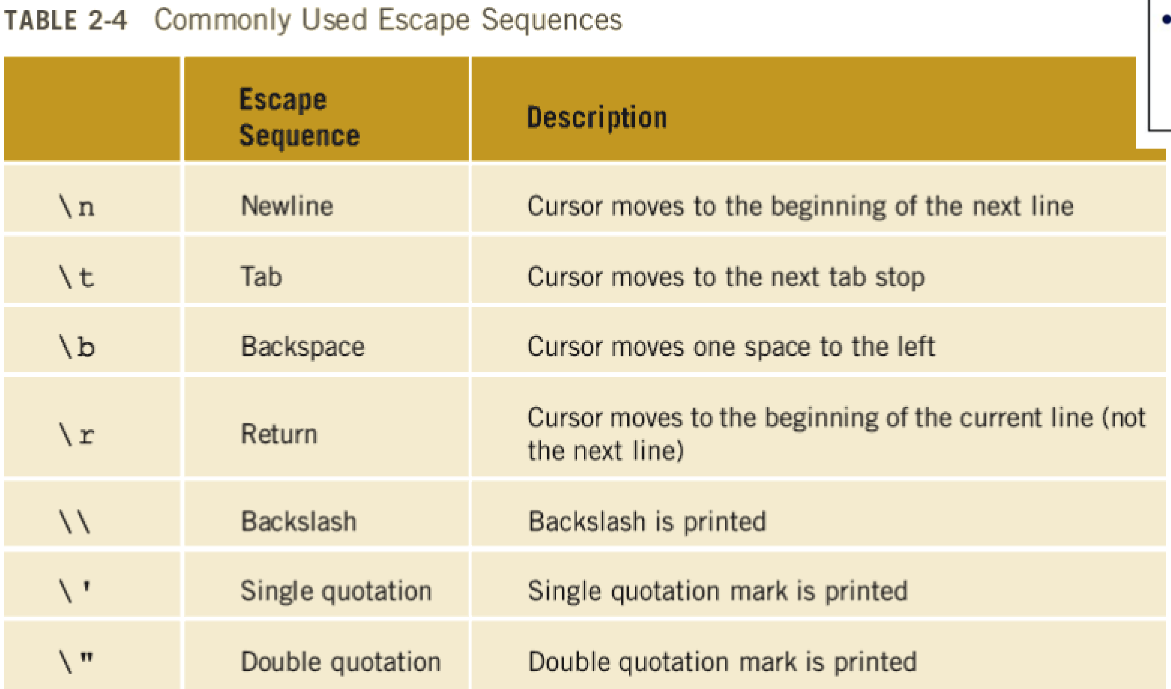
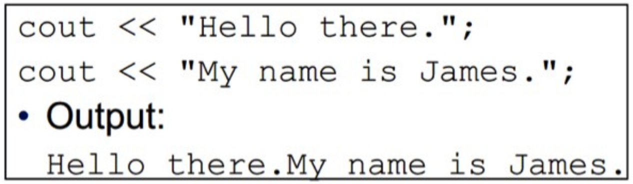


TABLE 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
`\n`	Newline	Cursor moves to the beginning of the next line
`\t`	Tab	Cursor moves to the next tab stop
`\b`	Backspace	Cursor moves one space to the left
`\r`	Return	Cursor moves to the beginning of the current line (not the next line)
`\\`	Backslash	Backslash is printed
`\'`	Single quotation	Single quotation mark is printed
`\"`	Double quotation	Double quotation mark is printed

Input/Output statements

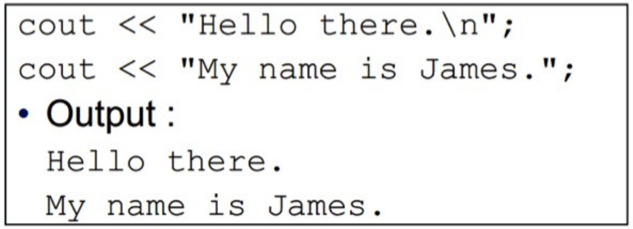
- Modifiers to change the format of the output



```
cout << "Hello there.";
cout << "My name is James.";
```

- Output:

```
Hello there.My name is James.
```



```
cout << "Hello there.\n";
cout << "My name is James.";
```

- Output :

```
Hello there.
My name is James.
```

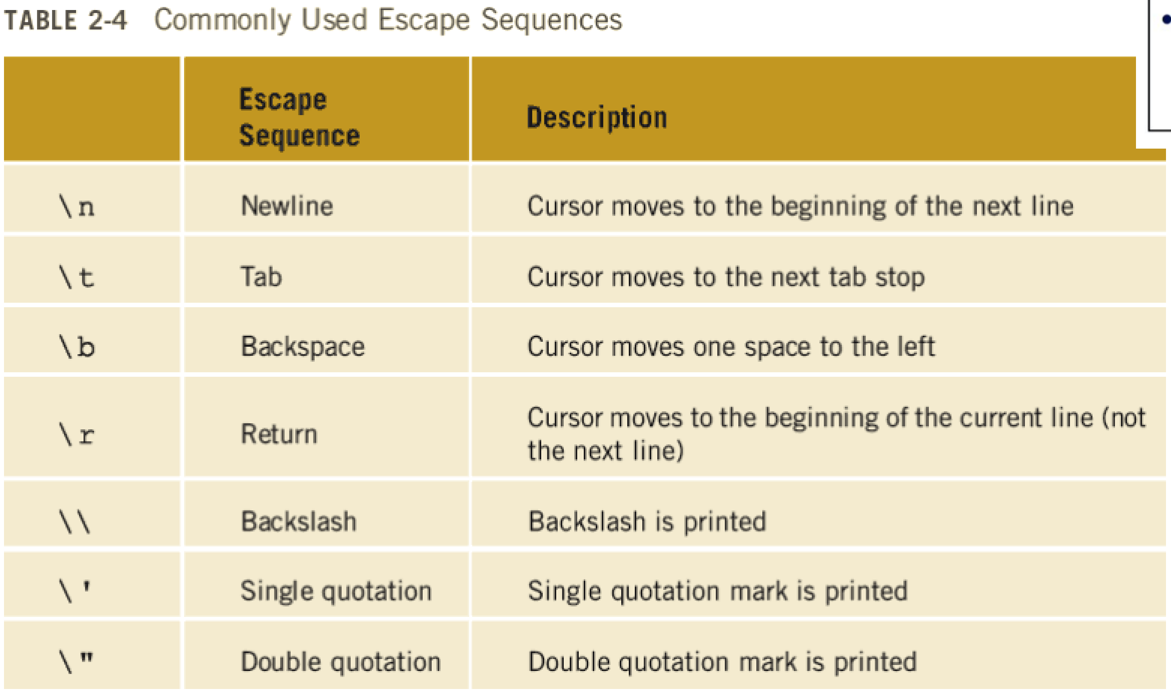


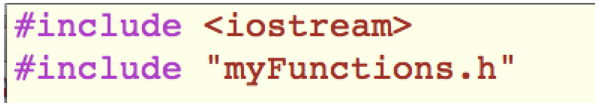
TABLE 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
`\n`	Newline	Cursor moves to the beginning of the next line
`\t`	Tab	Cursor moves to the next tab stop
`\b`	Backspace	Cursor moves one space to the left
`\r`	Return	Cursor moves to the beginning of the current line (not the next line)
`\\`	Backslash	Backslash is printed
`\'`	Single quotation	Single quotation mark is printed
`\"`	Double quotation	Double quotation mark is printed

Pre-processor directives

- C++ has a small number of operations
- Many functions and symbols needed to run a
- C++ program are provided as collection of libraries

  Every library has a name and is referred to by a header file
- Preprocessor directives are commands supplied to the preprocessor
- All preprocessor commands begin with #
- No semicolon at the end of these commands!
- Syntax to include header files:



```
#include <iostream>
#include "myFunctions.h"
```

Namespace

- Normal syntax

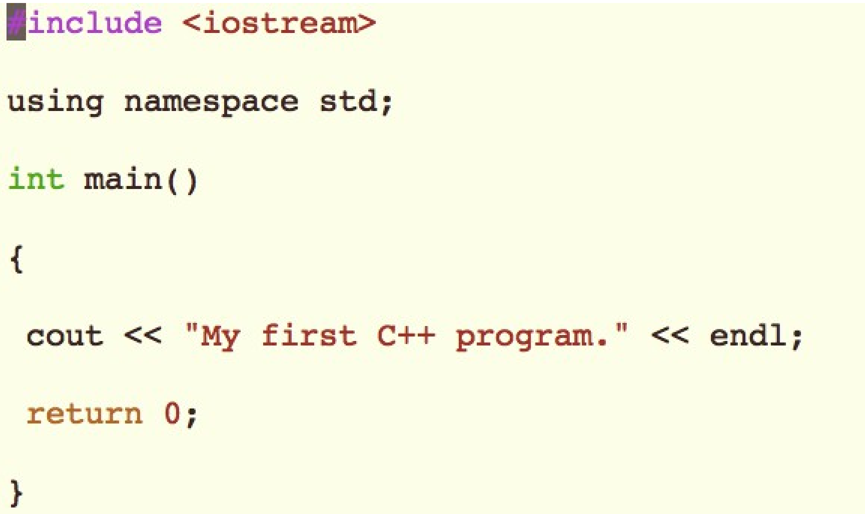
  std::cout << " The factorial of 5 is " << Factorial(5) << std::endl;

- std:: indicates that these commands belong to the standard library

  Will become more clear in next classes

- To avoid writing all the time std::

  using namespace std;



```
#include <iostream>

using namespace std;

int main()

{

 cout << "My first C++ program." << endl;

 return 0;

}
```

Exercise

- Write a program that takes as input a given length expressed in feet and inches

  Convert and output the length in centimetres

- Help:

  Inch = 2.54 cm

  1 foot = 12 inches

Flow Control

Control structures

- A computer can proceed:
  - In sequence
  - Selectively (branch) – making a choice
  - Repetitively (iteratively) – looping
- Some statements are executed only if certain conditions are met
  - A condition is met if it evaluates to true

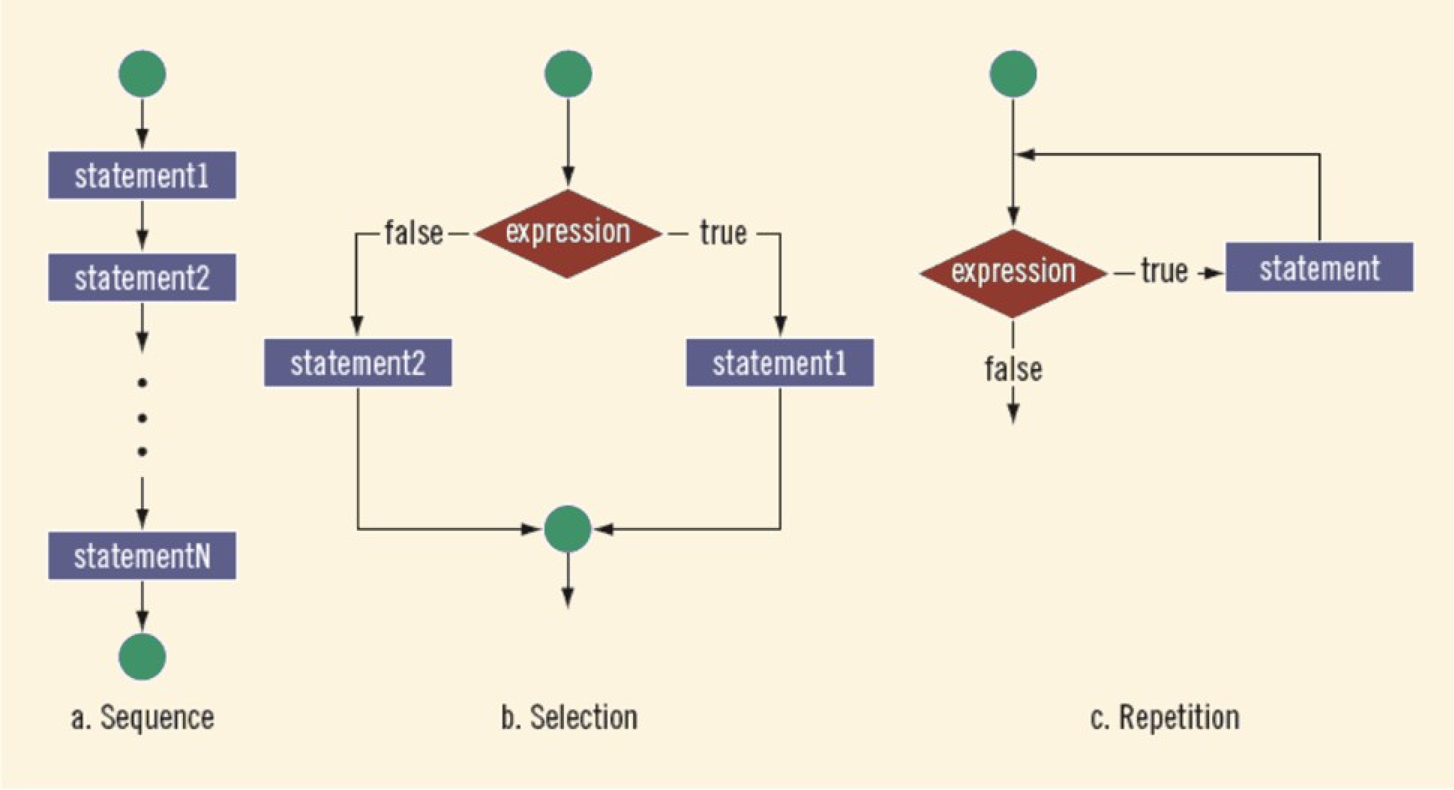
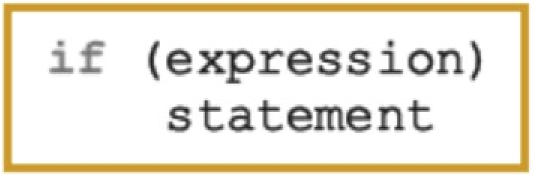


FIGURE 4-1 Flow of execution

if and if ... else

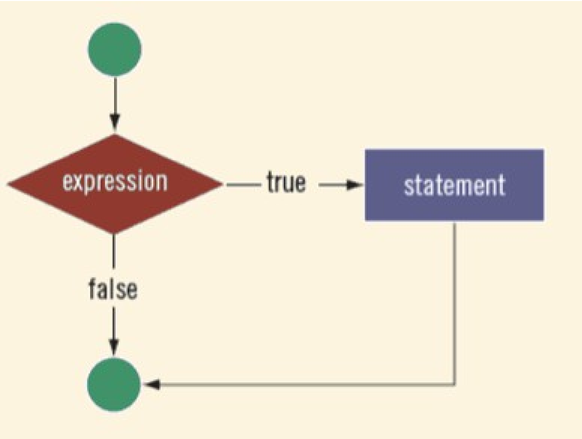
- One-Way Selection:

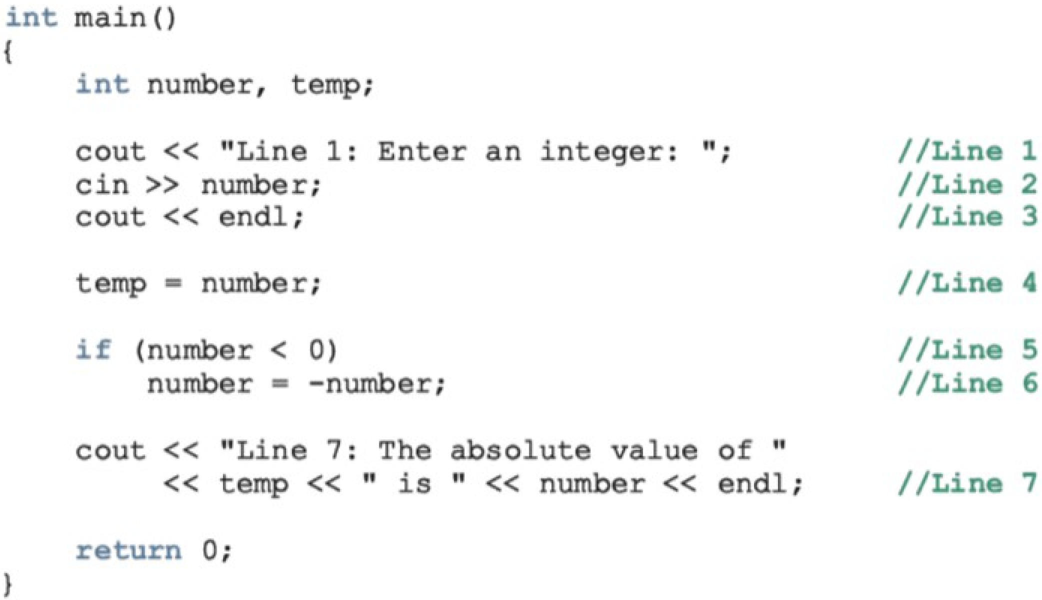


```
if (expression)
    statement
```

The statement is executed if the value of expression is true

If expression is false, the statement is not executed and the program continues

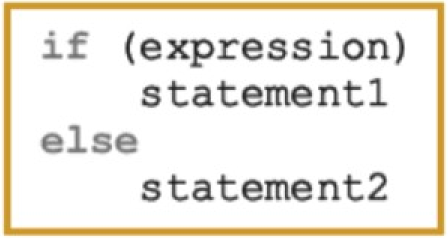




```
int main()
{
    int number, temp;

    cout << "Line 1: Enter an integer: ";           //Line 1
    cin >> number;                                  //Line 2
    cout << endl;                                   //Line 3

    temp = number;                                  //Line 4

    if (number < 0)                                 //Line 5
        number = -number;                           //Line 6

    cout << "Line 7: The absolute value of "
         << temp << " is " << number << endl;       //Line 7

    return 0;
}
```

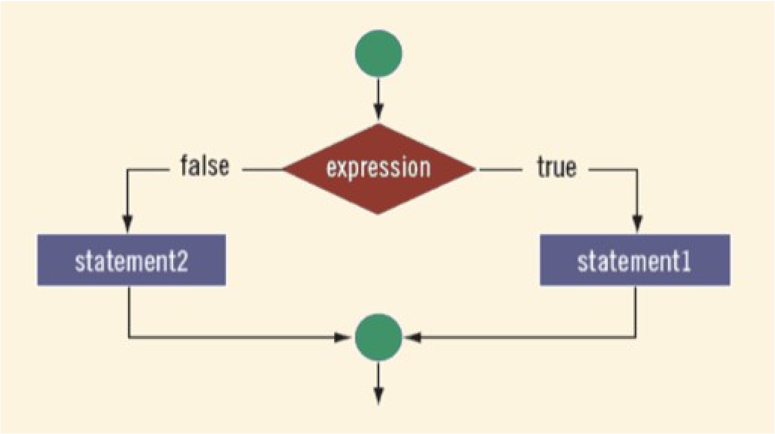
if and if ... else

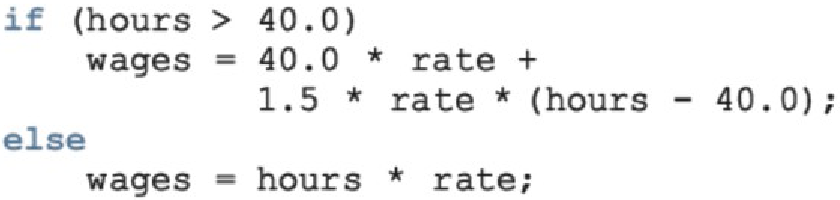
- Two-Way Selection:



```
if (expression)
    statement1
else
    statement2
```

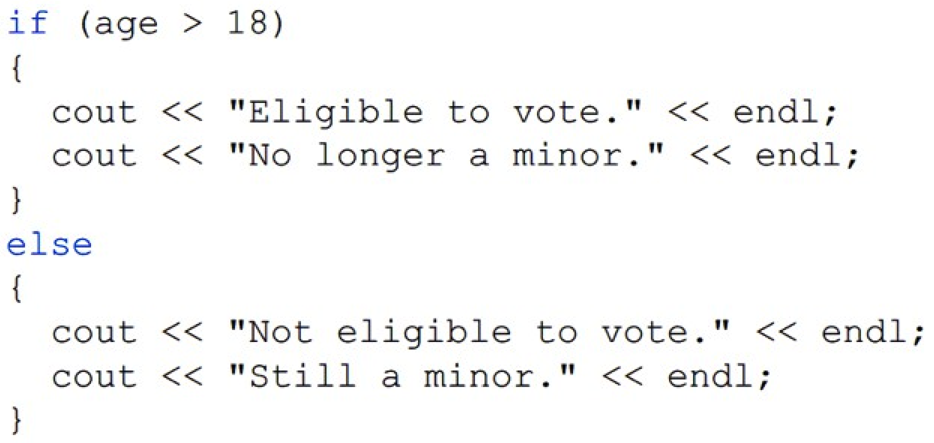
If expression is true, statement1 is executed; otherwise, statement2 is executed





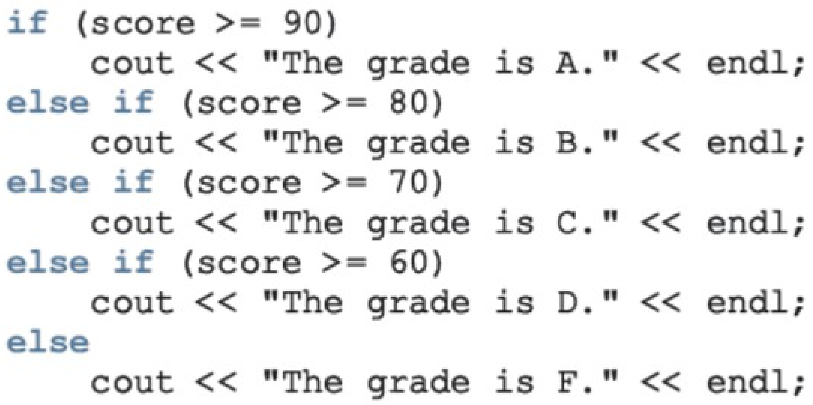
```
if (hours > 40.0)
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);
else
    wages = hours * rate;
```

- Block of statements:



```
if (age > 18)
{
  cout << "Eligible to vote." << endl;
  cout << "No longer a minor." << endl;
}
else
{
  cout << "Not eligible to vote." << endl;
  cout << "Still a minor." << endl;
}
```

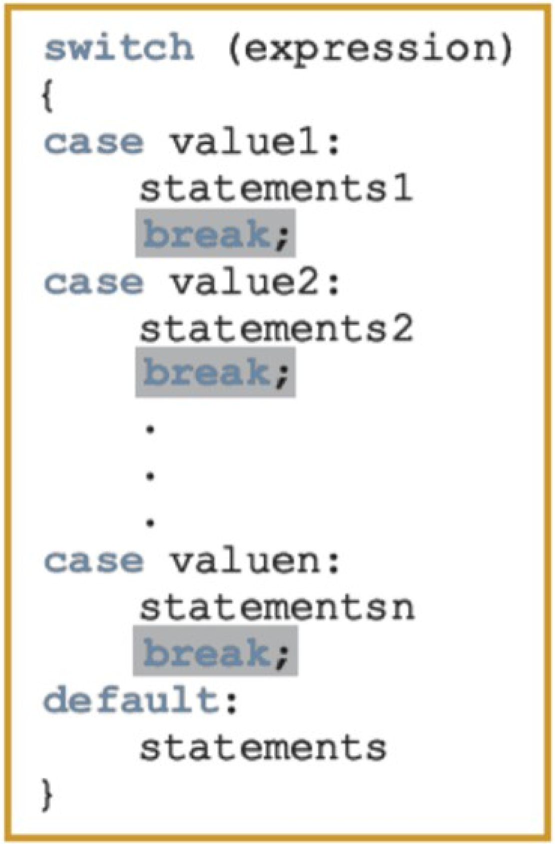
- Multiple options



```
if (score >= 90)
    cout << "The grade is A." << endl;
else if (score >= 80)
    cout << "The grade is B." << endl;
else if (score >= 70)
    cout << "The grade is C." << endl;
else if (score >= 60)
    cout << "The grade is D." << endl;
else
    cout << "The grade is F." << endl;
```

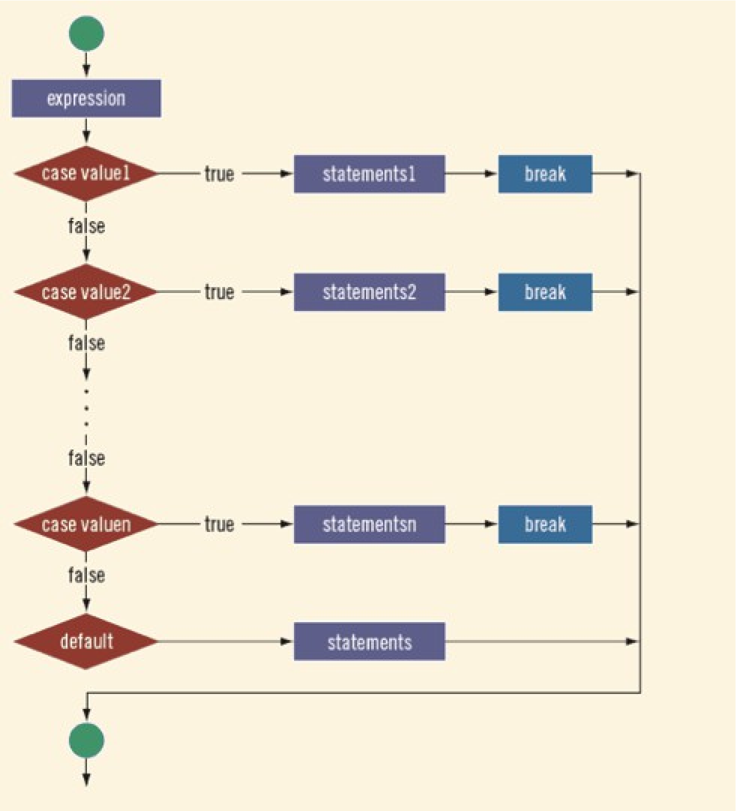
switch

- Alternative to a series of if... else
- The expression is evaluated. Depending on the value different statements will be executed
- More than one statement may follow
- Break may/may not appear
  - If it does not appear the following statements will be executed!

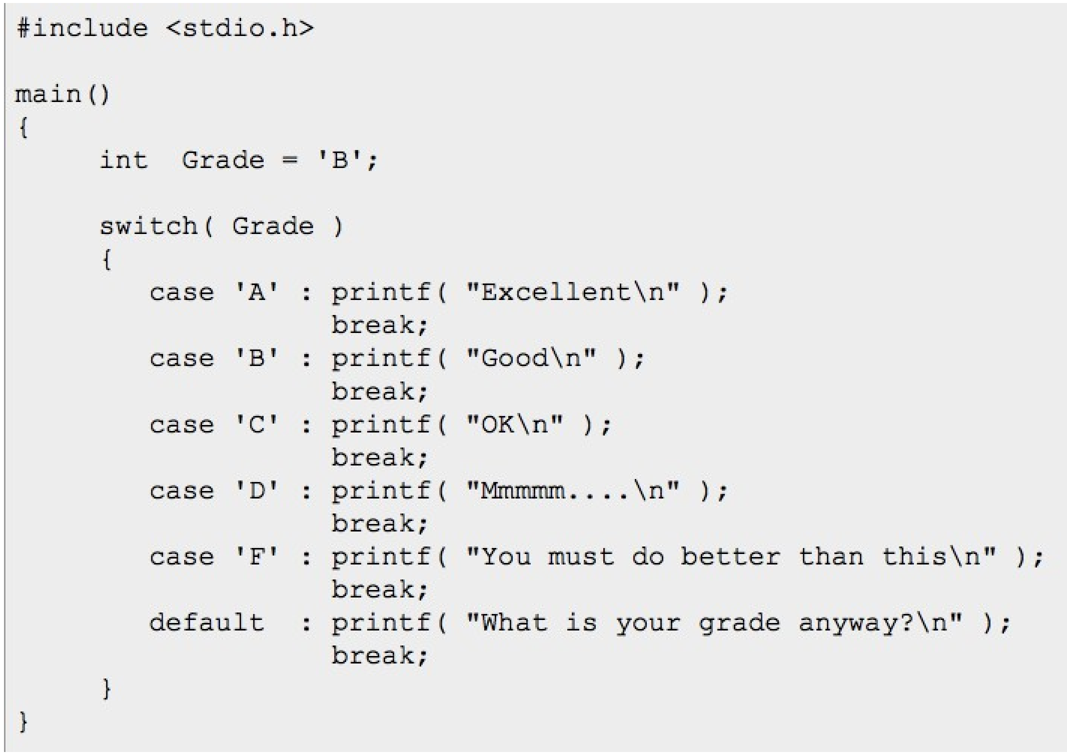


```
switch (expression)
{
case value1:
      statements1
      break;
case value2:
      statements2
      break;
      .
      .
      .
case valuen:
      statementsn
      break;
default:
      statements
}
```

✯ Flow diagram



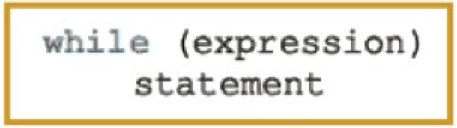
✯ Example:



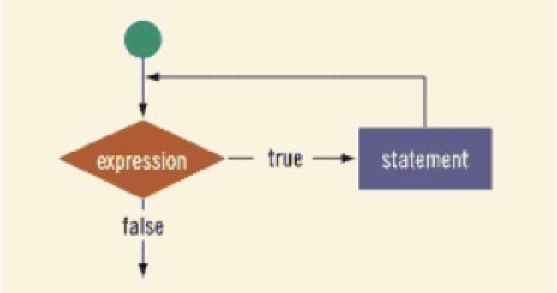
```
#include <stdio.h>

main()
{
      int  Grade = 'B';

      switch( Grade )
      {
         case 'A' : printf( "Excellent\n" );
                    break;
         case 'B' : printf( "Good\n" );
                    break;
         case 'C' : printf( "OK\n" );
                    break;
         case 'D' : printf( "Mmmmm....\n" );
                    break;
         case 'F' : printf( "You must do better than this\n" );
                    break;
         default  : printf( "What is your grade anyway?\n" );
                    break;
      }
}
```

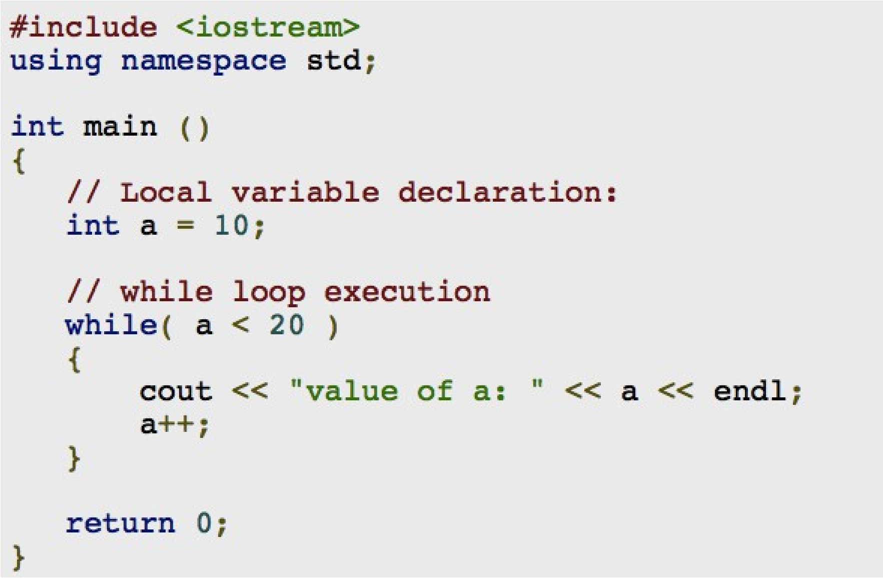
while loop

- While the expression is true, execute the statement
- Can become an infinite loop
  - Ensure that expression becomes false at certain point

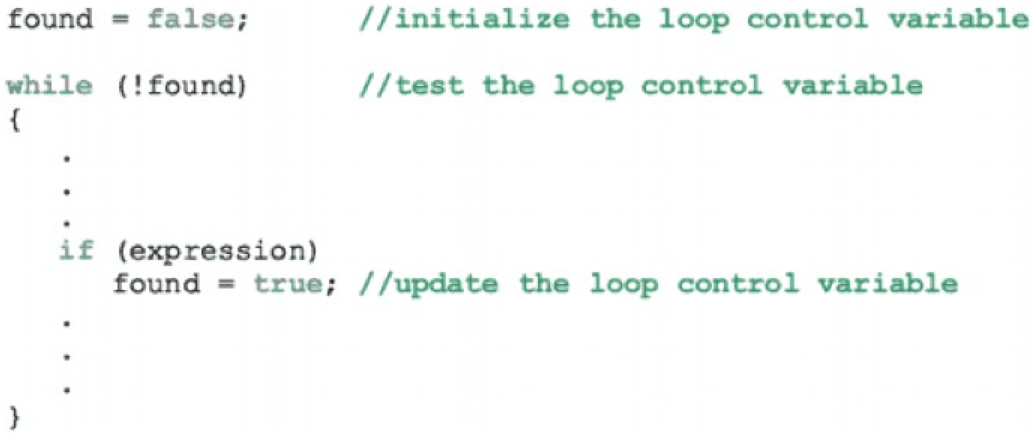


```
while (expression)
    statement
```





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

int main ()
{
   // Local variable declaration:
   int a = 10;

   // while loop execution
   while( a < 20 )
   {
       cout << "value of a: " << a << endl;
       a++;
   }

   return 0;
}
```



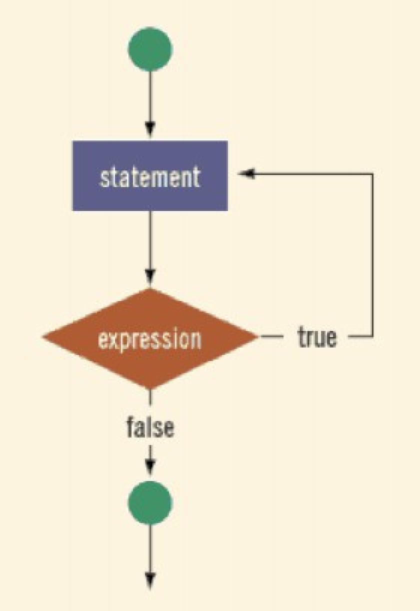
```
found = false;      //initialize the loop control variable

while (!found)      //test the loop control variable
{
    .
    .
    .
    if (expression)
        found = true; //update the loop control variable
    .
    .
    .
}
```

do ... while

- Execute the statement until expression is true

  Ensure that expression becomes true to avoid infinite loop

```
do
    statement
while (expression);
```

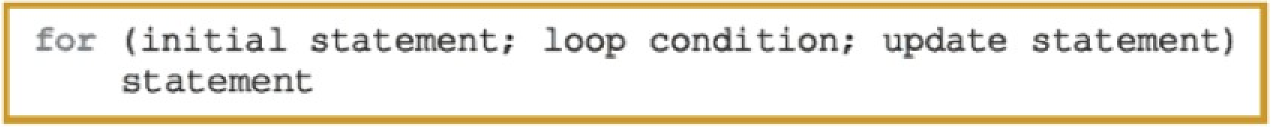


a.
```
i = 11;
while (i <= 10)
{
    cout << i << " ";
    i = i + 5;
}
cout << endl;
```

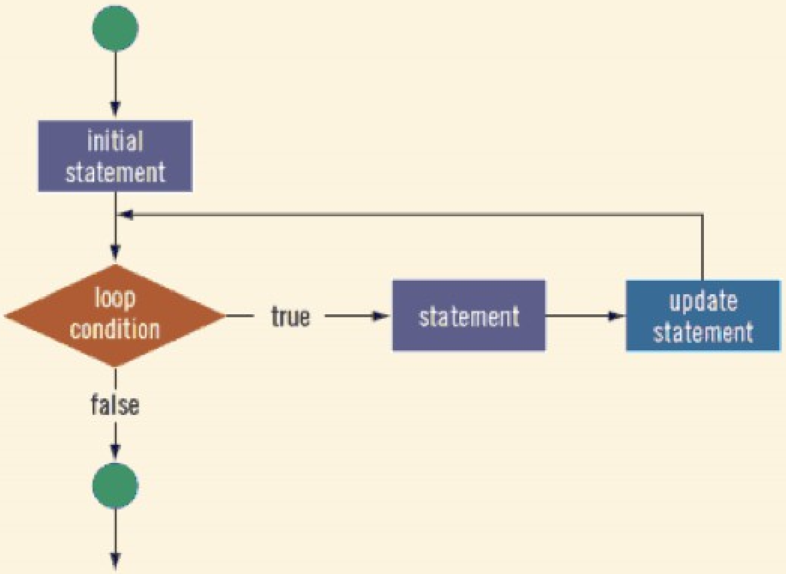
b.
```
i = 11;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 10);

cout << endl;
```

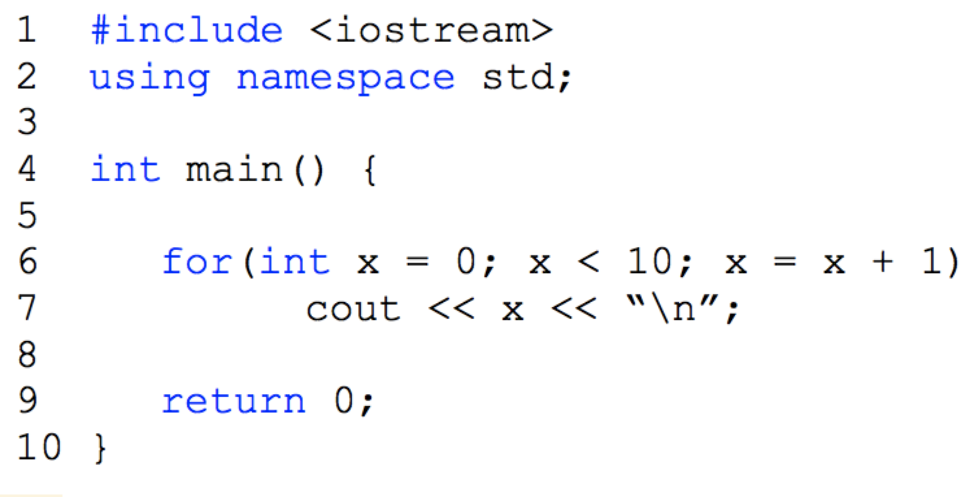
The for loop

- designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop.



```
for (initial statement; loop condition; update statement)
    statement
```





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

int main() {

    for(int x = 0; x < 10; x = x + 1)
          cout << x << "\n”;

    return 0;
}
```

for versus while

- For loop equivalent to while loop:

```
for(initialization; condition; incrementation)
{
        statement1
        statement2
        …
}
```

```
initialization
while(condition)
{
        statement1
        statement2
        …
        incrementation
}
```

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

int main() {

    for(int x = 0; x < 10; x = x + 1)
            cout << x << “\n”;

    return 0;
}
```

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

int main() {

    int x = 0;
    while(x < 10) {
            cout << x << “\n”;
            x = x + 1;
    }

    return 0;
}
```

break and continue

- They alter the flow of control
- break statement is used for two purposes:

  To exit early from a loop (eliminating the use of certain flag variables)

  To skip the remainder of the switch structure
- After break, the program continues with the first statement after the structure
- continue:

  It skips remaining statements and proceeds with the next iteration of the loop

Examples

- See program to find the first n prime numbers
- Notice:
  - Indentation: used for easy readability of the code
  - Comments: are used to help the reader
  - Variables declared within a loop or an if exist only inside!

User defined functions

Functions

- Building blocks

  Allow complicated programs to be divided into manageable pieces

- Some advantages of functions:

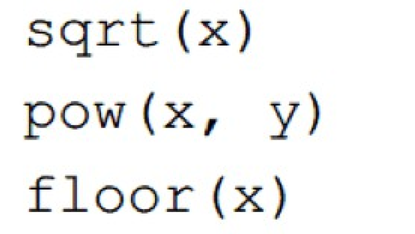
  A programmer can focus on just that part of the program and construct it, debug it, and perfect it

  Different people can work on different functions simultaneously

  Can be re-used (even in different programs)

  Enhance program readability

- Examples: pre-defined mathematical functions

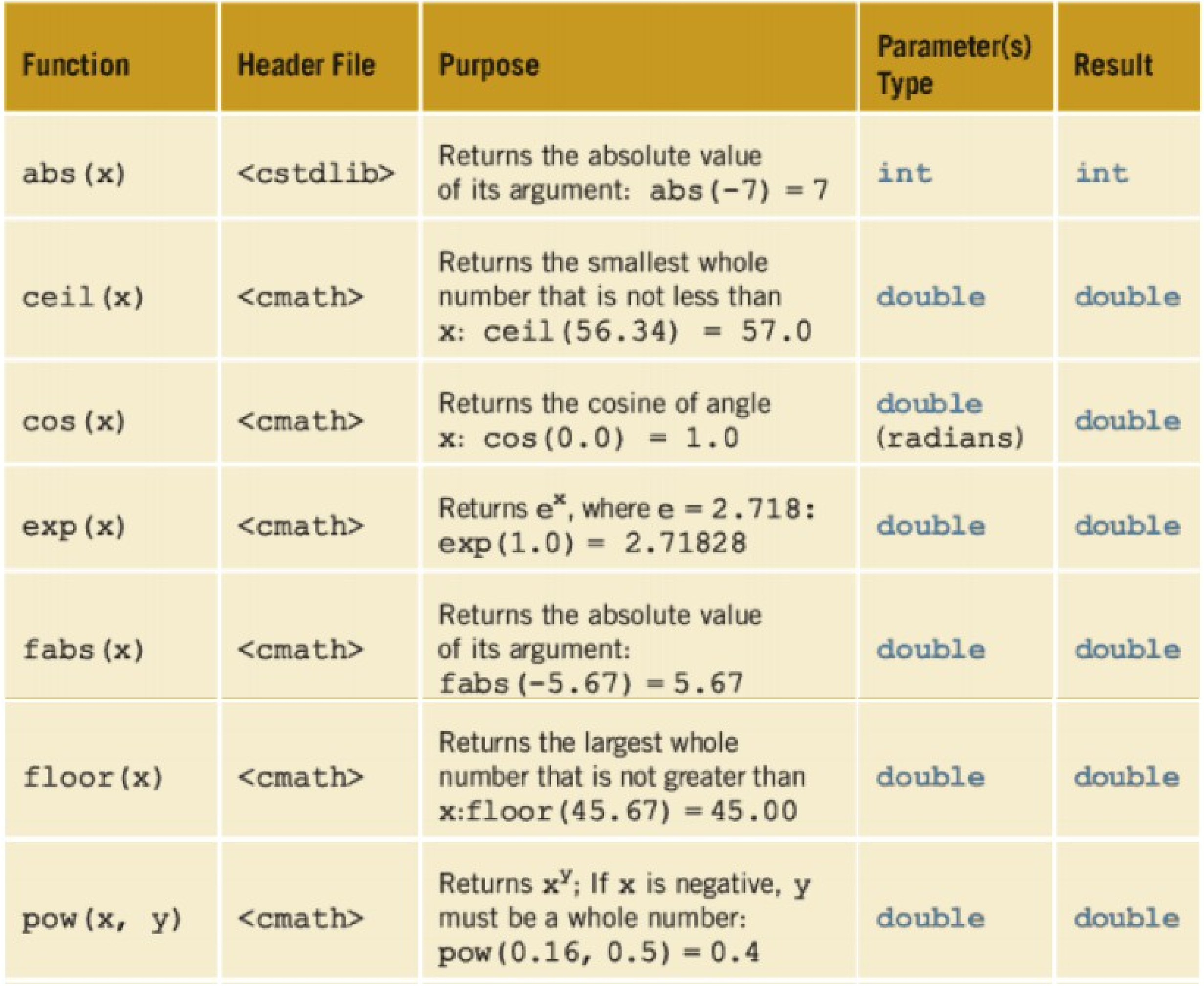


```
sqrt(x)
pow(x, y)
floor(x)
```

#include <cmath>

Examples: maths functions

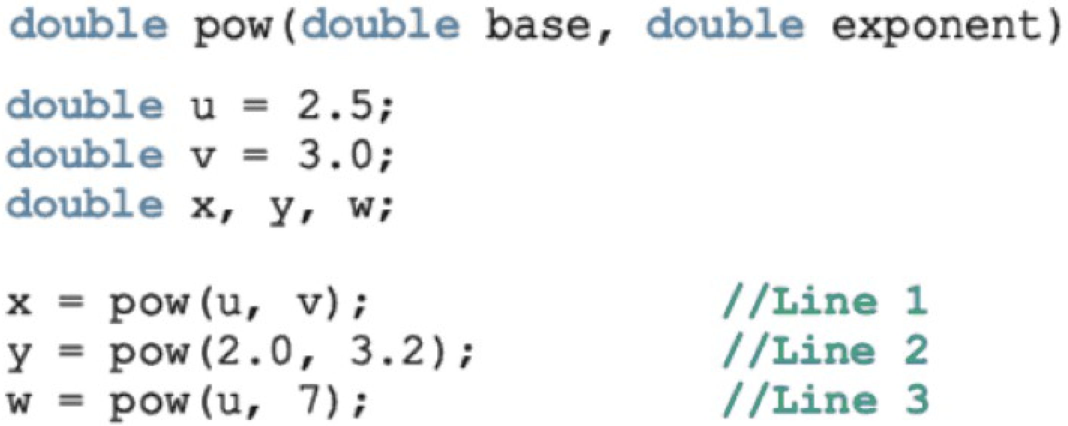
TABLE 6-1 Predefined Functions



Function	Header File	Purpose	Parameter(s) Type	Result
abs(x)	<cstdlib>	Returns the absolute value of its argument: abs(-7) = 7	int	int
ceil(x)	<cmath>	Returns the smallest whole number that is not less than x: ceil(56.34) = 57.0	double	double
cos(x)	<cmath>	Returns the cosine of angle x: cos(0.0) = 1.0	double (radians)	double
exp(x)	<cmath>	Returns e^x, where e = 2.718: exp(1.0) = 2.71828	double	double
fabs(x)	<cmath>	Returns the absolute value of its argument: fabs(-5.67) = 5.67	double	double
floor(x)	<cmath>	Returns the largest whole number that is not greater than x:floor(45.67) = 45.00	double	double
pow(x, y)	<cmath>	Returns x^y; If x is negative, y must be a whole number: pow(0.16, 0.5) = 0.4	double	double

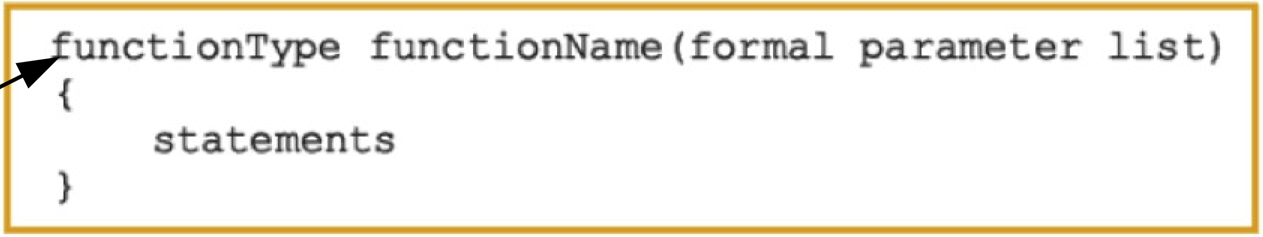
Functions

- Example on how to use them:



```
double pow(double base, double exponent)

double u = 2.5;
double v = 3.0;
double x, y, w;

x = pow(u, v);                  //Line 1
y = pow(2.0, 3.2);              //Line 2
w = pow(u, 7);                  //Line 3
```

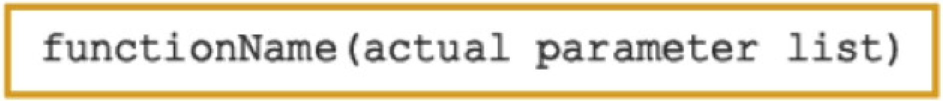
- Creating your own functions:



```
functionType functionName(formal parameter list)
{
    statements
}
```

Data type or return type

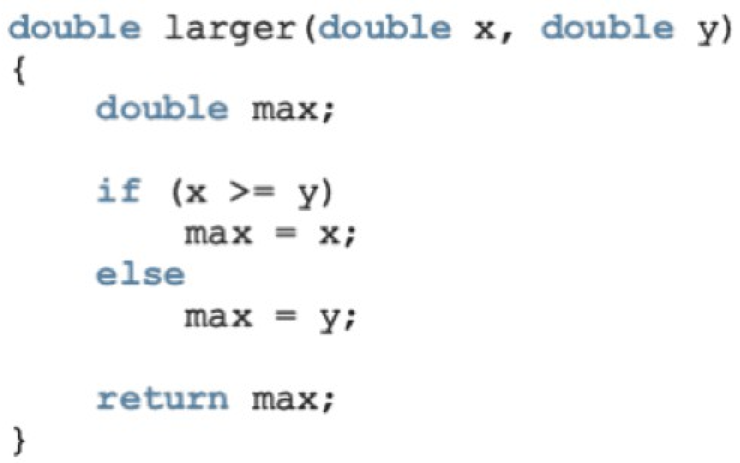
- Call to your function:



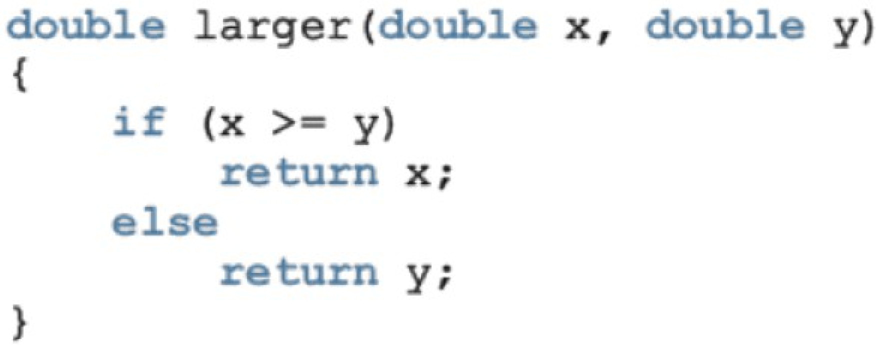
```
functionName(actual parameter list)
```

return

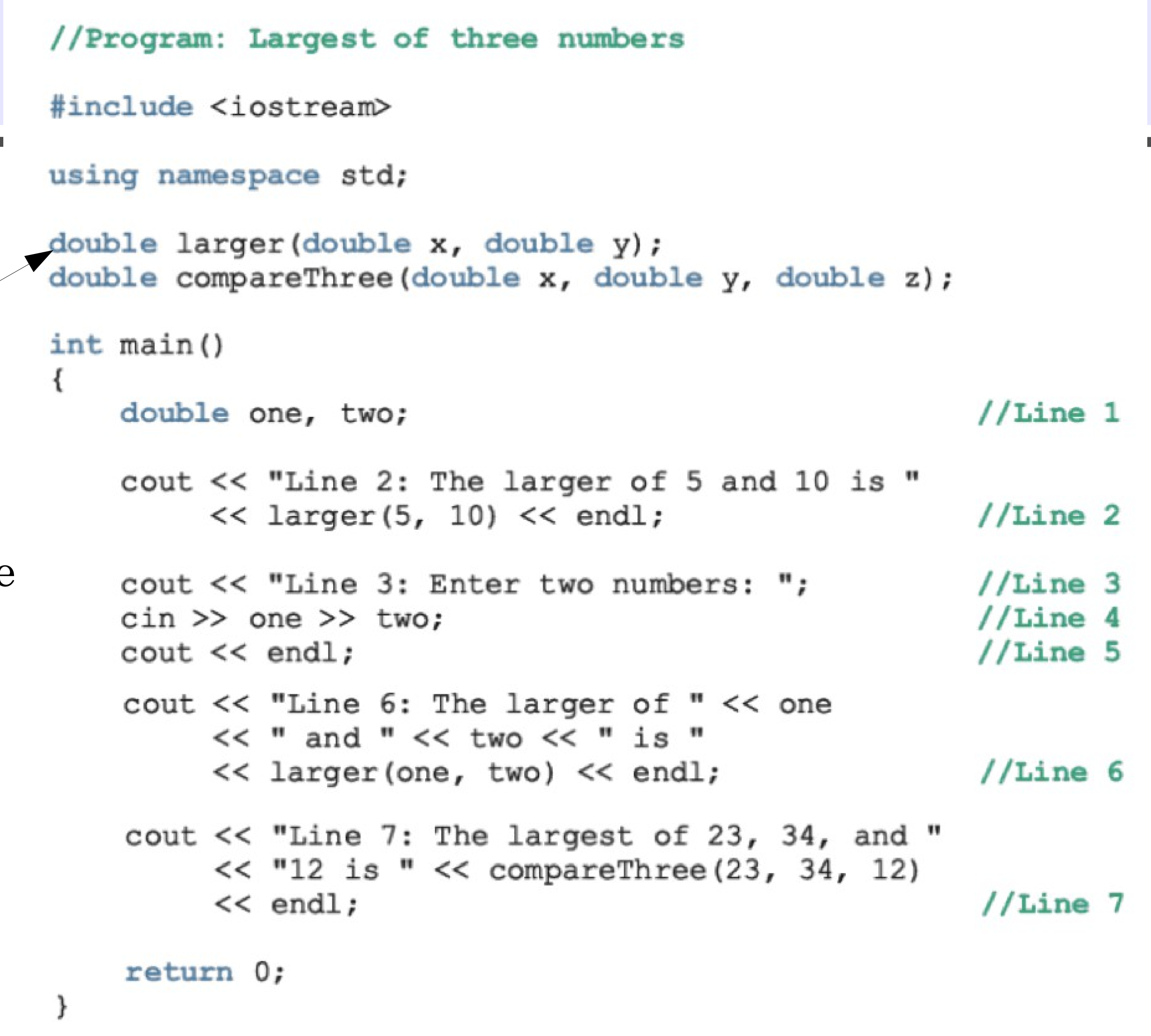
- The function returns a value via the return statement
  - It passes this value outside the function via the return statement
  - The function immediately terminates after the return statement



```
double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}
```

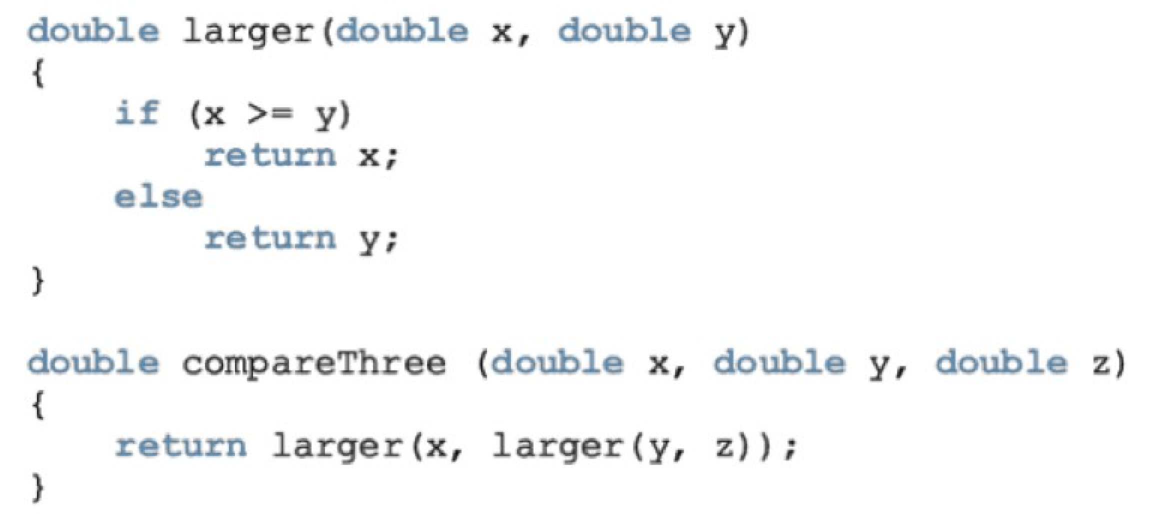


```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```



```cpp
//Program: Largest of three numbers

#include <iostream>

using namespace std;

double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                  //Line 2

    cout << "Line 3: Enter two numbers: ";          //Line 3
    cin >> one >> two;                              //Line 4
    cout << endl;                                   //Line 5

    cout << "Line 6: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;               //Line 6

    cout << "Line 7: The largest of 23, 34, and "
         << "12 is " << compareThree(23, 34, 12)
         << endl;                                   //Line 7

    return 0;
}
```

Declared here
Implemented
later on the
same/other file

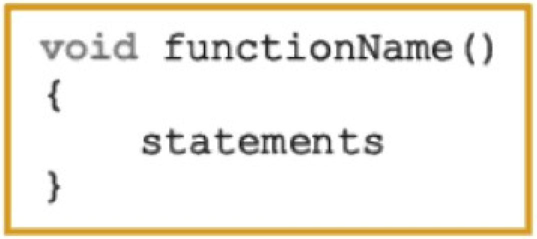


```
double larger(double x, double y)
{
      if (x >= y)
            return x;
      else
            return y;
}

double compareThree (double x, double y, double z)
{
      return larger(x, larger(y, z));
}
```

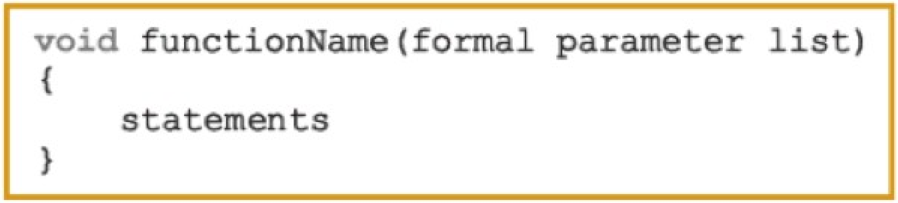
- Execution begins at the first statement in the function main
- Other functions executed only when called
- A function call results in transfer of control to the first statement in the body of the called function
- After the last statement of a function, control passed back to the point immediately following the function call
- After executing the function the returned value replaces the function call statement

Void function

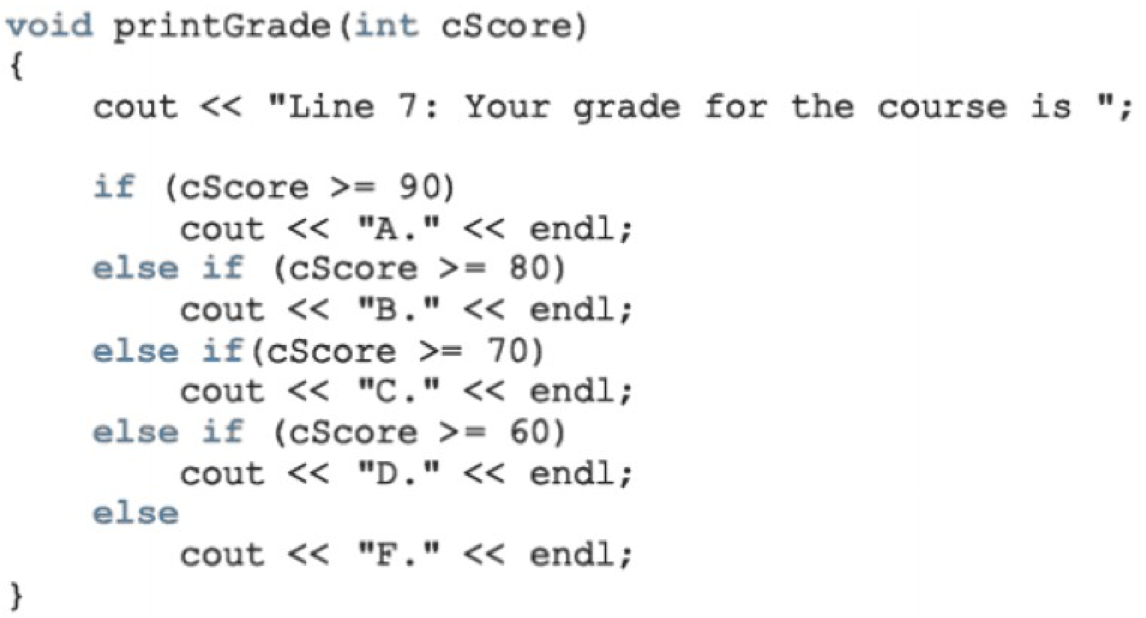
- Does not have a return type



```
void functionName()
{
     statements
}
```



```
void functionName(formal parameter list)
{
     statements
}
```



```
void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is ";

    if (cScore >= 90)
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if(cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}
```

Function overloading

- In a C++ program, several functions can have the same name

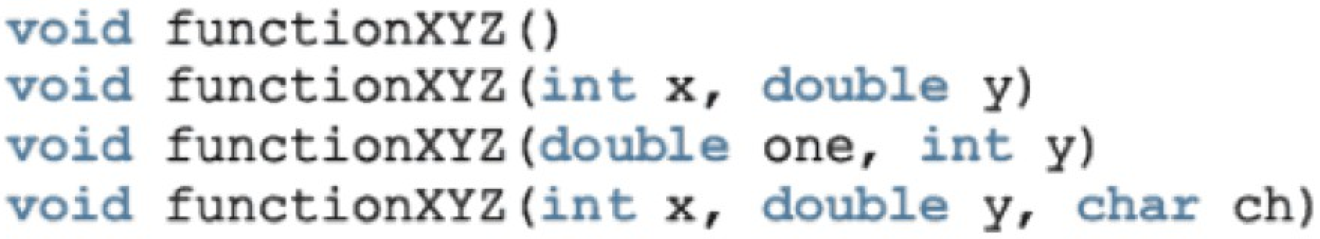
  Function overloading or overloading a function name

- Two functions are said to have different formal parameter lists if both functions have:

  A different number of formal parameters, or

  The data type of the formal parameters, in the order you list them, must differ in at least one position

- The signature of a function consists of the function name and its formal parameter list



```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

Arrays

Arrays

- Store multiple values together as an unit:

```
type arrayName[dimension];
```

```
int arr[4] = { 6, 0, 9, 6 };
```

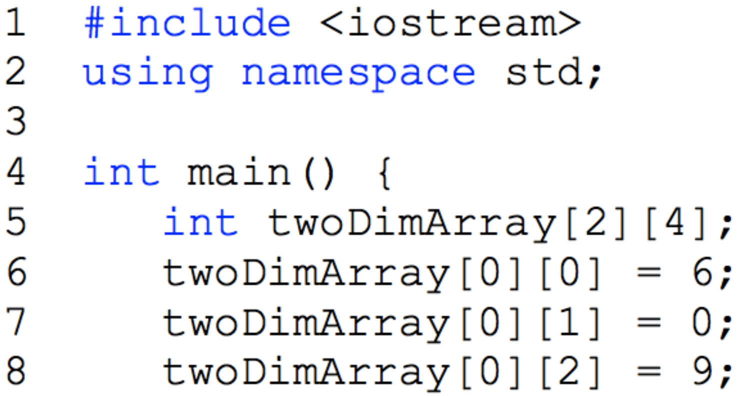
```
int arr[4];

arr[0] = 6;
arr[1] = 0;
arr[2] = 9;
arr[3] = 6;
```

```
int arr[] = { 6, 0, 9, 6, 2, 0, 1, 1 };
```

- Can have multiple dimensions:

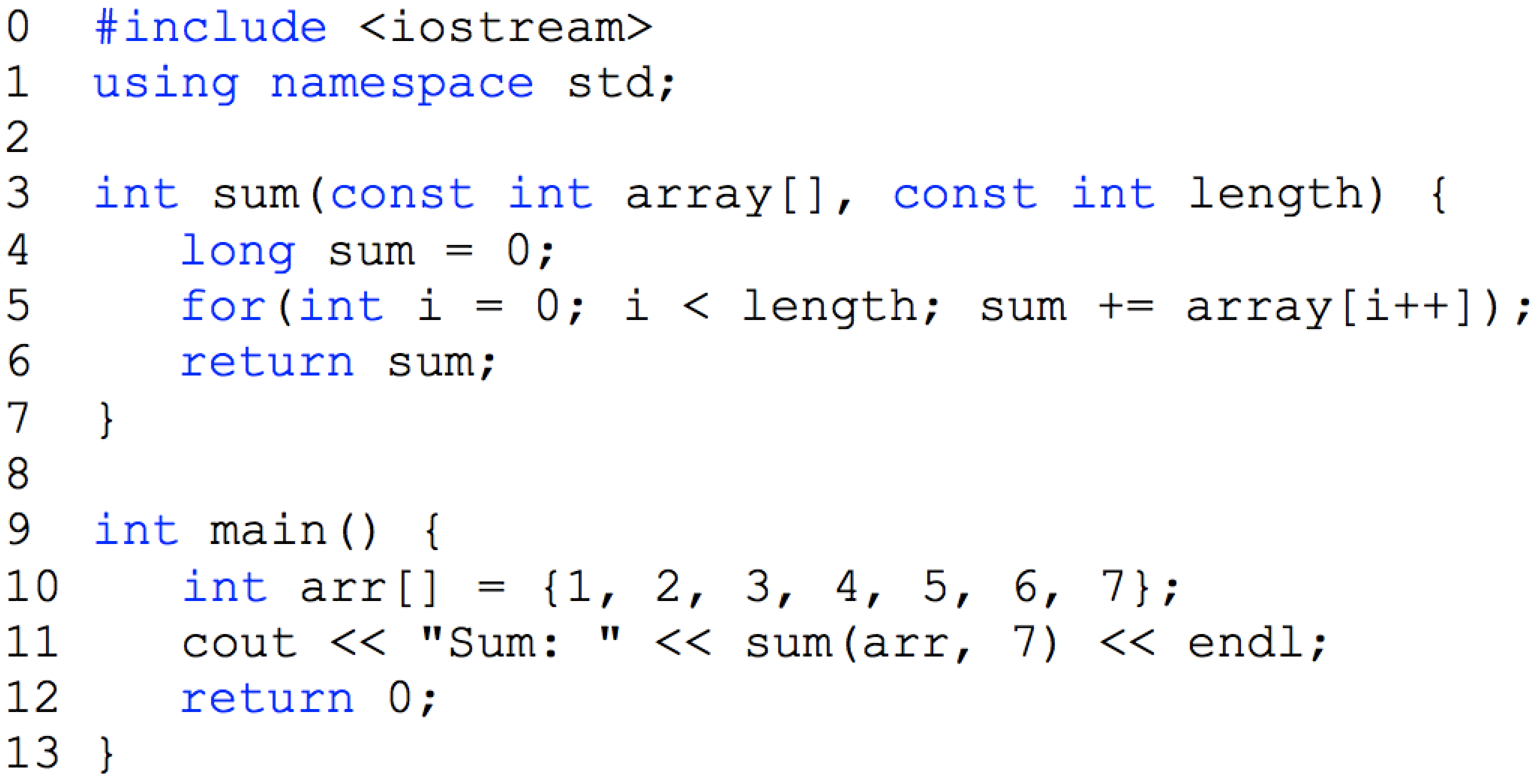
```
type arrayName[dimension1][dimension2];
```

Abstraction: elements in memory are in a simple array!



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

int main() {
    int twoDimArray[2][4];
    twoDimArray[0][0] = 6;
    twoDimArray[0][1] = 0;
    twoDimArray[0][2] = 9;
```

Arrays

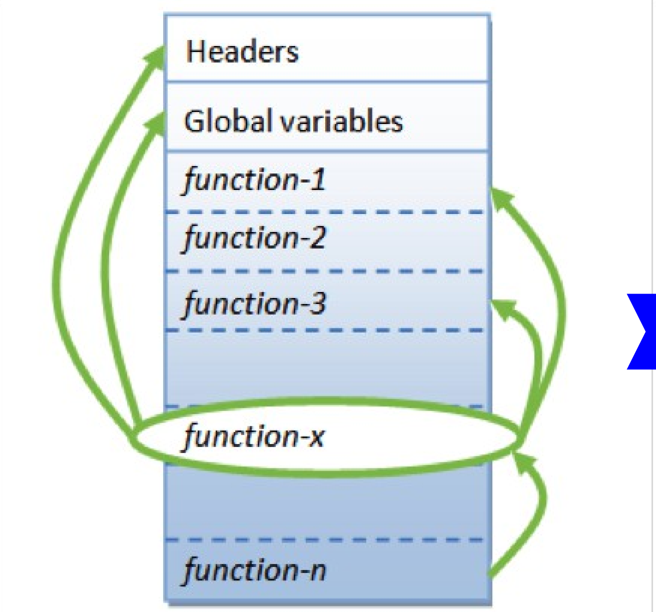


```
#include <iostream>
using namespace std;

int sum(const int array[], const int length) {
   long sum = 0;
   for(int i = 0; i < length; sum += array[i++]);
   return sum;
}

int main() {
   int arr[] = {1, 2, 3, 4, 5, 6, 7};
   cout << "Sum: " << sum(arr, 7) << endl;
   return 0;
}
```

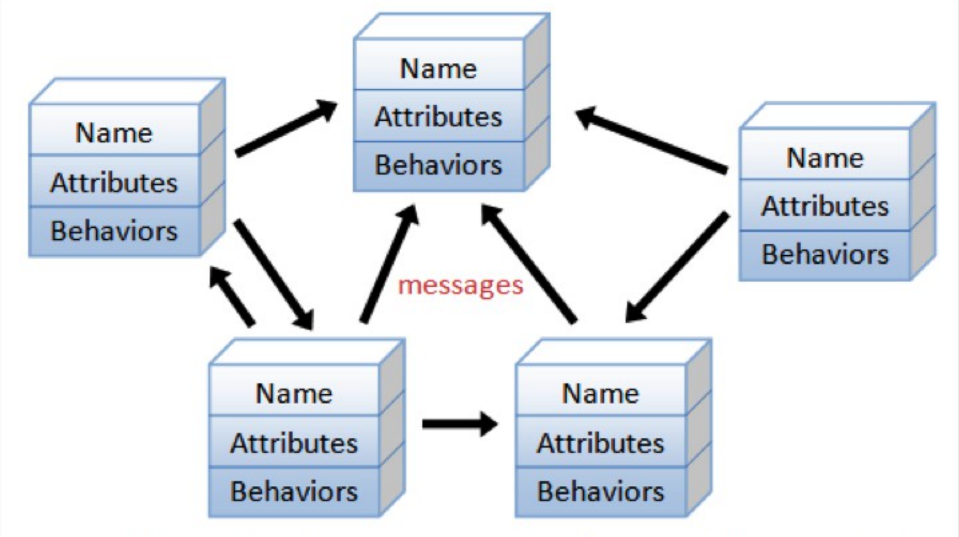
User defined data structures: classes

Object oriented programming

- In procedural programming paradigm programs are made of functions that are frequently not re-usable
  - Likely to reference headers, global variables, …
  - Not suitable for high level of abstraction

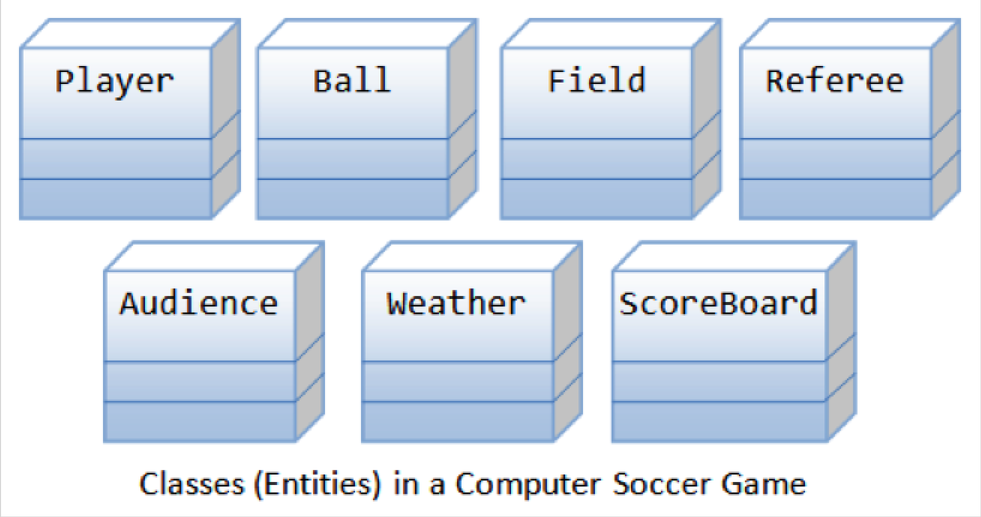


A function (in C) is not well-encapsulated



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Example football game



Classes (Entities) in a Computer Soccer Game

- Static classes but dynamical behaviour
- Player:

  has attributes:

  Name, number, location in the field, …

  Actions: run, kick the ball, stop, …
- Some of this objects, like player, could be re-used for a basketball game!

Object oriented programming

- Ease software design

  Dealing with high-level concepts and abstractions

- Ease software maintenance:

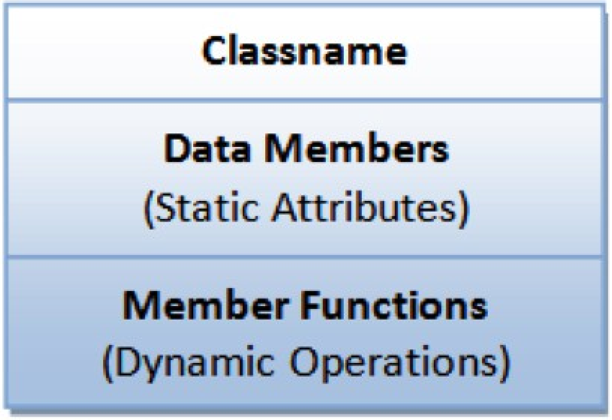
  object-oriented software are easier to understand, therefore easier to test, debug, and maintain.

- Reusable software

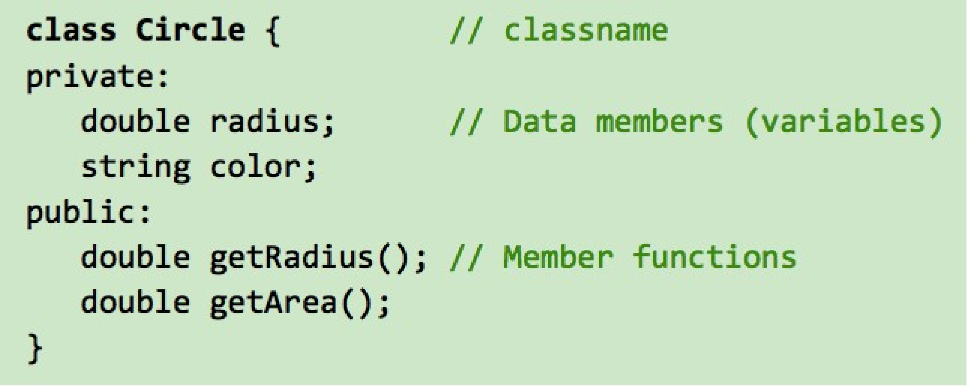
  Use already tested and debugged code

Class definition

- Classname: identifies the class.
- Data Members or Variables (or attributes, states, fields): contains the static attributes of the class.
- Member Functions (or methods, behaviors, operations): contains the dynamic operations of the class.



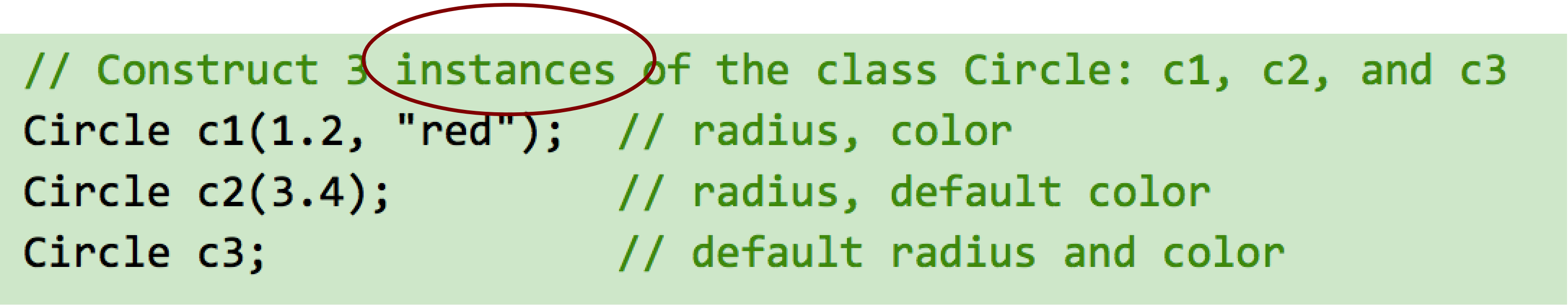
A class is a 3-compartment box encapsulating data and functions



```
class Circle {          // classname
private:
   double radius;       // Data members (variables)
   string color;
public:
   double getRadius(); // Member functions
   double getArea();
}
```

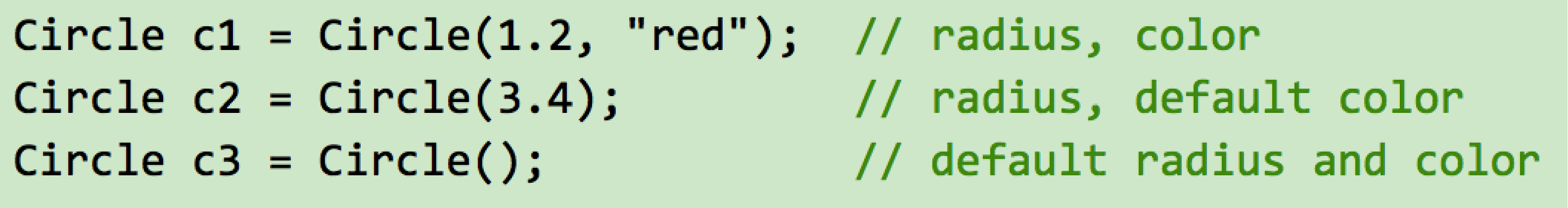
Classes can then be used as your own type of data

Class instantiation



```
// Construct 3 instances of the class Circle: c1, c2, and c3
Circle c1(1.2, "red");  // radius, color
Circle c2(3.4);         // radius, default color
Circle c3;              // default radius and color
```

- Call constructor directly:

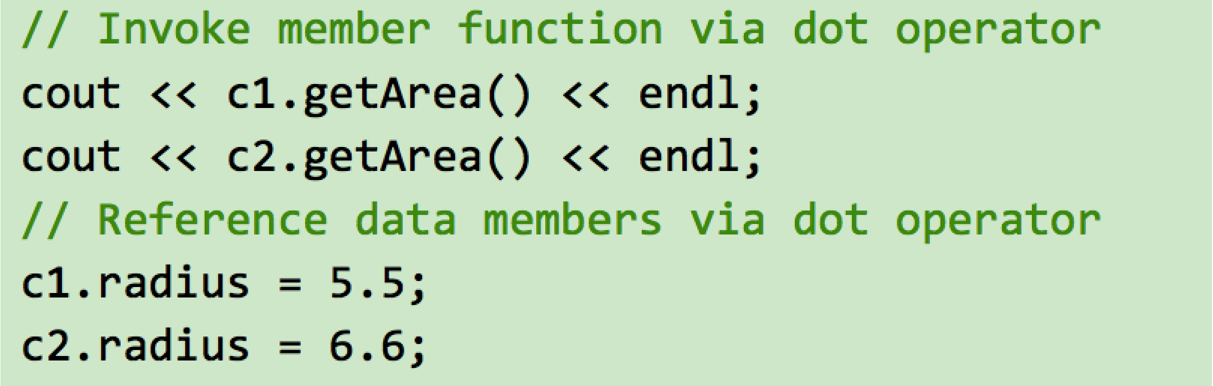


```
Circle c1 = Circle(1.2, "red");  // radius, color
Circle c2 = Circle(3.4);         // radius, default color
Circle c3 = Circle();            // default radius and color
```

- Access members:

anInstance.aData

anInstance.aFunction()

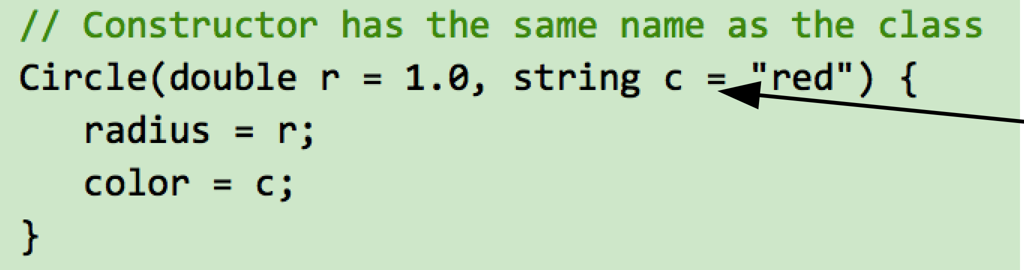


```
// Invoke member function via dot operator
cout << c1.getArea() << endl;
cout << c2.getArea() << endl;
// Reference data members via dot operator
c1.radius = 5.5;
c2.radius = 6.6;
```

Constructor

- Function with the same name as the class
- Used to construct and initialize all the members of the class
- To create an instance of a class you need to call the constructor

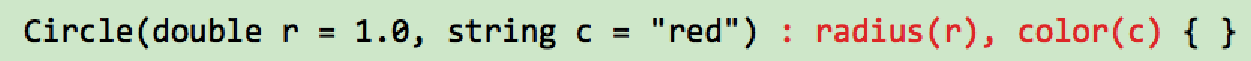
  Can only be called once per instance!
- Has no return type:



```
// Constructor has the same name as the class
Circle(double r = 1.0, string c = "red") {
   radius = r;
   color = c;
}
```

Default argument!

- Alternative syntax:



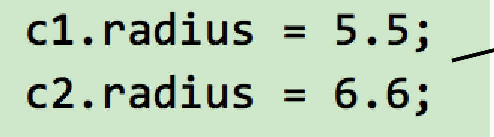
```
Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }
```

Private, public, getters and setters

- Private versus public members

  Private members are only accessible inside the class

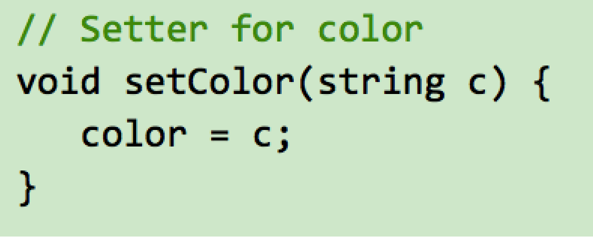
  Public members can be accessed:



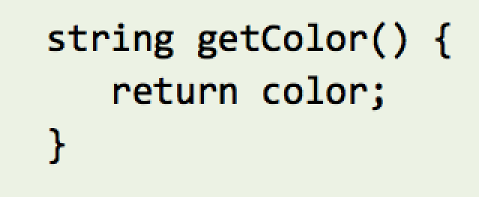
```
c1.radius = 5.5;
c2.radius = 6.6;
```

Only for public members!

- Can use getters and setters:

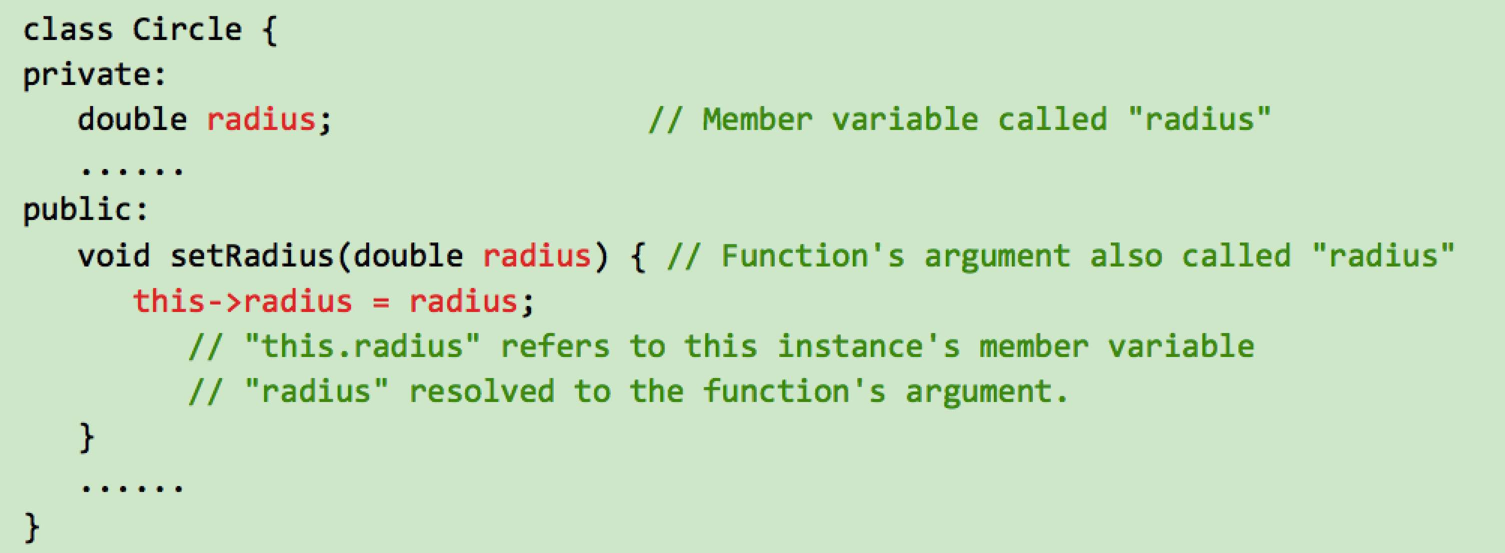


```
// Setter for color
void setColor(string c) {
   color = c;
}
```



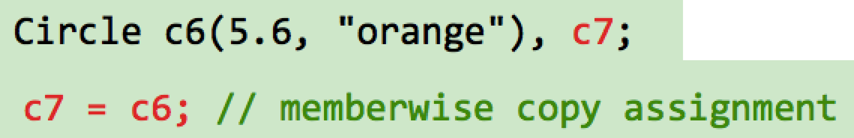
```
string getColor() {
   return color;
}
```

- Keyword this:



```
class Circle {
private:
   double radius;                       // Member variable called "radius"
   ......
public:
   void setRadius(double radius) { // Function's argument also called "radius"
      this->radius = radius;
         // "this.radius" refers to this instance's member variable
         // "radius" resolved to the function's argument.
   }
   ......
}
```

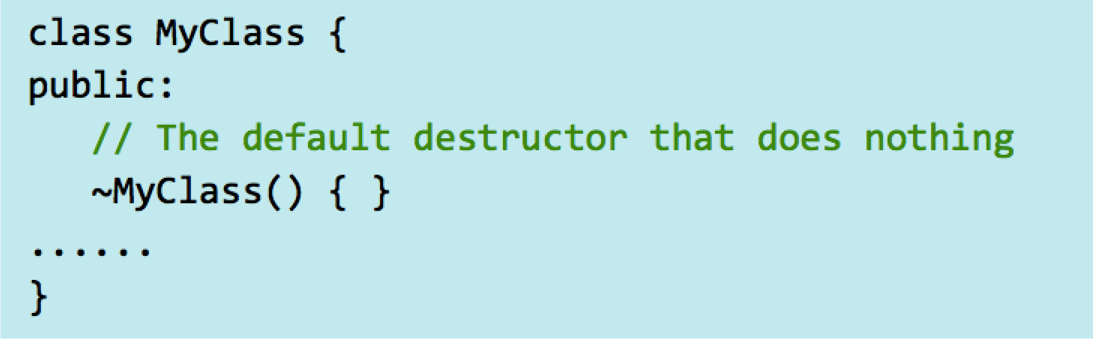
- Assignment operator (=):
   - Provided by the compiler
   - Assign one object to another of the same class via memberwise copy



```
Circle c6(5.6, "orange"), c7;
c7 = c6; // memberwise copy assignment
```

Destructor

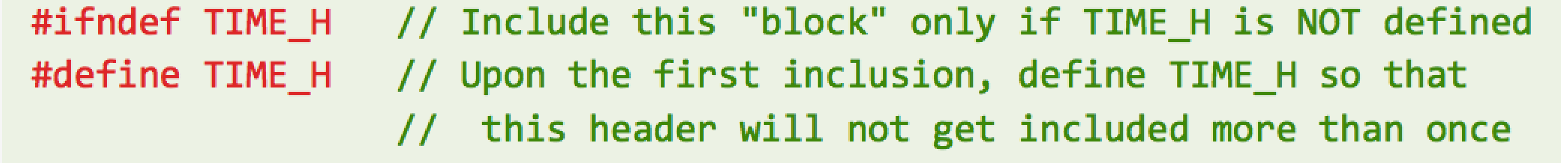
- Special function that has the same name as the classname
  - called implicitly when an object is destroyed
  - It will be very important when using pointers! (next class)



```
class MyClass {
public:
   // The default destructor that does nothing
   ~MyClass() { }
......
}
```

Example: clock class

- Header file contains declaration
- Cpp file contains the implementation
- Pre-processor options:



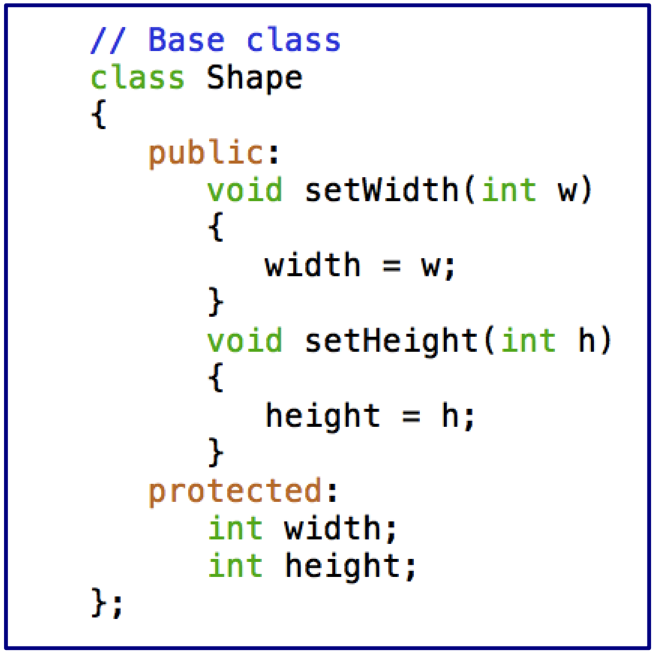
```
#ifndef TIME_H   // Include this "block" only if TIME_H is NOT defined
#define TIME_H   // Upon the first inclusion, define TIME_H so that
                 //  this header will not get included more than once
```

- 3 versions
  - Simple one
  - Using getters and setters
  - With functions to handle exceptions
- Notice the overloaded operators (version 3)

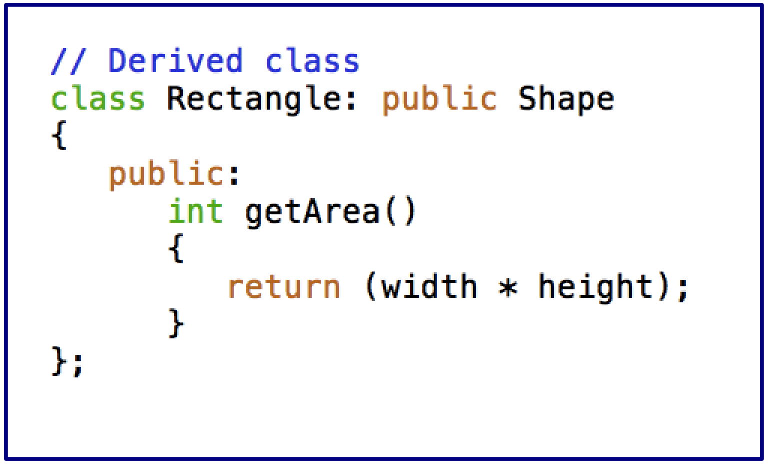
Inheritance

Inheritance

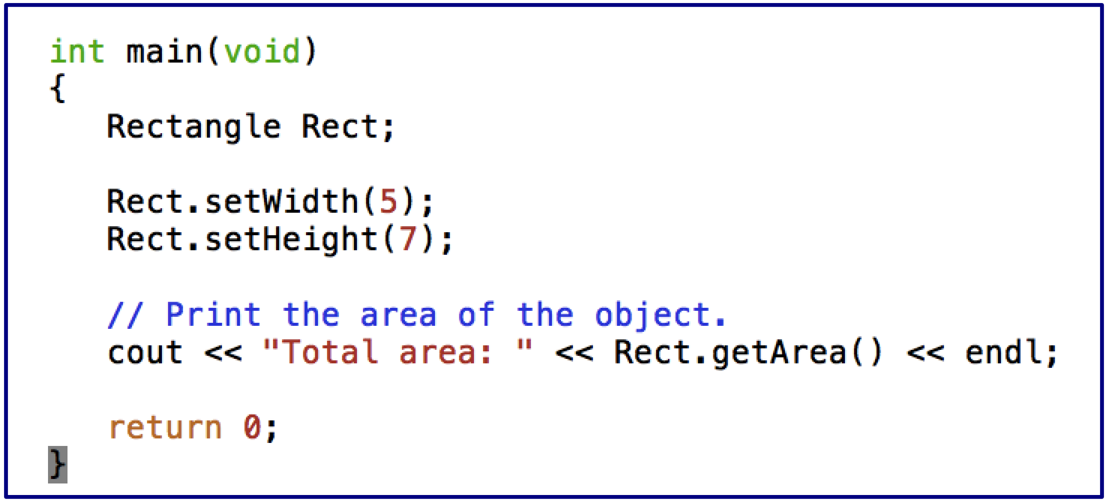
- Example



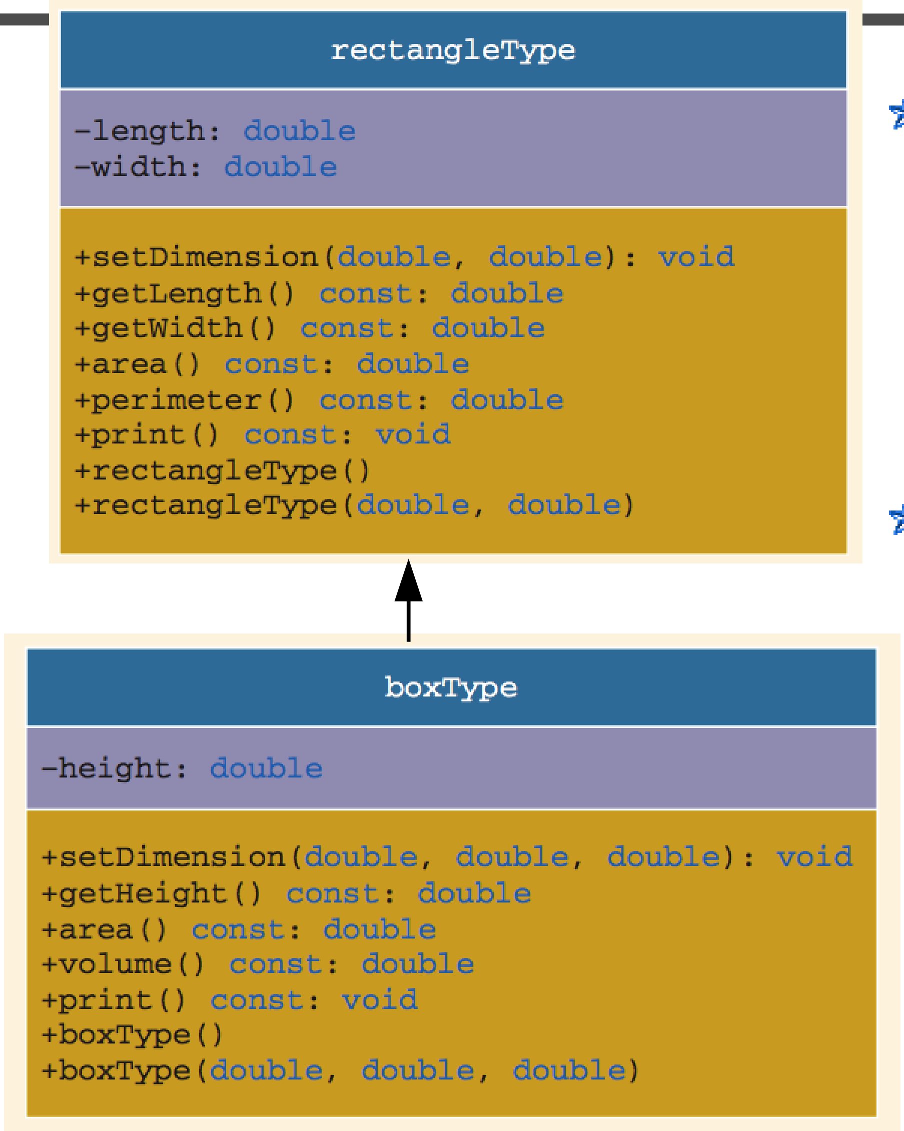
```
// Base class
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};
```



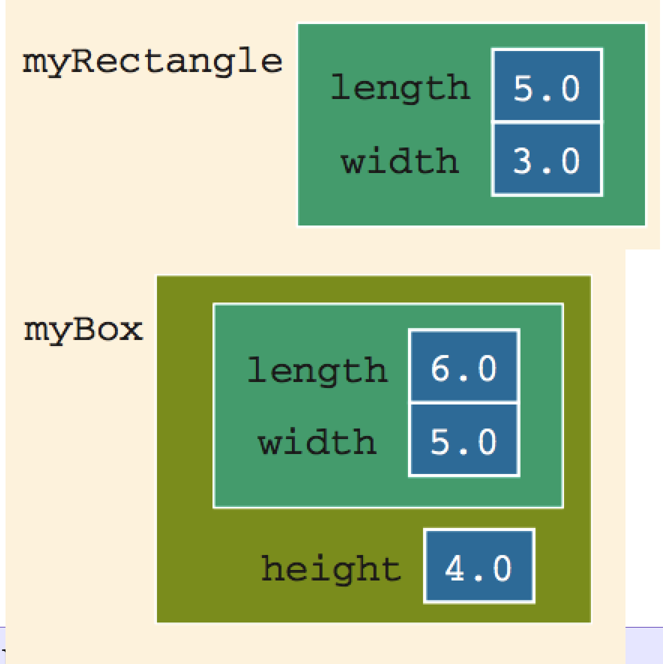
```
// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};
```



```
int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```



```
rectangleType
-length: double
-width: double
+setDimension(double, double): void
+getLength() const: double
+getWidth() const: double
+area() const: double
+perimeter() const: double
+print() const: void
+rectangleType()
+rectangleType(double, double)

boxType
-height: double
+setDimension(double, double, double): void
+getHeight() const: double
+area() const: double
+volume() const: double
+print() const: void
+boxType()
+boxType(double, double, double)
```

- Notice: using UML to define the class structure
  - UML = Unified Modeling Language
  - Very useful to design software
- 

Standard Library

The Standard Library

- Collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself
  - Complex data types: classes
  - Need always an include file
- Examples:
  - Standard input/output (cin, cout)
  - Write/read files
  - Strings: sequences of characters
  - Vector classes
  - …
  - (see www.cplusplus.com)

The std::string

- Programmed defined type used to handle strings of characters

  File to be included: `#include <string>`

  Examples of usage:

```
string str1, str2, str3;

str1 = "Hello"
str2 = "There"

str3 = str1 + ' ' + str2;
```

→ "Hello There"

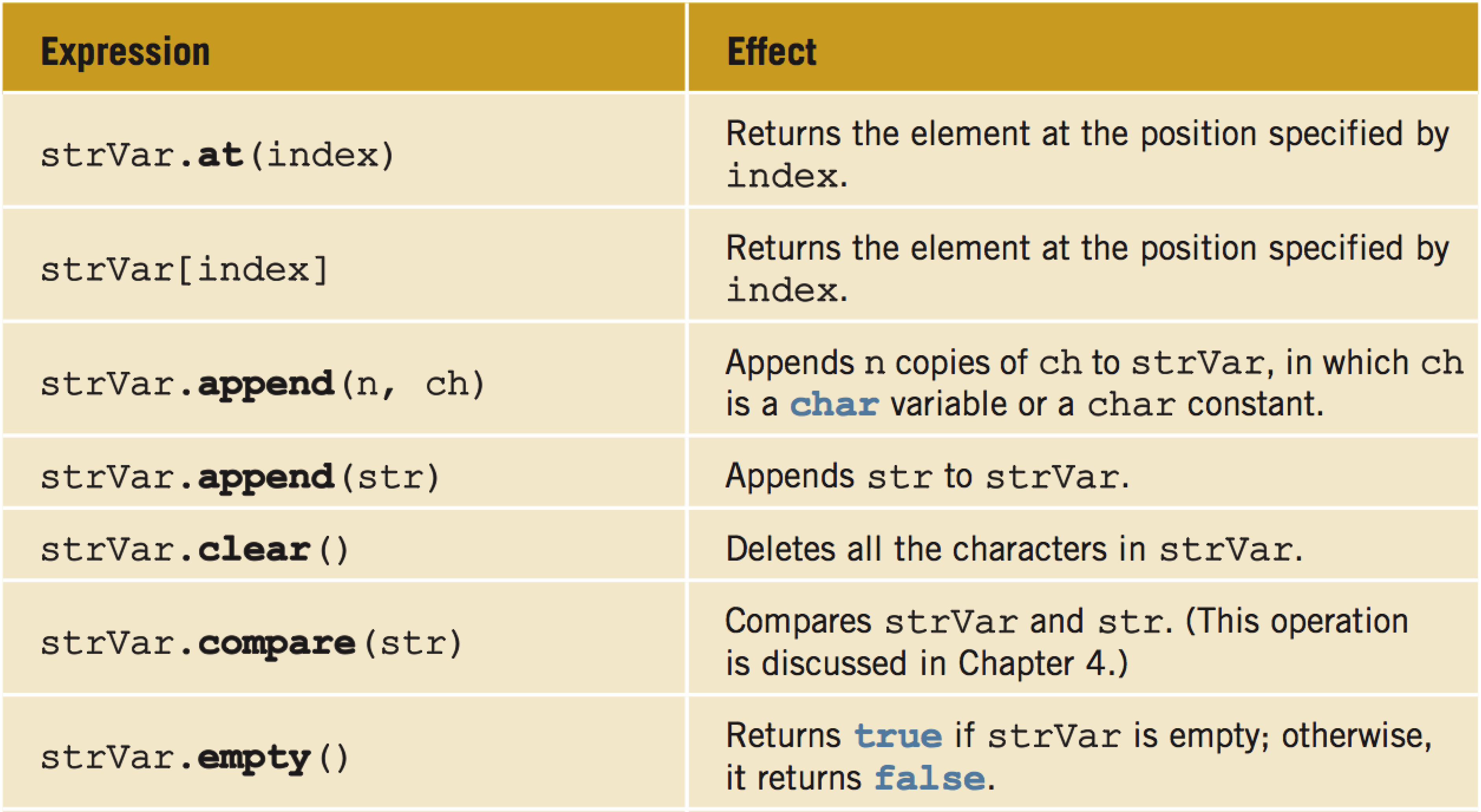
Replace one character:

```
str1 = "Hello there"
str1[6] = 'T';
```

It works as an array!

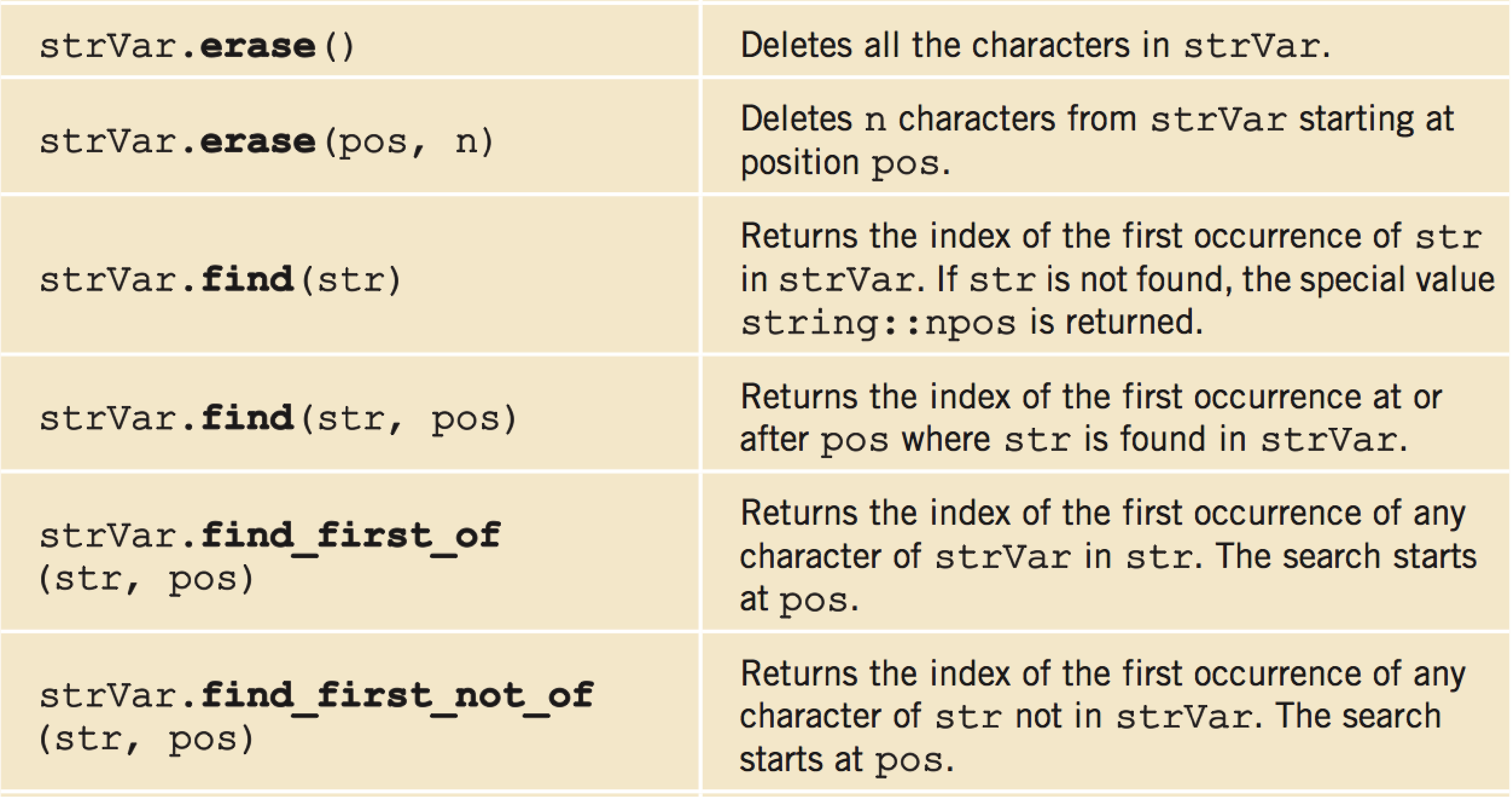
See example program

std::string functions



Expression	Effect
strVar.**at**(index)	Returns the element at the position specified by index.
strVar[index]	Returns the element at the position specified by index.
strVar.**append**(n, ch)	Appends n copies of ch to strVar, in which ch is a **char** variable or a char constant.
strVar.**append**(str)	Appends str to strVar.
strVar.**clear**()	Deletes all the characters in strVar.
strVar.**compare**(str)	Compares strVar and str. (This operation is discussed in Chapter 4.)
strVar.**empty**()	Returns **true** if strVar is empty; otherwise, it returns **false**.

More std::string functions

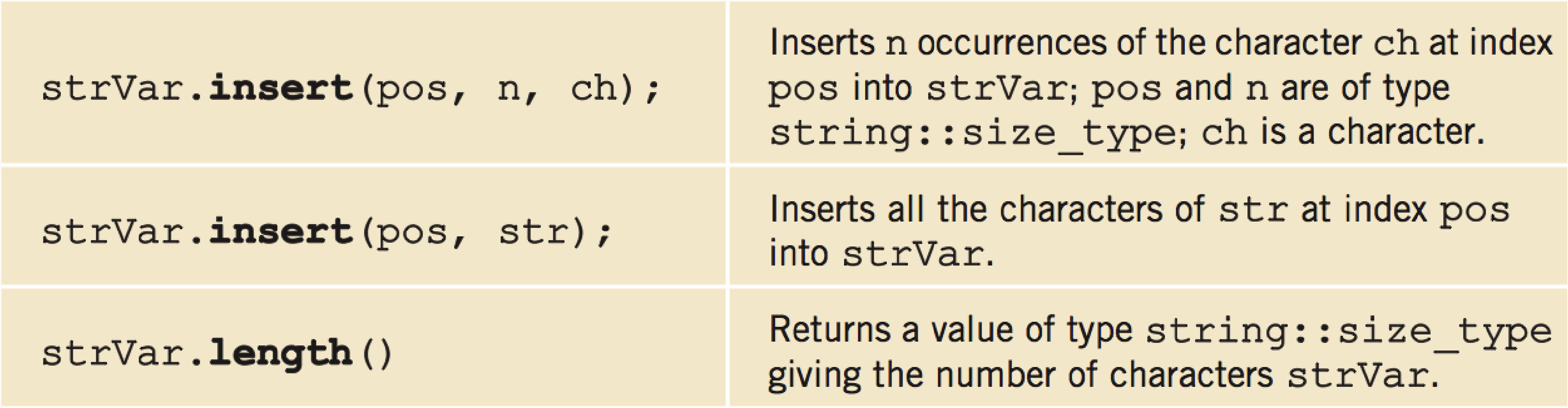


`strVar.`**`erase`**`()`	Deletes all the characters in `strVar`.
`strVar.`**`erase`**`(pos, n)`	Deletes `n` characters from `strVar` starting at position `pos`.
`strVar.`**`find`**`(str)`	Returns the index of the first occurrence of `str` in `strVar`. If `str` is not found, the special value `string::npos` is returned.
`strVar.`**`find`**`(str, pos)`	Returns the index of the first occurrence at or after `pos` where `str` is found in `strVar`.

`strVar.`**`find_first_of`**
`(str, pos)`
	Returns the index of the first occurrence of any character of `strVar` in `str`. The search starts at `pos`.

`strVar.`**`find_first_not_of`**
`(str, pos)`
	Returns the index of the first occurrence of any character of `str` not in `strVar`. The search starts at `pos`.

More std::string functions



`strVar.`**`insert`**`(pos, n, ch);`	Inserts `n` occurrences of the character `ch` at index `pos` into `strVar`; `pos` and `n` are of type `string::size_type`; `ch` is a character.
`strVar.`**`insert`**`(pos, str);`	Inserts all the characters of `str` at index `pos` into `strVar`.
`strVar.`**`length`**`()`	Returns a value of type `string::size_type` giving the number of characters `strVar`.

- See also www.cplusplus.com

Pointers & references

Why do we need pointers?

- Most people say
  - Oooohhh! They are a powerful tool!
- But... why?
  - Allow you to modify data inside a function
  - Allow you to dynamically allocate memory
    - You don't need to know in advance how much data your program is going to handle
- Example:
  - A function that changes the value of a variable
  - See example 2.

Pointers

- A pointer is a variable that stores/manipulates addresses in memory

  It's possible values are the memory allocations

- Declaring a pointer:

```
dataType *identifier;
```

Examples

```
int *p;
int*   p;
int *  p;
```

Be careful:

```
int*  p,  q;
```

only the first one is a pointer

```
int *p, *q;
```

both are pointers

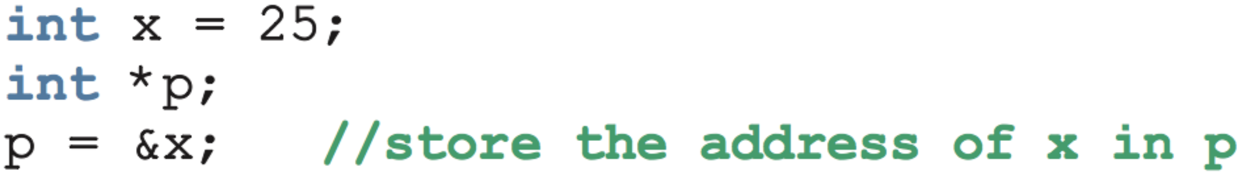
p, q: can store the memory address of any int variable

- Address operator &:

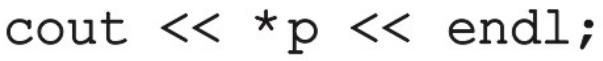
```
int x;
p = &x;
```

Pointers (II)

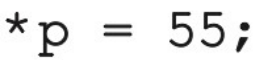
- Dereference operator *:



```
int x = 25;
int *p;
p = &x;    //store the address of x in p
```



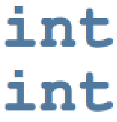
```
cout << *p << endl;
```



```
 *p = 55;
```

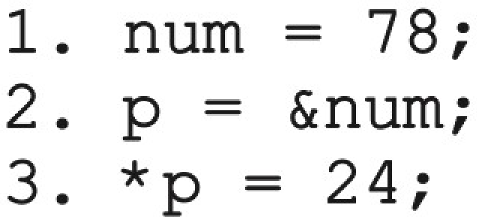
Accesses the value stored in the memory pointed to by p

- Example:



```
int *p;
int num;
```

Attention! Allocates memory for the pointer p (an address) not for *p



```
1. num = 78;
2. p = &num;
3. *p = 24;
```

Pointers (II)

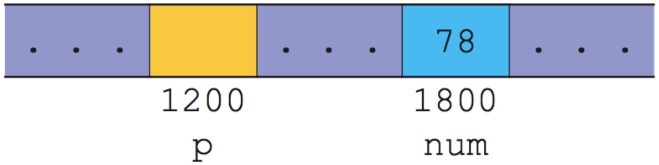
- Dereference operator *:

```
int x = 25;
int *p;
p = &x;    //store the address of x in p

cout << *p << endl;

 *p = 55;
```

Accesses the value stored in the memory pointed to by p

- Example:

```
int *p;
int num;
1. num = 78;
2. p = &num;
3. *p = 24;
```



Pointers (II)

- Dereference operator *:

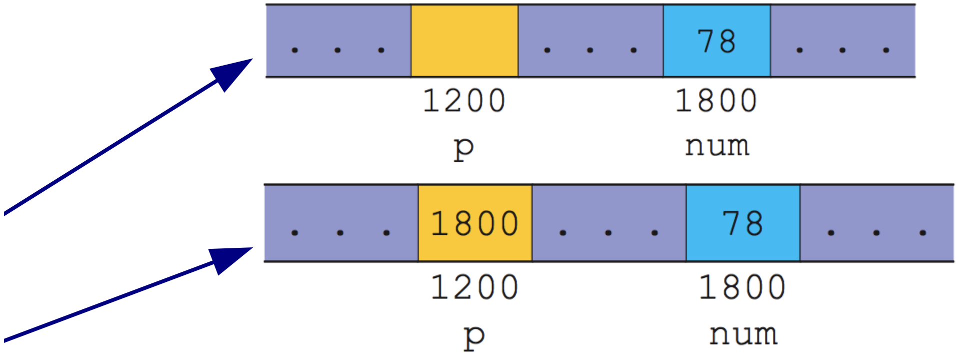
```
int x = 25;
int *p;
p = &x;    //store the address of x in p

cout << *p << endl;

 *p = 55;
```

Accesses the value stored in the memory pointed to by p

- Example:

```
int *p;
int num;
```

```
1. num = 78;
2. p = &num;
3. *p = 24;
```



. . .		. . .	78	. . .
	1200 p		1800 num	

. . .	1800	. . .	78	. . .
	1200 p		1800 num	

Pointers (II)

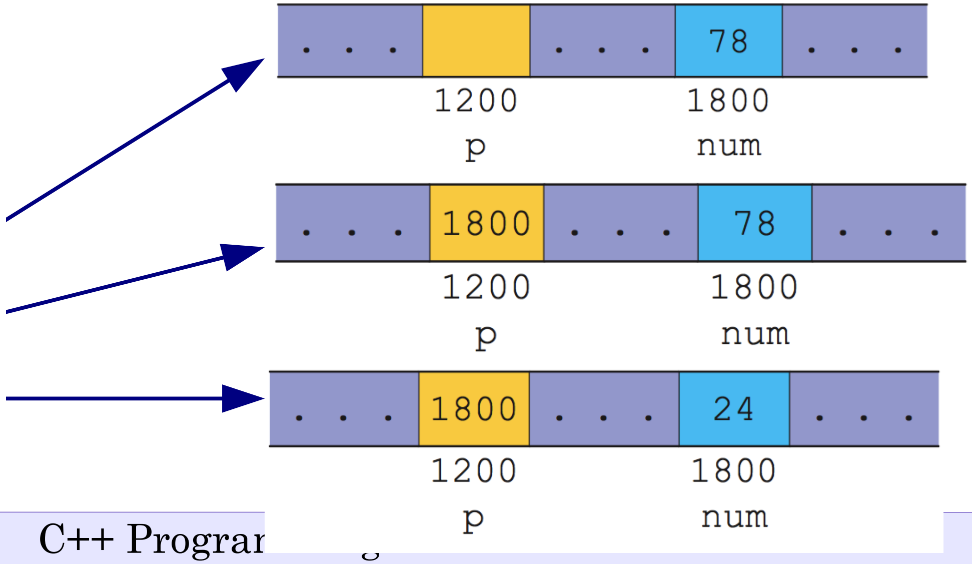
- Dereference operator *:

```
int x = 25;
int *p;
p = &x;    //store the address of x in p

cout << *p << endl;

 *p = 55;
```

Accesses the value stored in the memory pointed to by p

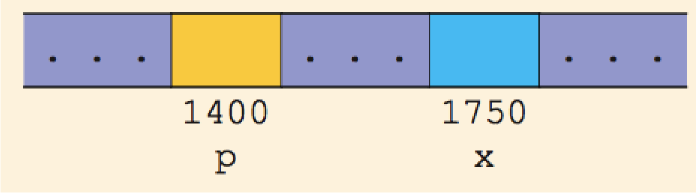
- Example:

```
int *p;
int num;
1. num = 78;
2. p = &num;
3. *p = 24;
```



*p, &p and p

```
int *p;
int x;
```



&p	1400	p	? (unknown)	*p	Does not exist (undefined)	&x	1750	x	? (unknown)

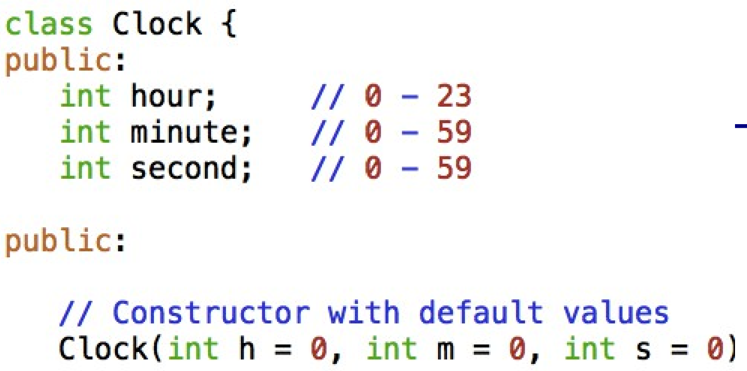
```
x = 50;
p = &x;
*p = 38;
```

&p	1400	p	? (unknown)	*p	Does not exist (undefined)	&x	1750	x	50
&p	1400	p	1750	*p	50	&x	1750	x	50
&p	1400	p	1750	*p	38	&x	1750	x	38

- See Example3

Pointers to classes

- You can also declare pointers to classes
- Remembering the clock class from last lecture:



```
class Clock {
public:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59

public:

   // Constructor with default values
   Clock(int h = 0, int m = 0, int s = 0)
```

```
Clock *myTime;
(*myTime).hour = 10;
(*myTime).print();
cout << (*myTime).hour
     << endl;
```

- Attention! The access operator . has preference

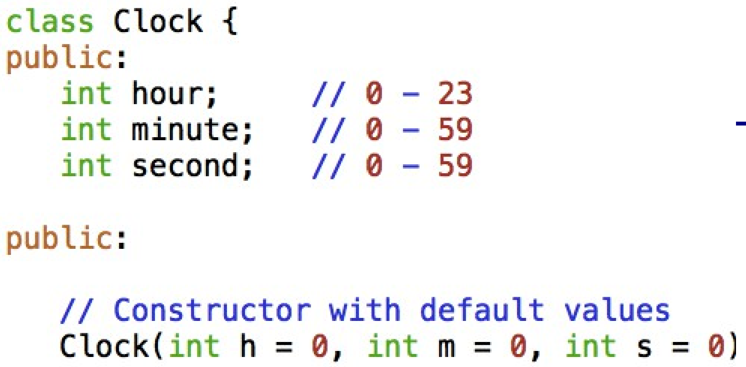
  Use () before the access operator .

`*myTime.hour = 10;` if hour were a pointer, would access its content

Pointers to classes

- To avoid problems: operator ->

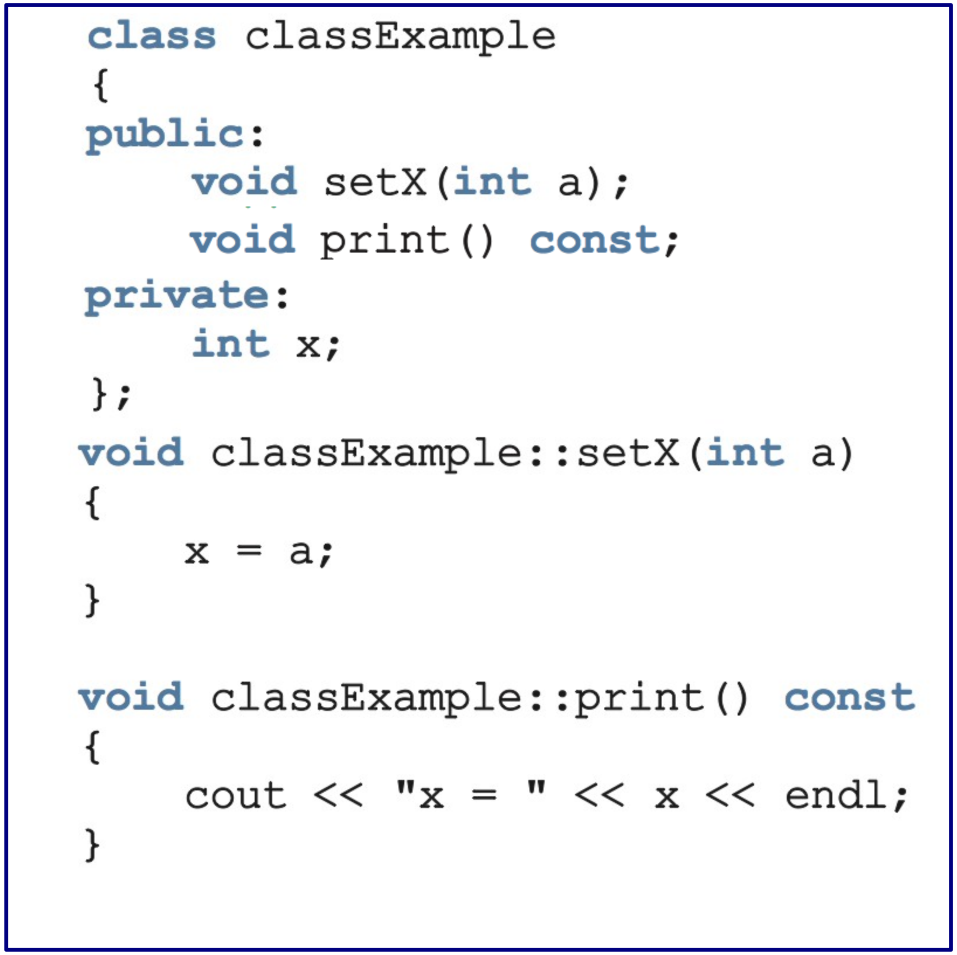
```
pointerVariableName->classMemberName
```



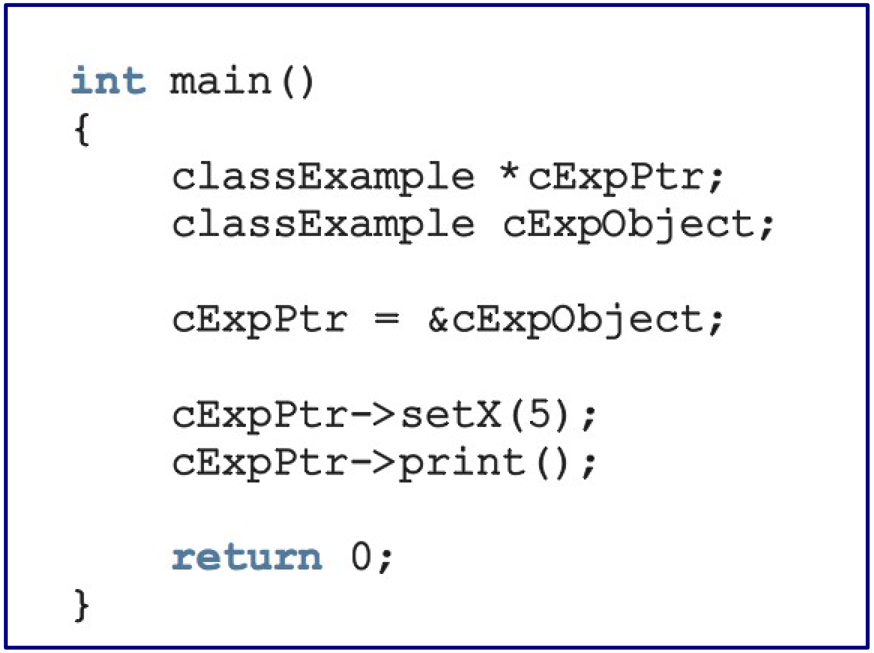
```
class Clock {
public:
   int hour;      // 0 - 23
   int minute;    // 0 - 59
   int second;    // 0 - 59

public:

   // Constructor with default values
   Clock(int h = 0, int m = 0, int s = 0)
```

```
Clock *myTime;
myTime->hour = 10;
myTime->print();
cout << myTime->hour
     << endl;
```

Example



```
class classExample
{
public:
     void setX(int a);
     void print() const;
private:
     int x;
};
void classExample::setX(int a)
{
     x = a;
}

void classExample::print() const
{
     cout << "x = " << x << endl;
}
```

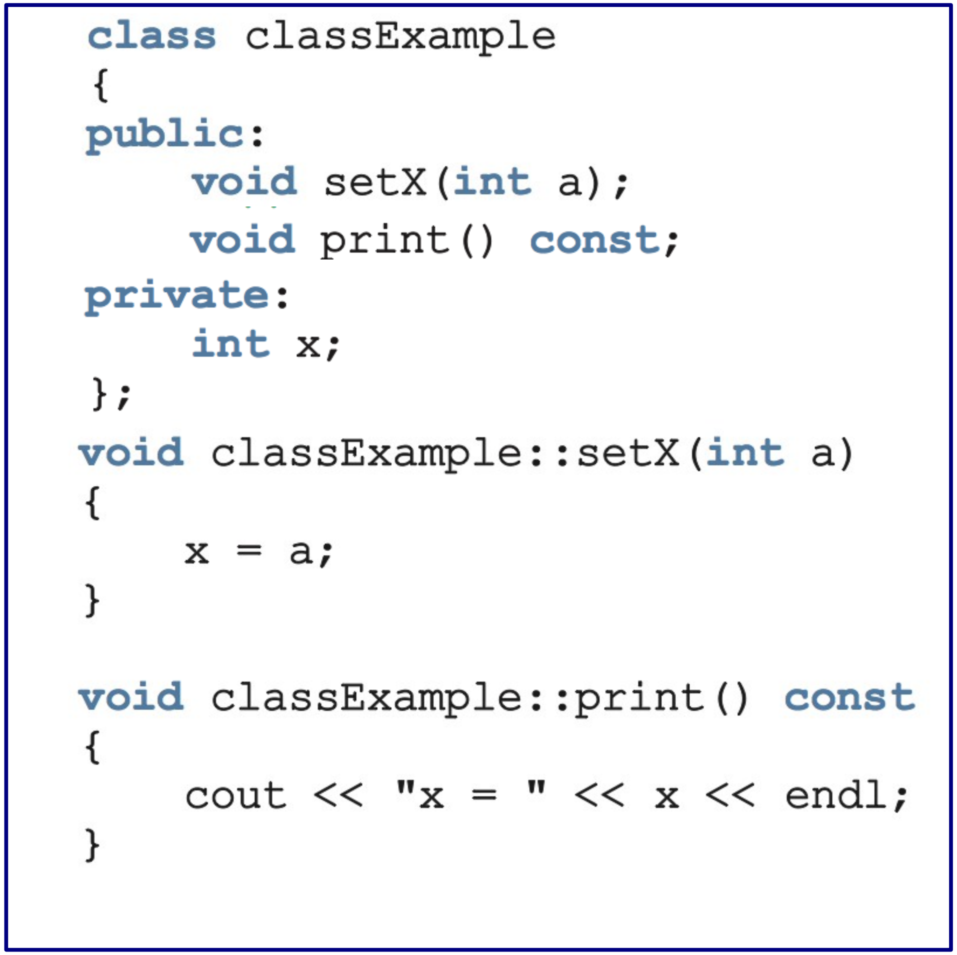


```
int main()
{
     classExample *cExpPtr;
     classExample cExpObject;

     cExpPtr = &cExpObject;

     cExpPtr->setX(5);
     cExpPtr->print();

     return 0;
}
```

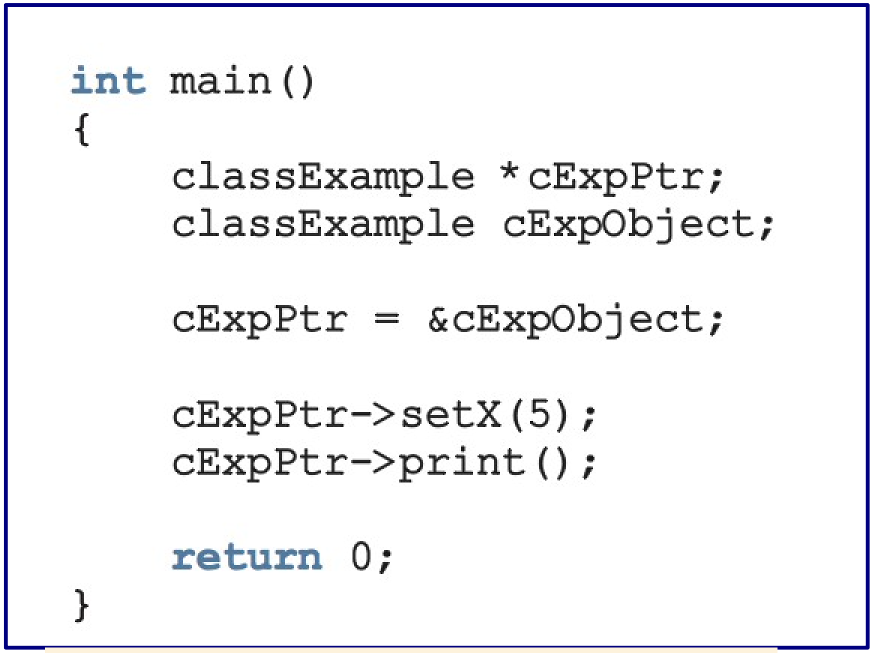
- Output:

```
x = 5
```

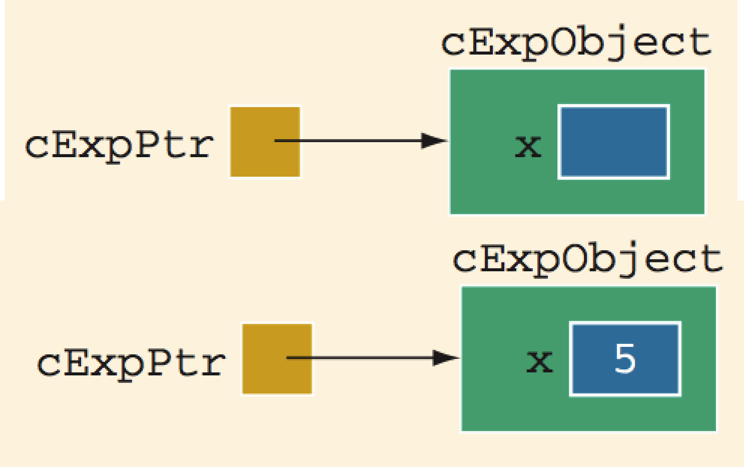
Example



```
class classExample
{
public:
     void setX(int a);
     void print() const;
private:
     int x;
};
void classExample::setX(int a)
{
     x = a;
}

void classExample::print() const
{
     cout << "x = " << x << endl;
}
```



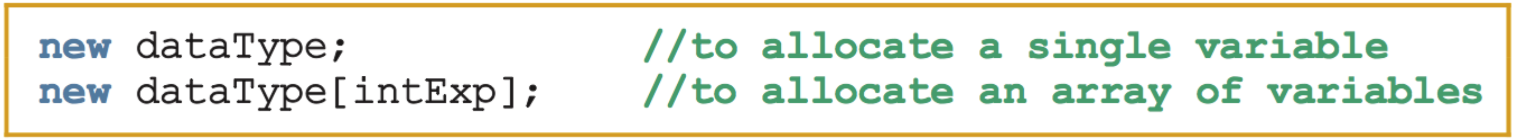
```
int main()
{
     classExample *cExpPtr;
     classExample cExpObject;

     cExpPtr = &cExpObject;

     cExpPtr->setX(5);
     cExpPtr->print();

     return 0;
}
```



Initialization of pointer variables

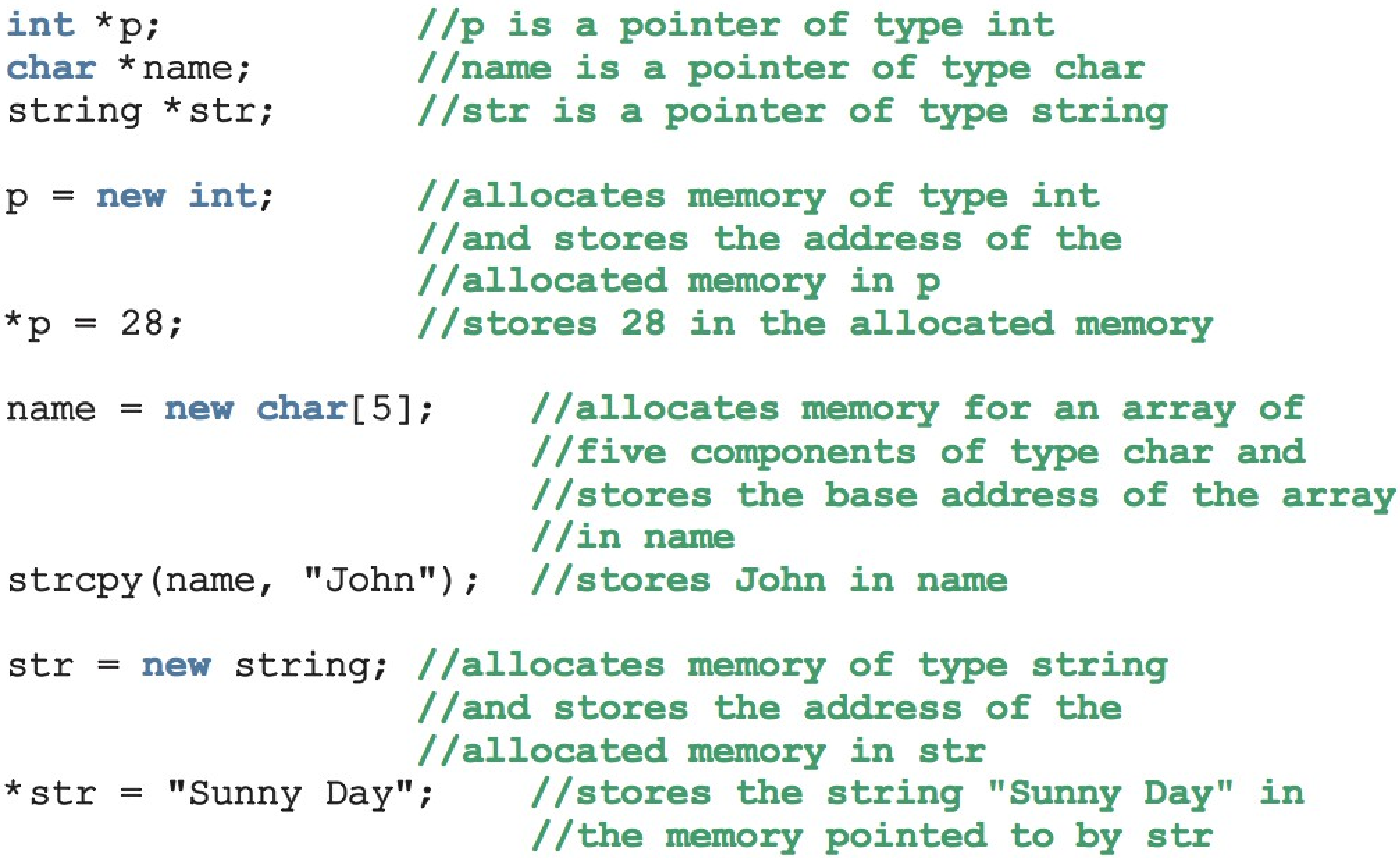
```
p = NULL;
p = 0;
```

- Pointer variables must be initialized
  - Point to nothing: `0, NULL`
- Pointers manipulate data in existing memory spaces
  - Why are they useful?
- Dynamic allocation of memory: the `new` operator



```
new dataType;               //to allocate a single variable
new dataType[intExp];       //to allocate an array of variables
```

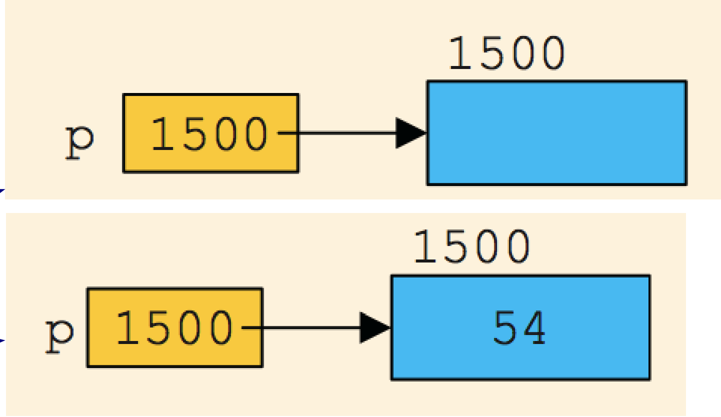
Examples: operator `new`



```
int *p;           //p is a pointer of type int
char *name;       //name is a pointer of type char
string *str;      //str is a pointer of type string

p = new int;      //allocates memory of type int
                  //and stores the address of the
                  //allocated memory in p
*p = 28;          //stores 28 in the allocated memory

name = new char[5];    //allocates memory for an array of
                       //five components of type char and
                       //stores the base address of the array
                       //in name
strcpy(name, "John");  //stores John in name

str = new string; //allocates memory of type string
                  //and stores the address of the
                  //allocated memory in str
*str = "Sunny Day";    //stores the string "Sunny Day" in
                       //the memory pointed to by str
```

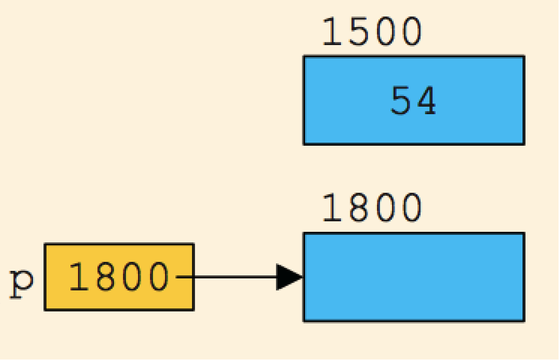
Operator `delete`

```
int *p;

p = new int;
*p = 54;
p = new int;
*p = 73;
```



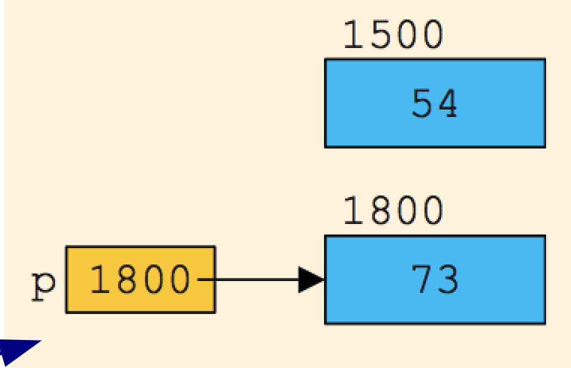
Operator `delete`

```
int *p;

p = new int;
*p = 54;
p = new int;
*p = 73;
```



Operator `delete`

```
int *p;

p = new int;
*p = 54;
p = new int;
*p = 73;
```



- Memory was allocated twice

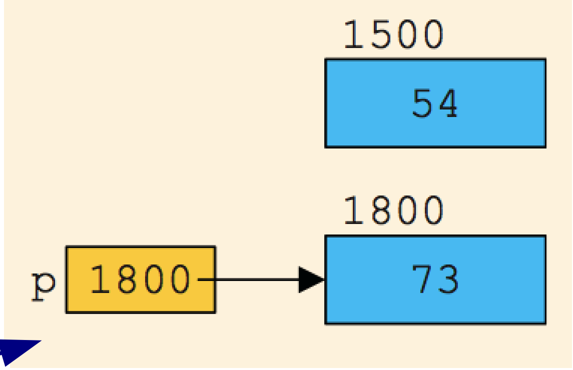
  The memory address `1500` can't be used any more but it cannot be accessed either because there is no pointer to it

- If repeated many times may consume all available memory!

  Memory leak

Operator `delete`

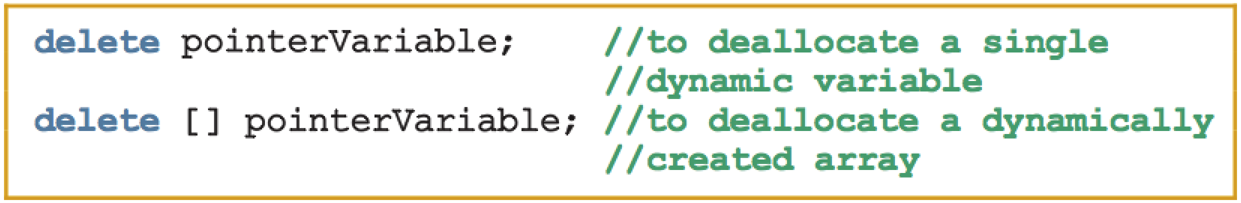
```
int *p;

p = new int;
*p = 54;
p = new int;
*p = 73;
```



- Memory was allocated twice

  The memory address 1500 can't be used any more but it cannot be accessed either because there is no pointer to it

- Use delete operator



```
delete pointerVariable;     //to deallocate a single
                            //dynamic variable
delete [] pointerVariable;  //to deallocate a dynamically
                            //created array
```

Pointer operations

```
int *p, *q;
```

```
p = q;
```

copy operator (copies memory addresses)

```
p == q
```

logical operator (true if both point to the same memory address)

```
p++;

p = p + 1;
```

Increment the memory address by one (i.e. points to the next memory space of size int, in this case)

Arrays and pointers

- Dynamic array:

```
int *p;
p = new int[10];
*p = 25;
p++;
*p = 35;
```

Creates an array of size 10

Stores the value 25 in the first element

Advances to the next memory address (second element)

Stores the value 25 in the second element

Equivalent to

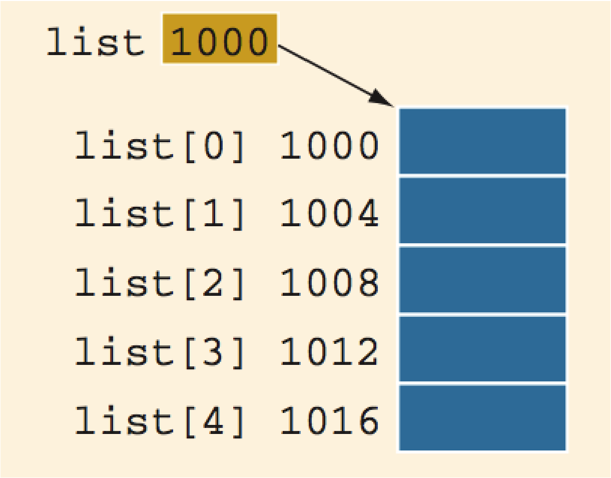
```
p[0] = 25;
p[1] = 35;
```

- Static array:

```
int list[5];
```

`list`: memory address of the first element

`list` is a pointer but the memory address it points to cannot be changed during the program execution



list 1000

list[0] 1000

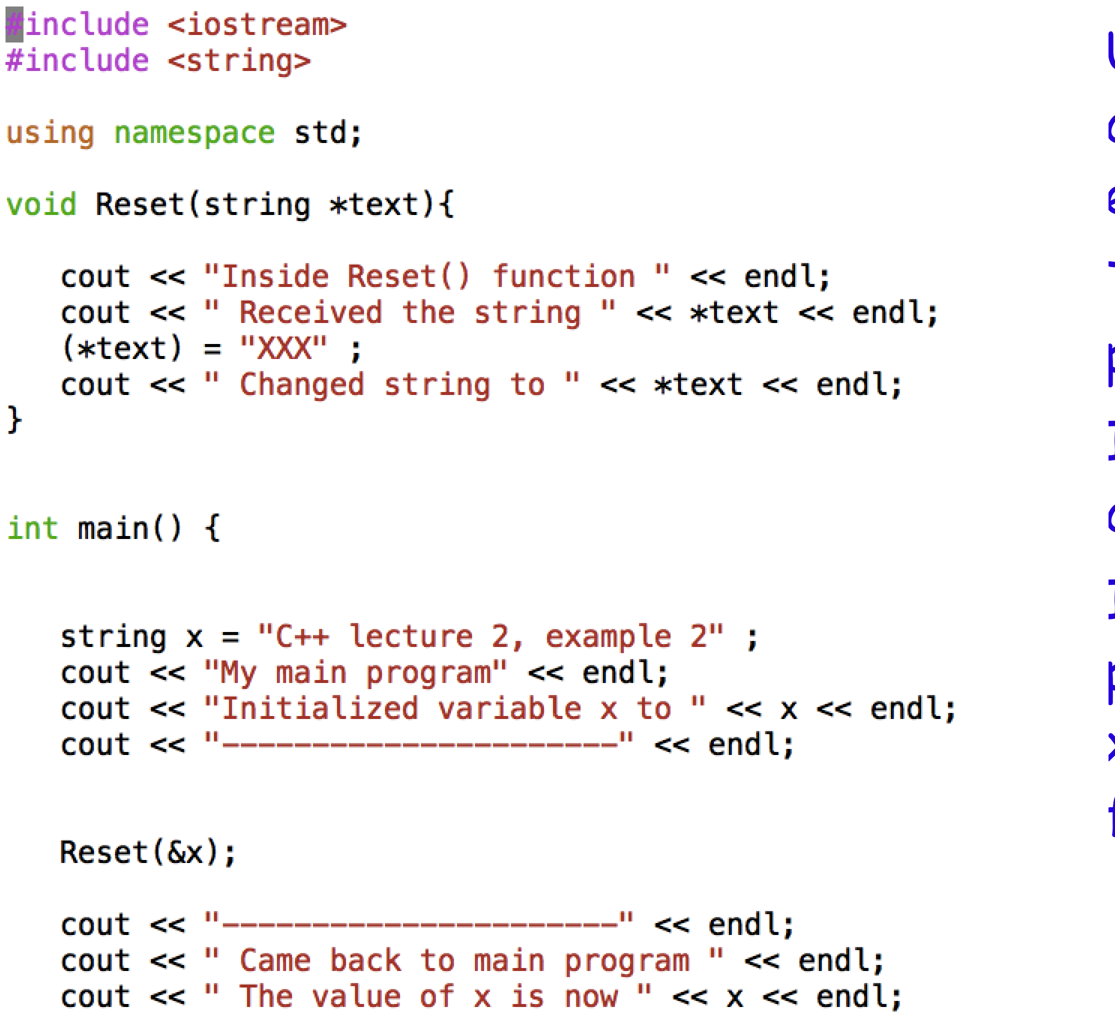
list[1] 1004

list[2] 1008

list[3] 1012

list[4] 1016

Coming back to example 2



```
#include <iostream>
#include <string>

using namespace std;

void Reset(string *text){

   cout << "Inside Reset() function " << endl;
   cout << " Received the string " << *text << endl;
   (*text) = "XXX" ;
   cout << " Changed string to " << *text << endl;
}


int main() {


   string x = "C++ lecture 2, example 2" ;
   cout << "My main program" << endl;
   cout << "Initialized variable x to " << x << endl;
   cout << "----------------------" << endl;


   Reset(&x);

   cout << "----------------------" << endl;
   cout << " Came back to main program " << endl;
   cout << " The value of x is now " << x << endl;
```

Using pointers, we can correctly implement the example 2

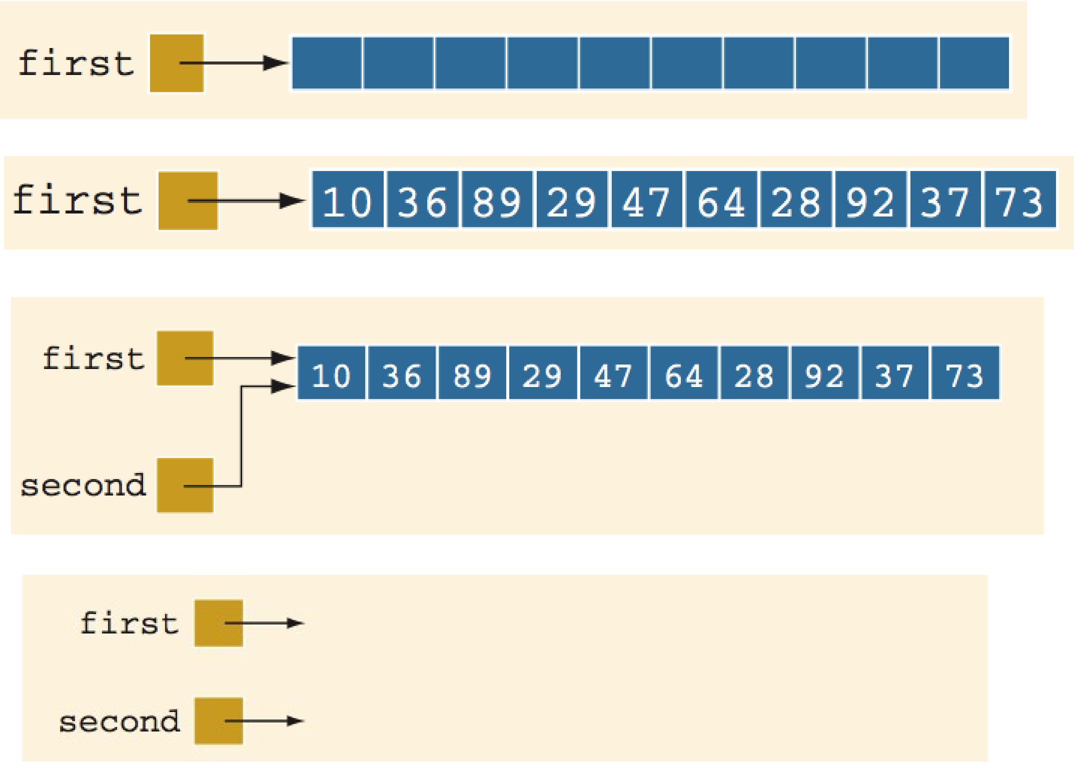
The function receives a pointer to a string

It resets the string to a certain value

In the main, we need to pass the address of the x variable to the function Reset()

Shallow versus deep copy

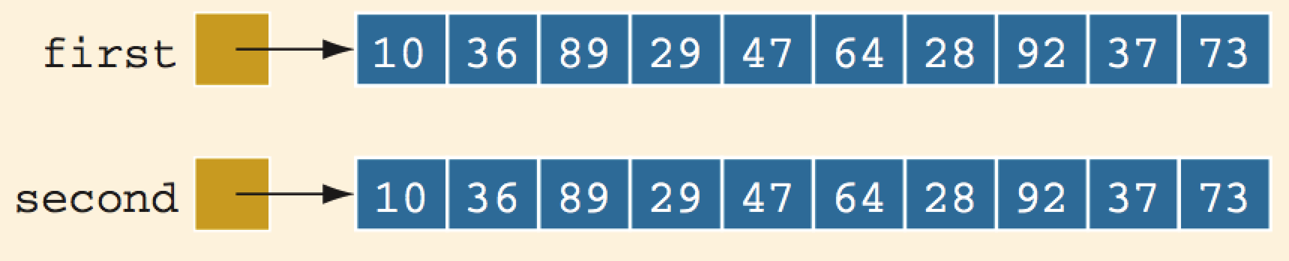
```
int *first;
int *second;

first = new int[10];

second = first;

delete [] second;
```



- After a sequence of this type, both pointers are dangling
  - If the program tries to access first, it will either crash or produce and invalid result

Shallow versus deep copy

```
second = new int[10];

for (int j = 0; j < 10; j++)
     second[j] = first[j];
```



first → 10 36 89 29 47 64 28 92 37 73

second → 10 36 89 29 47 64 28 92 37 73

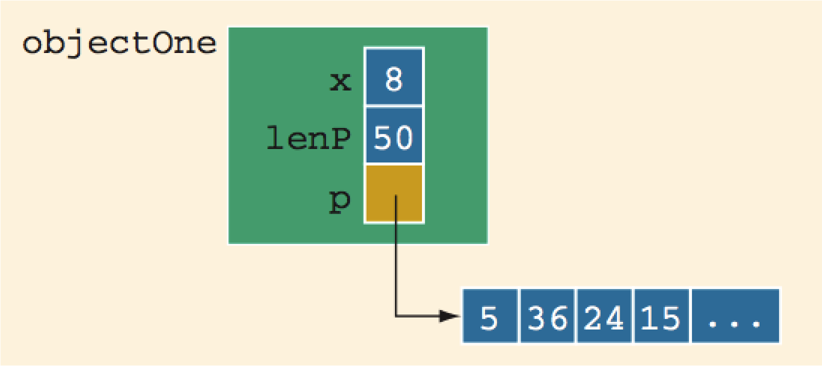
- Deleting the second pointer will not invalidate the first one

Destructor

- Consider the following example:

```
class ptrMemberVarType
{
public:
          .
          .
          .
private:
     int x;
     int lenP;
     int *p;
};
```

Object of type ptrMemberVarType



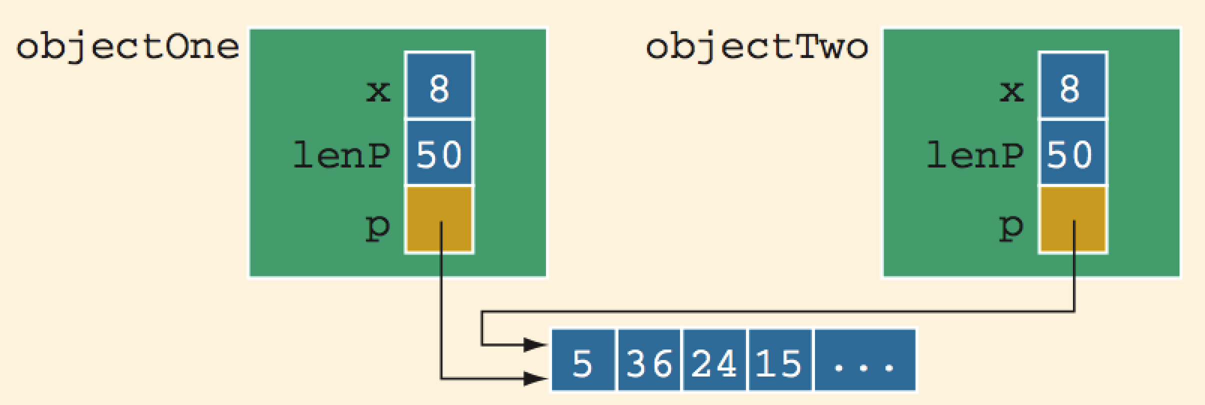
- When going out of scope, we need to free the memory allocated to p

```
ptrMemberVarType::~ptrMemberVarType()
{
     delete [] p;
}
```

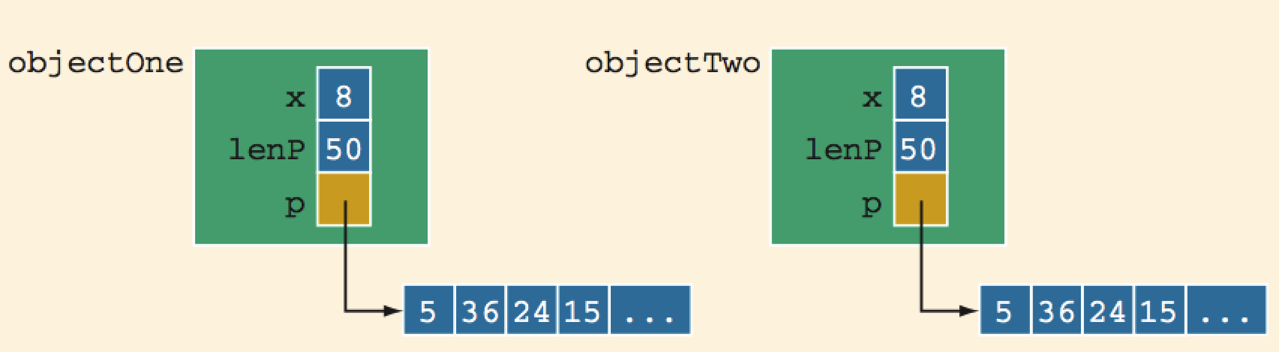
Notice: p should be properly initialized before destructing it!

Overloading the copy operator

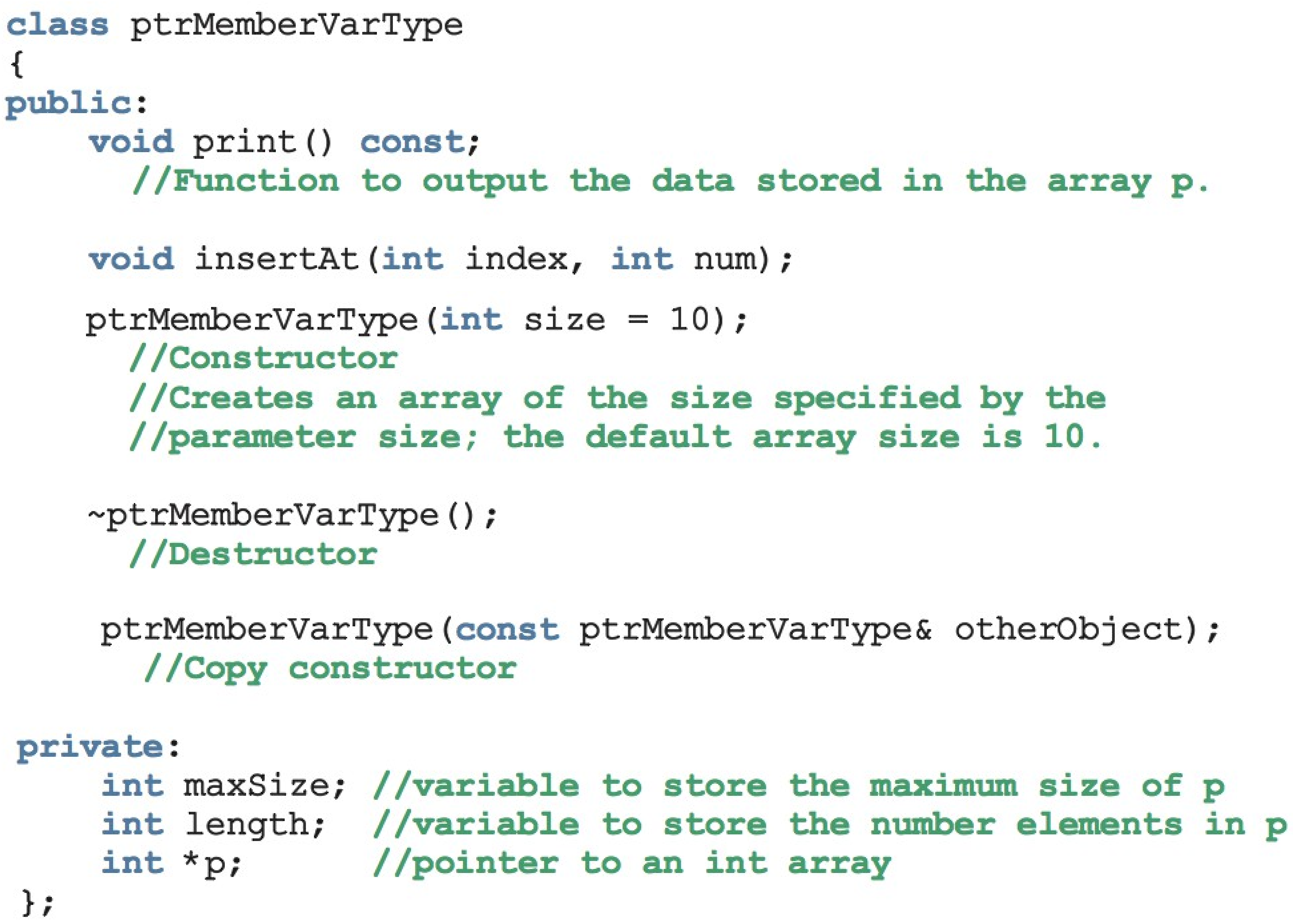
```
objectTwo = objectOne;
```



- If `objectOne` dealocates the memory of pointer `p`, `objectTwo` becomes invalid
- Overloading: C++ allows you to extend the copy operator

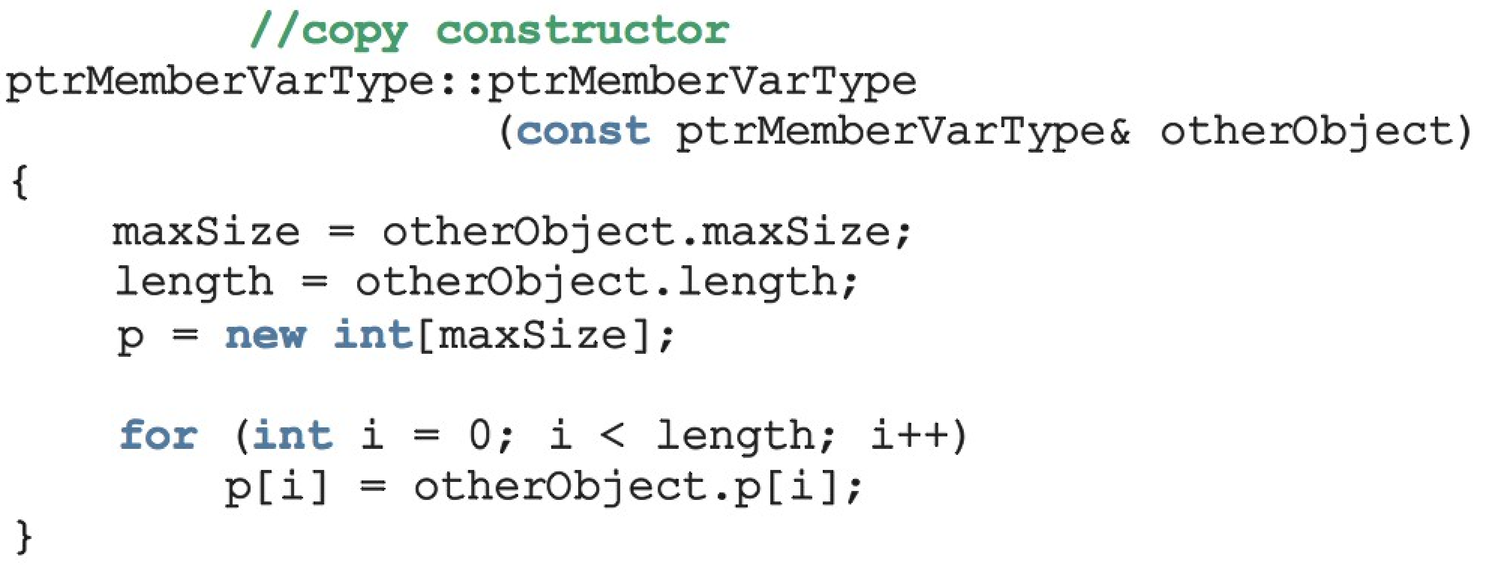


Copy constructor



```
class ptrMemberVarType
{
public:
    void print() const;
      //Function to output the data stored in the array p.

    void insertAt(int index, int num);

    ptrMemberVarType(int size = 10);
      //Constructor
      //Creates an array of the size specified by the
      //parameter size; the default array size is 10.

    ~ptrMemberVarType();
      //Destructor

     ptrMemberVarType(const ptrMemberVarType& otherObject);
       //Copy constructor

 private:
     int maxSize;  //variable to store the maximum size of p
     int length;   //variable to store the number elements in p
     int *p;       //pointer to an int array
 };
```

Copy constructor



```
           //copy constructor
ptrMemberVarType::ptrMemberVarType
                  (const ptrMemberVarType& otherObject)
{
    maxSize = otherObject.maxSize;
    length = otherObject.length;
    p = new int[maxSize];

    for (int i = 0; i < length; i++)
        p[i] = otherObject.p[i];
}
```

- Avoids shallow copy of the pointers

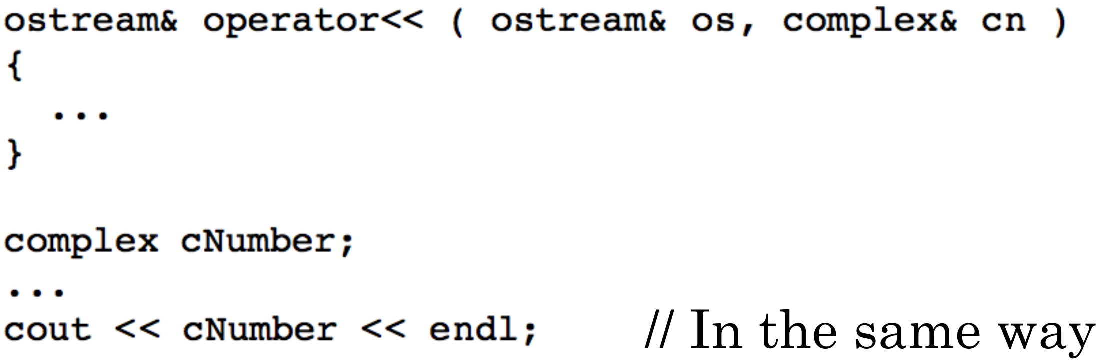
Reading/Writing files

Input/output

- I/O is the process of sending and receiving data
- I/O may be done to:
  - Persistent devices (such as file systems)
  - Volatile/ephemeral devices (screen, keyboard)
  - Persistent non-computer devices (printers)
- Programming languages provide interfaces to performing I/O and accessing persistent devices
  - C++ has the iostream library
- They also provide abstractions for doing so
  - Stream abstraction
  - File abstraction
  - C's stdio library

Streams

- Streams are made of basic types
  - Characters (bytes) in C++
- Every class for reading from input devices derives from: istream
- Every class for writing to output devices derives from: ostream
  - Functions that return ostream/istream references can write/read from any arbitrary device
  - Flexibility and reusability of interfaces



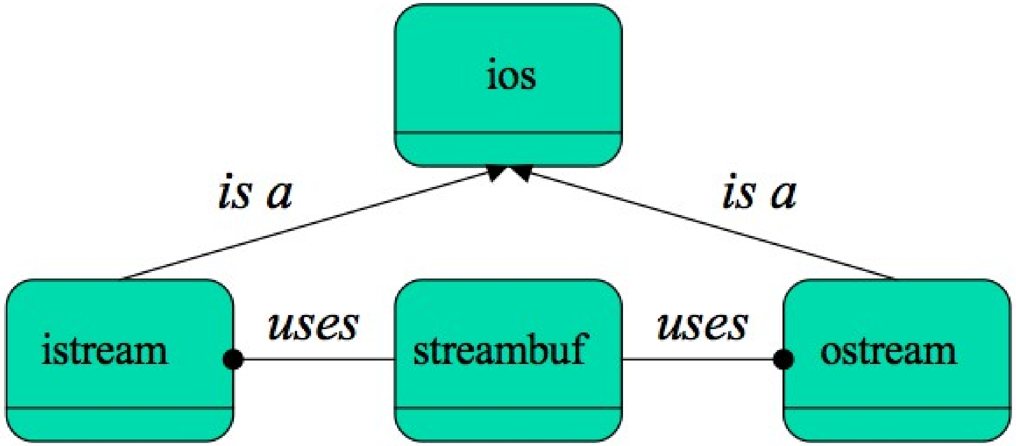
```
ostream& operator<< ( ostream& os, complex& cn )
{
   ...
}

complex cNumber;
...
cout << cNumber << endl;
```

// In the same way could send output to a file

iostream

- ios is a base class that
  - Manages error and format state of a stream
  - Communicates with a device's buffer
- streambuf is a helper class that
  - Buffers data
- istream and ostream are specializations of ios that define input and output specific

  operations
  - Example: <<and>>

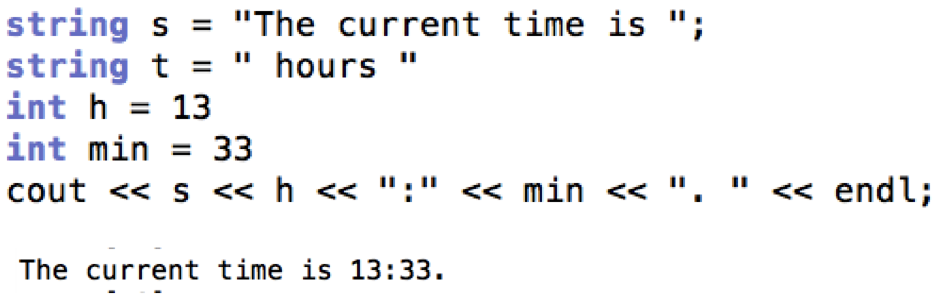


streambuf

- Associated to ostream/istream
- Memory block that acts as an intermediary between the stream and the physical file
  - Characters not flushed directly to file
  - Kept on buffer till data is written to the physical medium/freed
    - Synchronization
- Synchronization takes place when:
  - File is closed
  - The buffer is full
  - Explicitly, with manipulators (example: flush, endl).
  - Explicitly, with member function sync()

Formatting

- Formatting
  - Send the input into the stream abstraction
  - Convert arbitrary types to character streams
- Extended by class definitions of operator<< and operator>>
  - Which use the existing formatting for built in Types



```
string s = "The current time is ";
string t = " hours "
int h = 13
int min = 33
cout << s << h << ":" << min << ". " << endl;
```

```
 The current time is 13:33.
```

  - Easily extensible interface:

```
ostream & operator<< ( ostream& os, const complex & other )
{
  os << other.getReal() << “ + “ << other.getImag() << “i”;
  return os;
}
```

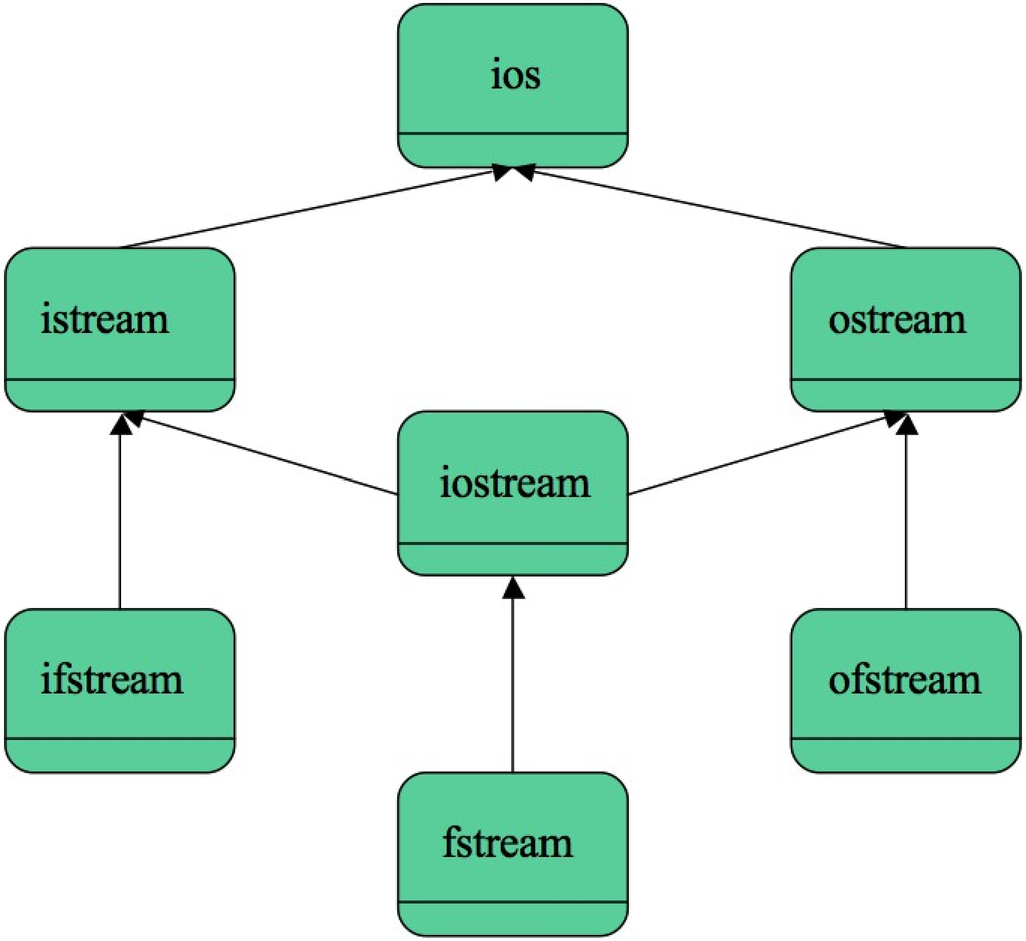
File streams

- Stream to read/write to a file
  - Data will be persistent
- File classes
  - Output class ofstream inherits from ostream
  - Input class ifstream inherits from istream
  - Input/output class fstream used to read/write to the same file
- Thus, standard stream interfaces can be used to read/write files
- Name of the file specified in the constructor

```
#include<fstream>
ifstream is ( “input.dat” );
ofstream os ( “output.dat” );
int n;
while ( is >> n )
{
  os << n << endl;
}
```

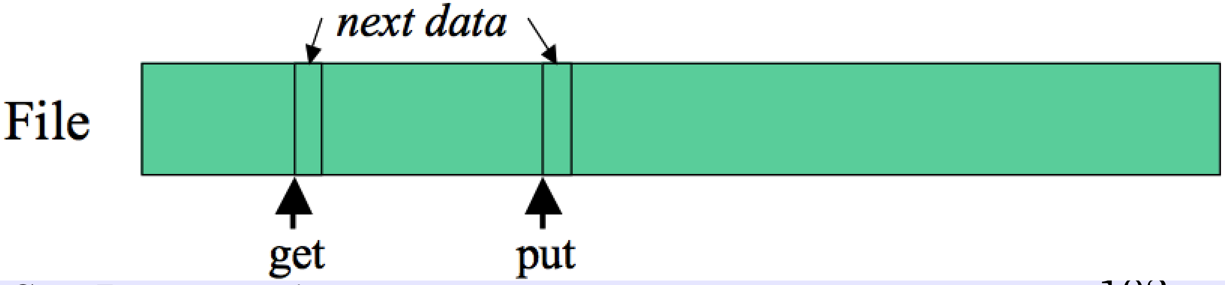
File streams

- File stream classes are a example of multiple inheritance



File abstraction

- A file is a stream
  - by definition as it inherits the properties
- A file contains persistent data
  - Write creates new data (or overwrites existing data)
  - Read returns existing data (without damaging the data)
  - Differs from other stream types which are destructive
- A file uses "pointers" to implement the stream abstraction
  - Get "pointer" for the next data to be read
  - Put "pointer" for the next data to be written
- Reading/writing advances the pointers



File attributes

- Properties of the file can be specified:
  - In the constructor
  - Using the open() function with a default constructor
- Properties dictate:
  - legal operations (read, write, append)
  - disposition of the file pointer (start, end)
  - naming/creation options
  - mode (binary or text)

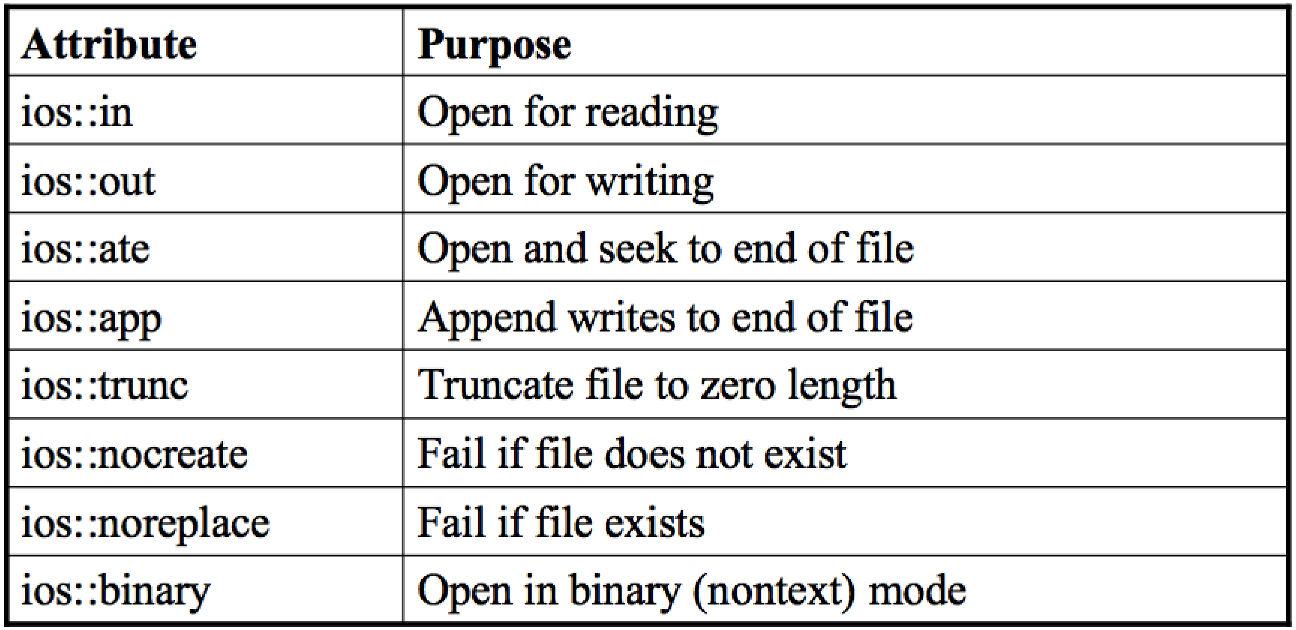
File attributes

- Properties of the file can be specified:

  In the constructor

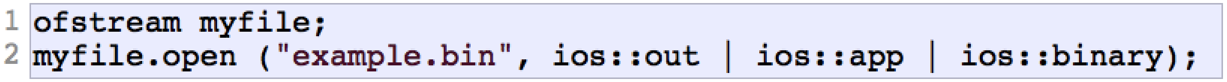
  Using the open() function with a default constructor

- Attributes:

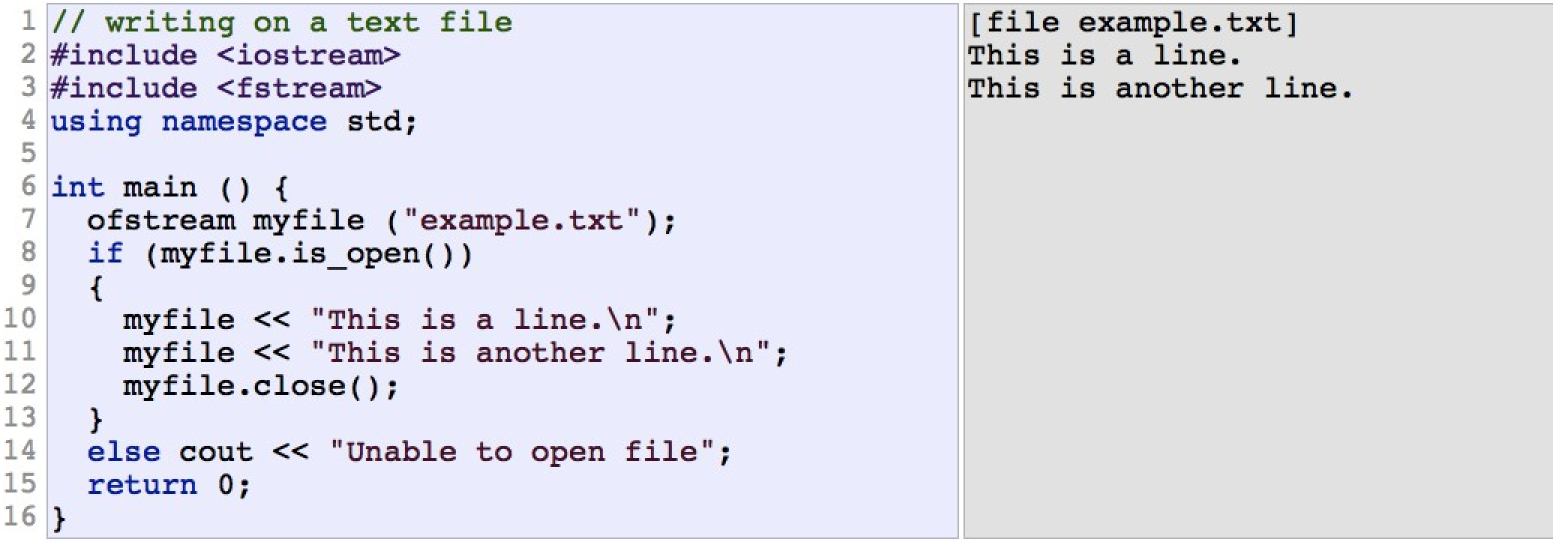


Attribute	Purpose
ios::in	Open for reading
ios::out	Open for writing
ios::ate	Open and seek to end of file
ios::app	Append writes to end of file
ios::trunc	Truncate file to zero length
ios::nocreate	Fail if file does not exist
ios::noreplace	Fail if file exists
ios::binary	Open in binary (nontext) mode

Examples



```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```



```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Backup