C/C++

A General-Purpose Programming Language

Engr. Qazi Ejaz Ur Rehman
Avionics Engineer

Graduate Teaching Assistant
Aeronautics & Astronautics Department
Institute of Space Technology
Islamabad

May 26, 2016

Institute of Space Technology

Outline

1. History
2. Labs
3. Functions
   - Inline Function
   - Overloads
   - Templates
   - Name visibility
   - Namespaces
4. Dynamic Memory
5. Structure
6. Classes
7. File I/O
8. Codes

Administration

Contact

- E-mail: qaziejazurrehman@gmail.com [1]
- Office hours: After 11:00 am

[1] qaziajazurrehman@gmail.com

Introduction

A Brief Introduction

A Chinese abacus

- The only mechanical device that existed for numerical computation at the beginning of human history was the abacus, invented in Sumeria circa 2500 BC
- And is still widely used by merchants, traders and clerks in Asia, Africa, and elsewhere

Introduction

Antikythera mechanism

The Antikythera mechanism (Fragment A – back)

The Antikythera mechanism (Fragment A – front)

- The Antikythera mechanism, some time around 100 BC in ancient Greece, is the first known analog computer (mechanical calculator)
- Designed to predict astronomical positions and eclipses for calendrical and astrological purposes as well as the Olympiads, the cycles of the ancient Olympic Games

Introduction

Badi′al – Zaman Abū al –′ Izz Ismā′īl ibnal – Razāzal – Jazarī

- ✍ The Kurdish medieval scientist Al-Jazari built programmable automata[2] in 1206 AD.
- Born: 1136 CE
- Era: Islamic GOlden Age
- Died: 1206 CE

[2] Same Idea as in Movie Automata (2014)

Introduction

Johann Bernoulli [3]

- 1667: Born in Switzerland, son of an apothecary (in medical profession)
- 1738: His son, Daniel Bernoulli published *Bernoulli's* principle
- Students include his son Daniel, EULER, *L'Hopital*
- *1748: Death*

[3] http://en.wikipedia.org/wiki/JohannBernoulli

Introduction

Leonhard Euler [4]

- 1707: Born in Switzerland, son of a pastor
- Among several other things, developed Euler's identity, $e^{j\omega} = cos(\omega) + jsin(\omega)$
- Also developed marvelous polyhedral fromula, nowadays written as "$v - e + f = 2$".
- Friend of his doctoral advisor's son, Daniel Bernoulli, who developed Bernoulli's principle
- 1783: Death

[4] http://en.wikipedia.org/wiki/LeonhardEuler

Introduction

Pierre-Simon Laplace [5]

- 1749: Born in France, son of a laborer
- 1770-death: Worked on probability, celestial mechanics, heat theory
- 1785: Examiner, examined and passed Napoleon in exam
- 1790: Paris Academy of Sciences, worked with Lavoisier, Coulomb
- 1827: Died

[5] http://en.wikipedia.org/wiki/Pierre-Simon Laplace

Introduction

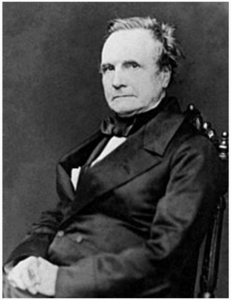
Joseph Fourier [6]

- 1768: Born in France, son of a tailor
- 1789-1799: Promoted the French Revolution
- 1798: Went with Napoleon to Egypt and made governor of Lower Egypt
- 1822: Showed that representing a function by a trigonometric series greatly simplifies the study of heat propagation
- 1830: Fell from stairs and died shortly afterward

[6] http://en.wikipedia.org/wiki/JosephFourier

Introduction

Charles Babbage



Charles Babbage in 1860

- Babbage is credited with inventing the first mechanical computer that eventually led to more complex designs.
- Born: 26 December 1791 London, England
- Considered by some to be a "father of the computer"
- Died: 18 October 1871 (aged 79) Marylebone, London, England

Introduction

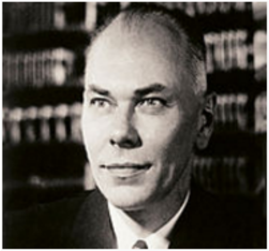
John Vincent Atanasoff (1903-1995)

Figure: Atanasoff, in the 1990s.

Built first digital computer in the 1930s.

Introduction

Howard Hathaway Aiken (1901-1980)



Howard Aiken

- Built Mark I, during 1939-1944
- Presented to public in 1944
- Reaction was great
    - Although Mark I meant a great deal for the development in computer science, it's not recognised greatly today.
    - The reason for this is the fact that Mark I (and also Mark II) was not electronic - it was electromagnetical

Introduction

J. Presper Eckert (1919-1995) and Mauchly (1907-1980)


Built ENIAC (Electronic Numerical Integrator and Computer), the first electronic general-purpose computer during 1943-1945 at a cost of $468,000.

Introduction

Alan Mathison Turing[7]

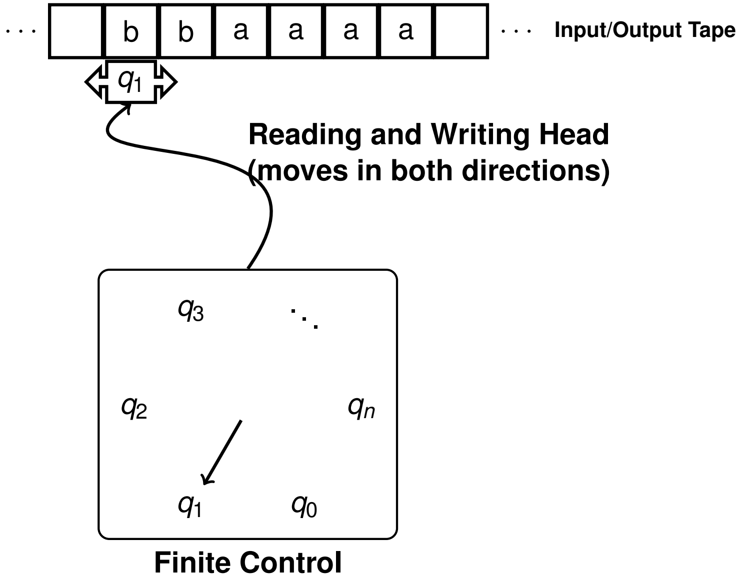
Turing aged 16

- **Born**: 23 June 1912
- Turing is widely considered to be the father of theoretical computer science and artificial intelligence
- Famous for Breaking Enigma Machine Code
- **Died**: 7 June 1954 (aged 41)

[7] The Imitation Game: A 2014 Movie biographied on turing

Turing Machine

Institute of Space Technology



History

FORTRAN

- Inventor: John Backus
- FORTRAN, derived from Formula Translating System
- It is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing. Originally developed by IBM
- First Appeared: 1957; 59 years ago

History

C++

Bjarne Stroustrup

Bjarne Stroustrup

- Inventor: Bjarne Stroustrup (at Bell Labs)
- It is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation
- C++ is standardized by the International Organization for Standardization (ISO)
- First Appeared: 1983; 33 years ago

Excellence of Human

Equations: Changed The World

17 Equations That Changed The World

Pythagora.s Theorem	$a^2 + b^2 = c^2$	Pythagoras,530 BC
Logarithms	$logxy = logx + logy$	John Napier, 1610
Calculus	$\frac{df}{dt} = \lim_{h\to 0} \frac{f(t+h)-f(t)}{h}$	Newton, 1668
Law of Gravity	$F = G\frac{m_1 m_2}{r^2}$	Newton, 1687
Complex Identity	$i^2 = -1$	Euler, 1750
Polyhedra Formula	$V - E + F = 2$	Euler, 1751
Normal Distribution	$\phi(x) = \frac{1}{\sqrt{2\pi\rho}} e^{\frac{(x-\mu)^2}{2\rho^2}}$	C.F. Gauss, 1810
Wave Equation	$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$	J. d'Almbert,1746
Fourier Transform	$f(\omega) = \int_{-\infty}^{\infty} f(x)e^{-2\pi i x\omega}dx$	J. Fourier, 1822
Navier-Stokes Equation	$\rho(\frac{\partial v}{\partial t} + v.\nabla v) = -\nabla p + \nabla.T + f$	C. Navier, G. Stokes, 1845
Maxwell's Equations	$\nabla E = \frac{\rho}{\epsilon 0}$ $\quad \nabla.H = 0$ $\quad \nabla \times E = -\frac{1}{c}\frac{\partial H}{\partial t}$ $\quad \nabla \times H = \frac{1}{c}\frac{\partial E}{\partial t}$	J.C. Maxwell, 1865
Second Law of Thermosynamics	$dS \geq 0$	L. Boltzmann, 1874
Relativity	$E = mc^2$	Einstein, 1905
Schrodinger's Equation	$i\hbar\frac{\partial}{\partial t}\blacksquare = H\blacksquare$	E. Schrodinger, 1927
Information Theory	$H = -\sum p(x)logp(x)$	C. Shannon, 1949
Chaos Theory	$x_{t+1} = kx_t(1 - x_t)$	Robert May,1975
Black-Scholes Equation	$\frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} + \frac{\partial V}{\partial t} - rV = 0$	F. Black, M. Scholes, 1990

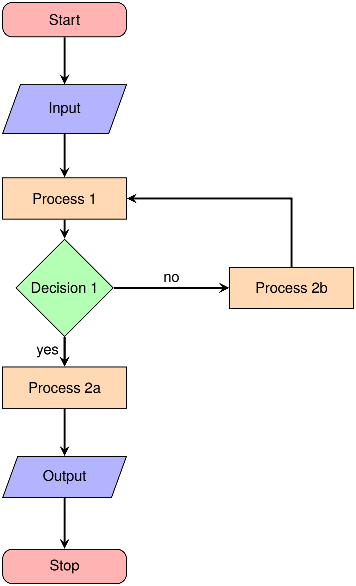
Introduction

Programming Accessories

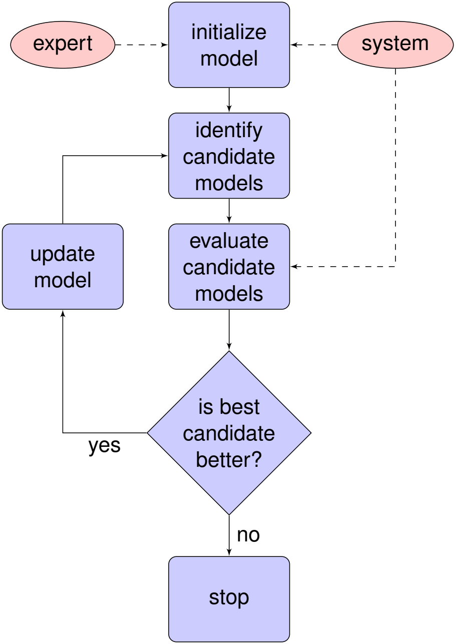
Whatever the approach to development may be, the final program must satisfy some fundamental properties. The following properties are among the most important

- Reliability
- Robustness
- Usability
- Portability
- Maintainability
- Efficiency/performance

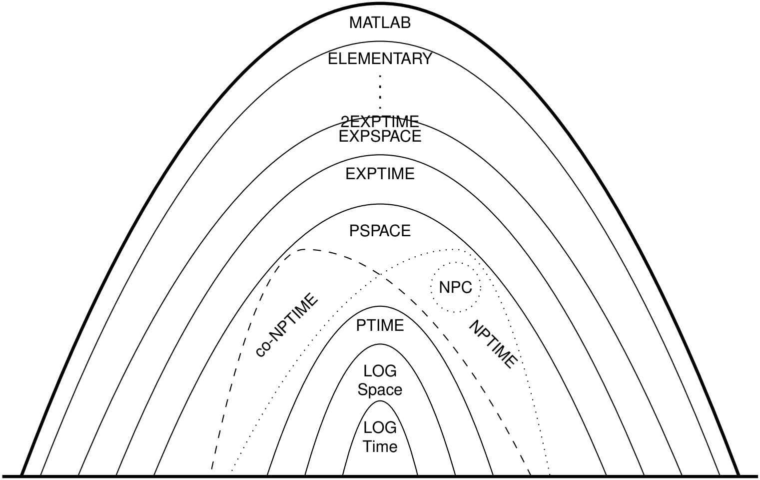
Flow Chart


Start
Input
Process 1
Decision 1
no
Process 2b
yes
Process 2a
Output
Stop

Flow Chart


expert
initialize model
system
identify candidate models
update model
evaluate candidate models
is best candidate better?
yes
no
stop

Computational Complexity Classes



Lab1

Learn to Record and Share Your Results Electronically

- Learn how to make a website and put your results on it
  - Website files must not be path dependent, i.e, if I copy them to any location such as a USB, or different directory, the website must still work
  - The main file of the website must be index.html
- Many tools are available, but a good cross-platform open source software is kompozer available from `http://www.kompozer.net/`

Lab1

Learn to Extend Existing Work in a Controls Topic

Make groups, pick a research topic, create a website with following headings:

1. Introduction
2. Technical Background
3. Expected Experiments
4. Expected Results
5. Expected Conclusions

Present your website. Every group member will be quizzed randomly. Your final work will count towards your lab exam.

C++

Functions

- Functions allow to structure programs in segments of code to perform individual tasks.
- A function is a group of statements that is given a name, and which can be called from some point of the program
- type name (parameter1, parameter2, ...) { statements }
  Where:
  - type is the type of the value returned by the function.
  - name is the identifier by which the function can be called.
  - parameters each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.
  - statements is the function's body. And specify what the function actually does.

Functions

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

Functions

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

C++

Functions

Arguments passed by value and by reference

- In the functions seen earlier, arguments have always been passed by value.
- This means what is passed to the function are the values of these arguments on the moment of the call
  - And are copied into the variables represented by the function parameters
- In certain cases, though, it may be useful to access an external variable from within a function.

```
void duplicate (int& a,int& b,int& c)

                    x          y          z

      duplicate (   x   ,    y   ,    z   );
```

Functions

Function with no type

```
// passing parameters by reference
#include <iostream>
using namespace std;

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z="
       << z;
  return 0;
}
```

- Call by value
  * Values passed as aurgument are copied
    - This is a relatively inexpensive operation for fundamental types such as *int*
    - An overhead occur if data is compound type

```
string concatenate (string a, string b)
{
  return a+b;
}
```

C++
Functions
Efficiency considerations and const references

- Call by reference
  * Overheading can be avoided
    - The function operates directly on the strings passed as arguments
    - The functions with reference parameters are generally perceived as functions that modify the arguments passed

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Defining arguments as constant can be used to prevent modifying the referenced arguments for function

```
string concatenate (const string& a, const string&
     b)
{
  return a+b;
}
```

Note: using const will call this function by value in more efficient way

C++
Functions
Efficiency considerations and const references

- Call by reference
  - * Overheading can be avoided
    - The function operates directly on the strings passed as arguments
    - The functions with reference parameters are generally perceived as functions that modify the arguments passed

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Defining arguments as constant can be used to prevent modifying the referenced arguments for function

```
string concatenate (const string& a, const string&
     b)
{
  return a+b;
}
```

Note: using const will call this function by value in more efficient way

- Call by reference
  * Overheading can be avoided
    - The function operates directly on the strings passed as arguments
    - The functions with reference parameters are generally perceived as functions that modify the arguments passed

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Defining arguments as constant can be used to prevent modifying the referenced arguments for function

```
string concatenate (const string& a, const string&
     b)
{
  return a+b;
}
```

Note: using const will call this function by value in more efficient way

C++

Inline Function

- An optimized technique used by compiler to reduce the execution time
- Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime
- For big functions (in term of executable instruction), Compiler treat them as normal function

C++

Inline Function

- An optimized technique used by compiler to reduce the execution time
- Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime
- For big functions (in term of executable instruction), Compiler treat them as normal function

C++

Inline Function

- An optimized technique used by compiler to reduce the execution time
- Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime
- For big functions (in term of executable instruction), Compiler treat them as normal function

C++

Inline Function

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

inline int sqr(int x)
{
    int y;
    y = x * x;
    return y;
}
int main()
{
    int a =3, b;
    b = sqr(a);
    cout <<b;
    return 0;
}
```

C++

Inline Function

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// Main function for the program
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) <<
       endl;
   return 0;
}
```

C++

Overloaded Function

- In C++, two different functions can have the same name
- But there arguments type should be different
- Compiler determines by the type of argument being passed

C++

Overloaded Function

- In C++, two different functions can have the same name
- But there arguments type should be different
- Compiler determines by the type of argument being passed

C++

Overloaded Function

- In C++, two different functions can have the same name
- But there arguments type should be different
- Compiler determines by the type of argument being passed

C++

Overloaded Function

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

C++

Overloaded Function

```
// overloaded functions
#include <iostream>
using namespace std;

int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

C++

Templates

- The function could be overloaded for a lot of types (e.g previous slide)
- And it could make sense for all of them to have the same body
- C++ has the ability to define functions with generic types, known as function templates

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

C++
Templates

- The function could be overloaded for a lot of types (e.g previous slide)
- And it could make sense for all of them to have the same body
- C++ has the ability to define functions with generic types, known as function templates

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

C++

Templates

- The function could be overloaded for a lot of types (e.g previous slide)
- And it could make sense for all of them to have the same body
- C++ has the ability to define functions with generic types, known as function templates

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

C++

Templates

- The function could be overloaded for a lot of types (e.g previous slide)
- And it could make sense for all of them to have the same body
- C++ has the ability to define functions with generic types, known as function templates

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

C++
Templates

- The function could be overloaded for a lot of types (e.g previous slide)
- And it could make sense for all of them to have the same body
- C++ has the ability to define functions with generic types, known as function templates

```cpp
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

C++

Template Function

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

C++

Template Function

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
return 0;
}
```

C++

Non-type template arguments

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

C++

Scope (Local & global) of variables

- The point in the program where variable declaration happens influences its visibility:
- An entity declared outside any block has global scope
- While an entity declared within a block (function or a selective statement), has block/local scope

```
int foo;          // global variable
int some_function ()
{
  int bar;        // local variable
  bar = 0;
}
int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from
      this function
}
```

C++

Scope (Local & global) of variables

- The point in the program where variable declaration happens influences its visibility:
- An entity declared outside any block has global scope
- While an entity declared within a block (function or a selective statement), has block/local scope

```
int foo;          // global variable
int some_function ()
{
  int bar;        // local variable
  bar = 0;
}
int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from
      this function
}
```

C++

Scope (Local & global) of variables

- The point in the program where variable declaration happens influences its visibility:
- An entity declared outside any block has global scope
- While an entity declared within a block (function or a selective statement), has block/local scope

```
int foo;          // global variable
int some_function ()
{
  int bar;        // local variable
  bar = 0;
}
int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from
      this function
}
```

C++

Scope (Local & global) of variables

- The point in the program where variable declaration happens influences its visibility:
- An entity declared outside any block has global scope
- While an entity declared within a block (function or a selective statement), has block/local scope

```
int foo;          // global variable
int some_function ()
{
  int bar;        // local variable
  bar = 0;
}
int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from
      this function
}
```

C++

Scope (Local & global) of variables

- The point in the program where variable declaration happens influences its visibility:
- An entity declared outside any block has global scope
- While an entity declared within a block (function or a selective statement), has block/local scope

```
int foo;          // global variable
int some_function ()
{
  int bar;        // local variable
  bar = 0;
}
int other_function ()
{
  foo = 1;  // ok: foo is a global variable
  bar = 2;  // wrong: bar is not visible from
      this function
}
```

C++

Namespaces

- Non-local names bring more possibilities for name collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes,
- This allows organizing the elements of programs into different logical scopes referred to by names

```
namespace identifier
{
  named_entities
}
```

C++

Namespaces

- Non-local names bring more possibilities for name collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes,
- This allows organizing the elements of programs into different logical scopes referred to by names

```
namespace identifier
{
  named_entities
}
```

C++

Namespaces

- Non-local names bring more possibilities for name collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes,
- This allows organizing the elements of programs into different logical scopes referred to by names

```
namespace identifier
{
  named_entities
}
```

C++
Namespaces

- Non-local names bring more possibilities for name collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes,
- This allows organizing the elements of programs into different logical scopes referred to by names

```
namespace identifier
{
  named_entities
}
```

C++

Namespaces

- Non-local names bring more possibilities for name collision
- Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes,
- This allows organizing the elements of programs into different logical scopes referred to by names

```
namespace identifier
{
  named_entities
}
```

C++

Namespaces

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

namespace foo
{
  int value() { return 5; }
}

namespace bar
{
  const double pi = 3.1416;
  double value() { return 2*pi; }
}

int main () {
  cout << foo::value() << '\n';
  cout << bar::value() << '\n';
  cout << bar::pi << '\n';
  return 0;
}
```

C++

using function

The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name.

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

namespace first
{  int x = 5;
   int y = 10; }
namespace second
{  double x = 3.1416;
   double y = 2.7183;}

int main () {
   using first::x;
   using second::y;
   cout << x << '\n';
   cout << y << '\n';
   cout << first::y << '\n';
   cout << second::x << '\n';
   return 0;
}
```

C++

using function

The keyword using introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name.

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

namespace first
{  int x = 5;
  int y = 10; }
namespace second
{  double x = 3.1416;
  double y = 2.7183;}

int main () {
  using first::x;
  using second::y;
  cout << x << '\n';
  cout << y << '\n';
  cout << first::y << '\n';
  cout << second::x << '\n';
  return 0;
}
```

C++

using & namespace function

The keyword usingcan also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

C++

using & namespace function

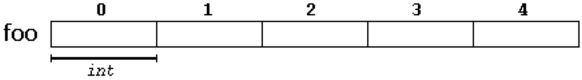
The keyword usingcan also be used as a directive to introduce an entire namespace:

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

namespace first
{
  int x = 5;
  int y = 10;
}

namespace second
{
  double x = 3.1416;
  double y = 2.7183;
}

int main () {
  using namespace first;
  cout << x << '\n';
  cout << y << '\n';
  cout << second::x << '\n';
  cout << second::y << '\n';
  return 0;
}
```

C++

Array

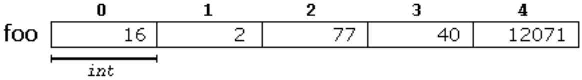
- An array is a series of elements of the same type placed in contiguous memory locations.
- An array containing 5 integer values of type int called foo could be represented as:



- An array must be declared before it is used. A typical declaration for an array in C++ is:

type name[*elements*];

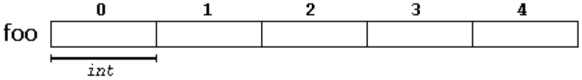
int foo[5] = {16, 2, 77, 40, 12071};



C++

Array

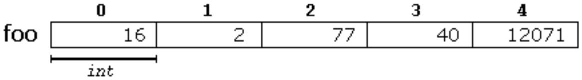
- An array is a series of elements of the same type placed in contiguous memory locations.
- An array containing 5 integer values of type int called foo could be represented as:



- An array must be declared before it is used. A typical declaration for an array in C++ is:

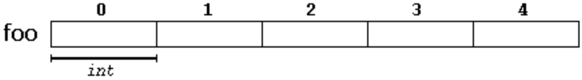
type name[*elements*];

int foo[5] = {16, 2, 77, 40, 12071};



C++
Array

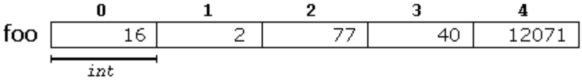
- An array is a series of elements of the same type placed in contiguous memory locations.
- An array containing 5 integer values of type int called foo could be represented as:



- An array must be declared before it is used. A typical declaration for an array in C++ is:

$$\textit{type name}[\textit{elements}];$$

$$\textit{int foo}[5] \quad = \{16,\ 2,\ 77,\ 40,\ 12071\};$$



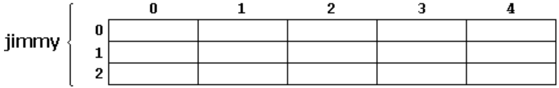
C++
Array

```
// arrays example
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

C++

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays"
- For example, a bidimensional array can be imagined as a two-dimensional table with elements of same uniform data type

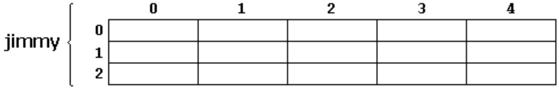


- The C++ syntax for this is:
  `int jimmy [3][5];`
- Multidimensional arrays are not limited to two indices
  `char century [100][365][24][60][60];`
- Multidimensional arrays are just an abstraction for programmers
  `int jimmy [3][5];` // is equivalent to
  `int jimmy [15];` // (3 * 5 = 15)

C++

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays"
- For example, a bidimensional array can be imagined as a two-dimensional table with elements of same uniform data type

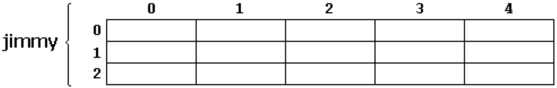


- The C++ syntax for this is:
  int jimmy [3][5];
- Multidimensional arrays are not limited to two indices
  char century [100][365][24][60][60];
- Multidimensional arrays are just an abstraction for programmers
  int jimmy [3][5]; // is equivalent to
  int jimmy [15]; // (3 * 5 = 15)

C++

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays"
- For example, a bidimensional array can be imagined as a two-dimensional table with elements of same uniform data type

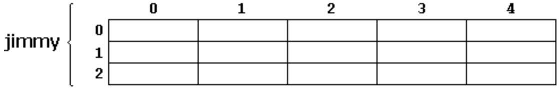


- The C++ syntax for this is:
  `int jimmy [3][5];`
- Multidimensional arrays are not limited to two indices
  `char century [100][365][24][60][60];`
- Multidimensional arrays are just an abstraction for programmers
  `int jimmy [3][5]; // is equivalent to`
  `int jimmy [15]; // (3 * 5 = 15)`

C++

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays"
- For example, a bidimensional array can be imagined as a two-dimensional table with elements of same uniform data type



- The C++ syntax for this is:
  `int jimmy [3][5];`
- Multidimensional arrays are not limited to two indices
  `char century [100][365][24][60][60];`
- Multidimensional arrays are just an abstraction for programmers
  `int jimmy [3][5]; // is equivalent to`
  `int jimmy [15];  // (3 * 5 = 15)`

C++

Multidimensional arrays

- Multidimensional arrays can be described as "arrays of arrays"
- For example, a bidimensional array can be imagined as a two-dimensional table with elements of same uniform data type

jimmy

	0	1	2	3	4
0					
1					
2					

- The C++ syntax for this is:
  `int jimmy [3][5];`
- Multidimensional arrays are not limited to two indices
  `char century [100][365][24][60][60];`
- Multidimensional arrays are just an abstraction for programmers
  `int jimmy [3][5];` // is equivalent to
  `int jimmy [15];` // (3 * 5 = 15)

C++

Multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

- Above (both) codes do not produce any output
- But Assign values to the memory block called jimmy in the following way

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

C++

Multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

- Above (both) codes do not produce any output
- But Assign values to the memory block called jimmy in the following way

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

C++

Multidimensional array

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
}
```

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
  for (n=0; n<HEIGHT; n++)
    for (m=0; m<WIDTH; m++)
    {
      jimmy[n*WIDTH+m]=(n+1)*(m+1);
    }
}
```

- Above (both) codes do not produce any output
- But Assign values to the memory block called jimmy in the following way

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

C++

Arrays as parameters

- To pass an array to a function as a parameter
- In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument
- But what can be passed instead is its address. As it is efficient way
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Arrays as parameters

- To pass an array to a function as a parameter
- In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument
- But what can be passed instead is its address. As it is efficient way
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Arrays as parameters

- To pass an array to a function as a parameter
- In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument
- But what can be passed instead is its address. As it is efficient way
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Arrays as parameters

- To pass an array to a function as a parameter
- In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument
- But what can be passed instead is its address. As it is efficient way
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Arrays as parameters

- To pass an array to a function as a parameter
- In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument
- But what can be passed instead is its address. As it is efficient way
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Arrays as parameters

```
// arrays as parameters
#include <iostream>
using namespace std;

void printarray (int arg[], int length) {
  for (int n=0; n<length; ++n)
    cout << arg[n] << ' ';
  cout << '\n';
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
}
```

C++

Library arrays

- The arrays explained above are directly implemented as a language feature
- Restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container (a type template (a class template, in fact))
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Library arrays

- The arrays explained above are directly implemented as a language feature
- Restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container (a type template (a class template, in fact))
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Library arrays

- The arrays explained above are directly implemented as a language feature
- Restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container (a type template (a class template, in fact))
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Library arrays

- The arrays explained above are directly implemented as a language feature
- Restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container (a type template (a class template, in fact))
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

Library arrays

- The arrays explained above are directly implemented as a language feature
- Restricting its copy and easily decay into pointers, they probably suffer from an excess of optimization
- C++ provides an alternative array type as a standard container (a type template (a class template, in fact))
- To accept an array as parameter for a function
- The parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array e.g.

```
void procedure (int arg[])
int myarray [40];
procedure (myarray);
```

C++

language built-in array Vs container library array

```
#include <iostream>

using namespace std;

int main()
{
  int myarray[3] = {10,20,30};

  for (int i=0; i<3; ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

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

int main()
{
  array<int,3> myarray {10,20,30};

  for (int i=0; i<myarray.size(); ++i)
    ++myarray[i];

  for (int elem : myarray)
    cout << elem << '\n';
}
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

```
pointer = new type
pointer = new type [number_of_elements]
```

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

  pointer = new type

  pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

pointer = new type

pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

  pointer = new type

  pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

pointer = new type

pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

  pointer = new type

  pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- In previous programs all memory needs were determined before program execution by defining the variables
- But when the memory needed depends on user input.
- On these cases, programs need to dynamically allocate memory
  - Dynamic memory is allocated using operator new
  - new is followed by a data type specifier
- If a sequence of more than one element is required, the number of these within brackets [].
- It returns a pointer to the beginning of the new block of memory allocated.

  pointer = new type

  pointer = new type [number_of_elements]

```
int * foo;
foo = new int [5];
```

C++

Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  * One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  * The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++

Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  * One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  * The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++

Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  * One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  * The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++

Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  * One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  * The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++
Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  - One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  - The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++

Dynamic Memory

- C++ provides two standard mechanisms to check if the allocation was successful:
  - * One is by handling exceptions i.e.(exception of type bad_alloc is thrown when the allocation fails.)
    - This exception method is the method used by default by new

      foo = new int [5];
  - * The other method is known as nothrow (the pointer returned by new is a null pointer)
    - On failure, memory is not thrown over bad_alloc

      foo = new (nothrow) int [5];
- Failure is detected by null pointer

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

C++

Erasing Pointers

- Memory allocated dynamically is only needed during specific periods of time within a program
- To free memory allocated dynamically *operator* delete is used

```
delete pointer;
delete[] pointer;
```

Note :

C++

Erasing Pointers

- Memory allocated dynamically is only needed during specific periods of time within a program
- To free memory allocated dynamically *operator* delete is used

```
delete pointer;
delete[] pointer;
```

Note :

C++
Erasing Pointers

- Memory allocated dynamically is only needed during specific periods of time within a program
- To free memory allocated dynamically *operator* delete is used

```
delete pointer;
delete[] pointer;
```

Note : The first statement releases the memory of a single element allocated using new

C++

Erasing Pointers

- Memory allocated dynamically is only needed during specific periods of time within a program
- To free memory allocated dynamically *operator* delete is used

```
delete pointer;
delete[] pointer;
```

Note : The second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

C++

Dynamic Memory

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```

C++

Structure

- A data structure is a group of data elements grouped together under one name
- These data elements, known as members, can have different types and different lengths
- syntax is

```
struct type_name
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
object_names;
```

C++
Structure

- A data structure is a group of data elements grouped together under one name
- These data elements, known as members, can have different types and different lengths
- syntax is

```
struct type_name
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
object_names;
```

C++

Structure

- A data structure is a group of data elements grouped together under one name
- These data elements, known as members, can have different types and different lengths
- syntax is

```
struct type_name
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
object_names;
```

C++

Structure

- A data structure is a group of data elements grouped together under one name
- These data elements, known as members, can have different types and different lengths
- syntax is

```
struct type_name
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
object_names;
```

```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

Note:

It is important to clearly differentiate between what is the **structure type** name (product),

C++

Structure

```
struct product {
  int weight;
  double price;
} ;

product apple;
product banana, melon;
```

```
struct product {
  int weight;
  double price;
} apple, banana, melon;
```

Note:

It is important to clearly differentiate between what is an object of this **type (apple, banana, and melon)**

C++

Important Characters

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed to by a	(*a).b
*a.b	Value pointed to by member b of object a	*(a.b)

C++

Nested Structure

```
struct movies_t {
  string title;
  int year;
};

struct friends_t {
  string name;
  string email;
  movies_t favorite_movie;
} charlie, maria;

friends_t * pfriends = &charlie
```

```
charlie.name
maria.favorite_movie.title
charlie.favorite_movie.year
pfriends->favorite_movie.year
```

C++

Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

  ```
  class class_name
  access_specifier_1:
  member1;
  access_specifier_2:
  member2;
  ...
  object_names;
  ```

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++

Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

  ```
  class class_name
  access_specifier_1:
  member1;
  access_specifier_2:
  member2;
  ...
  object_names;
  ```

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++

Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

```
class class_name
access_specifier_1:
member1;
access_specifier_2:
member2;
...
object_names;
```

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++

Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

```
class class_name
access_specifier_1:
member1;
access_specifier_2:
member2;
...
object_names;
```

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++

Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

  class class_name
  access_specifier_1:
  member1;
  access_specifier_2:
  member2;
  ...
  object_names;

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++
Classes

- Classes are an expanded concept of data structures:
- They can contain data members, but they can also contain functions as members.
- Classes are defined using either keyword class or keyword struct, syntax is:

```
class class_name
access_specifier_1:
member1;
access_specifier_2:
member2;
...
object_names;
```

- Classes have new things called access specifiers. And is one of the following three keywords:private, public or protected
- Classes can be defined not only with keyword class, but also with keywords struct and union.

C++

Access Specifiers

- private members of a class are accessible only from within other members of the same class (or from their "friends").
- protected members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.
- By default, all members of a class declared with the class keyword have private access for all its members

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

C++

Access Specifiers

- private members of a class are accessible only from within other members of the same class (or from their "friends").
- protected members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.
- By default, all members of a class declared with the class keyword have private access for all its members

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

C++

Access Specifiers

- private members of a class are accessible only from within other members of the same class (or from their "friends").
- protected members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.
- By default, all members of a class declared with the class keyword have private access for all its members

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

C++

Access Specifiers

- private members of a class are accessible only from within other members of the same class (or from their "friends").
- protected members are accessible from other members of the same class (or from their "friends"), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.
- By default, all members of a class declared with the class keyword have private access for all its members

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

Classes

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;}
```

Classes

Multiple Type

- The most important property of a class is that it is a type
- we can declare multiple objects of it.
- For example, following with the previous example of class Rectangle, we could have declared the object rectb in addition to object rect:

Classes

Multiple Type

```
// example: one class, two objects
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Classes

Constructors

- What would happen in the previous example if we called the member function area before having called set_values?
- An undetermined result, since the members width and height had never been assigned a value.
- In order to avoid that, a class can include a special function called its constructors
- The Rectangle class above can easily be improved by implementing a constructor

Classes

Constructors

- What would happen in the previous example if we called the member function area before having called set_values?
- An undetermined result, since the members width and height had never been assigned a value.
- In order to avoid that, a class can include a special function called its constructors
- The Rectangle class above can easily be improved by implementing a constructor

Classes

Constructors

- What would happen in the previous example if we called the member function area before having called set_values?
- An undetermined result, since the members width and height had never been assigned a value.
- In order to avoid that, a class can include a special function called its constructors
- The Rectangle class above can easily be improved by implementing a constructor

Classes

Constructors

- What would happen in the previous example if we called the member function area before having called set_values?
- An undetermined result, since the members width and height had never been assigned a value.
- In order to avoid that, a class can include a special function called its constructors
- The Rectangle class above can easily be improved by implementing a constructor

Classes

Constructors

```
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Classes

Overloading Constructors

- A constructor can also be overloaded with different versions taking different parameters
- The compiler will automatically call the one whose parameters match the arguments

Classes

Overloading Constructors

- A constructor can also be overloaded with a different number of parameters and/or parameters of different types
- The compiler will automatically call the one whose parameters match the arguments

Classes

Overloading Constructors

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Classes

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*
- class_name object_name = initialization_value;
- class_name object_name { value, value, value, ... }

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum()
       << '\n';
  return 0;
}
```

Classes

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*
- class_name object_name = initialization_value;
- class_name object_name { value, value, value, ... }

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum()
       << '\n';
  return 0;
}
```

Classes

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*
- class_name object_name = initialization_value;
- class_name object_name { value, value, value, ... }

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum()
       << '\n';
  return 0;
}
```

Classes

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*
- class_name object_name = initialization_value;
- class_name object_name { value, value, value, ... }

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum()
      << '\n';
  return 0;
}
```

Classes

Uniform Initialization

- The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*
- class_name object_name = initialization_value;
- class_name object_name { value, value, value, ... }

```
// classes and uniform initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) { radius = r; }
    double circum() {return 2*radius*3.14159265;}
};

int main () {
  Circle foo (10.0);   // functional form
  Circle bar = 20.0;   // assignment init.
  Circle baz {30.0};   // uniform init.
  Circle qux = {40.0}; // POD-like

  cout << "foo's circumference: " << foo.circum()
       << '\n';
  return 0;
}
```

Classes

Member initialization in constructors

```
// member initialization
#include <iostream>
using namespace std;

class Circle {
    double radius;
  public:
    Circle(double r) : radius(r) { }
    double area() {return radius*radius*3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
  public:
    Cylinder(double r, double h) : base (r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo (10,20);

  cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

File Input/Output

Institute of Space Technology

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream**: Stream class to write on files
  - **ifstream**: Stream class to read from files
  - **fstream**: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Input/Output

Institute of Space Technology

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream**: Stream class to write on files
  - **ifstream**: Stream class to read from files
  - **fstream**: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Input/Output

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream**: Stream class to write on files
  - **ifstream**: Stream class to read from files
  - **fstream**: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Input/Output

Institute of Space Technology

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream**: Stream class to write on files
  - **ifstream**: Stream class to read from files
  - **fstream**: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Input/Output

- C++ provides the following classes to perform output and input of characters to/from files
  - **ofstream**: Stream class to write on files
  - **ifstream**: Stream class to read from files
  - **fstream**: Stream class to both read and write from/to files.
- These classes are derived directly or indirectly from the classes istream and ostream

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

File Input/Output

Open a file

- In order to open a file with a stream object we use its member function open:
  - open (filename, mode);
- Filename is a string representing the name of the file to be opened
- Mode is an optional parameter with a combination of the following flag

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

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

File Input/Output

Open a file

- In order to open a file with a stream object we use its member function open:
  - **open (filename, mode);**
- Filename is a string representing the name of the file to be opened
- Mode is an optional parameter with a combination of the following flag

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

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

File Input/Output

Open a file

- In order to open a file with a stream object we use its member function open:
  - **open (filename, mode);**
- Filename is a string representing the name of the file to be opened
- Mode is an optional parameter with a combination of the following flag

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

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

File Input/Output

Open a file

- In order to open a file with a stream object we use its member function open:
  - **open (filename, mode);**
- Filename is a string representing the name of the file to be opened
- Mode is an optional parameter with a combination of the following flag

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

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

File Input/Output

Open a file

- classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- File streams opened in binary mode perform input and output operations independently of any format considerations
- Non-binary files are known as text files

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

- To check if a file stream was successful opening a file,use *is_open*

```
if (myfile.is_open()) { /* ok, proceed with
    output */ }
```

- Closing a file
  myfile.close();

File Input/Output

Open a file

- classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- File streams opened in binary mode perform input and output operations independently of any format considerations
- Non-binary files are known as text files

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

- To check if a file stream was successful opening a file,use *is_open*

```
if (myfile.is_open()) { /* ok, proceed with
    output */ }
```

- Closing a file
  myfile.close();

File Input/Output

Open a file

- classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- File streams opened in binary mode perform input and output operations independently of any format considerations
- Non-binary files are known as text files

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

- To check if a file stream was successful opening a file,use *is_open*

```
if (myfile.is_open()) { /* ok, proceed with
    output */ }
```

- Closing a file
  myfile.close();

File Input/Output

Open a file

Institute of Space Technology

- classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- File streams opened in binary mode perform input and output operations independently of any format considerations
- Non-binary files are known as text files

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

- To check if a file stream was successful opening a file,use *is_open*

```
if (myfile.is_open()) { /* ok, proceed with
    output */ }
```

- Closing a file
  myfile.close();

File Input/Output

Open a file

- classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- File streams opened in binary mode perform input and output operations independently of any format considerations
- Non-binary files are known as text files

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

- To check if a file stream was successful opening a file,use *is_open*

```
if (myfile.is_open()) { /* ok, proceed with
    output */ }
```

- Closing a file
  myfile.close();

Text files

Writing

Writing operations on text files are performed in the same way we operated with cout:

```
// 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;
}
```

Text files

Reading

Reading from a file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

File Input/Output

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- **bad()**
  Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- **fail()**
  Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
- **eof()**
  Returns true if a file open for reading has reached the end.
- **good()**
  It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function **clear()** can be used to reset the state flags.

File Input/Output

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- **bad()**
  Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- **fail()**
  Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
- **eof()**
  Returns true if a file open for reading has reached the end.
- **good()**
  It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function **clear()** can be used to reset the state flags.

File Input/Output

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- **bad()**
  Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- **fail()**
  Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
- **eof()**
  Returns true if a file open for reading has reached the end.
- **good()**
  It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function **clear()** can be used to reset the state flags.

File Input/Output

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- **bad()**
  Returns true if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
- **fail()**
  Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
- **eof()**
  Returns true if a file open for reading has reached the end.
- **good()**
  It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true. Note that good and bad are not exact opposites (good checks more state flags at once).

The member function **clear()** can be used to reset the state flags.

File Input/Output

get and put stream positioning

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- All i/o streams objects keep internally -at least- one internal position:
- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.
- Finally, fstream, keeps both, the get and the put position, like iostream.

tellg() and **tellp()**
These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
seekg() and **seekp()**
These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:
seekg (position);
seekp (position);

File Input/Output

get and put stream positioning

Institute of Space Technology

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- All i/o streams objects keep internally -at least- one internal position:
- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.
- Finally, fstream, keeps both, the get and the put position, like iostream.

tellg() and **tellp()**
These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
seekg() and **seekp()**
These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:
seekg (position);
seekp (position);

File Input/Output

get and put stream positioning

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- All i/o streams objects keep internally -at least- one internal position:
- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.
- Finally, fstream, keeps both, the get and the put position, like iostream.

tellg() and **tellp()**
These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
seekg() and **seekp()**
These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:
seekg (position);
seekp (position);

File Input/Output

get and put stream positioning

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- All i/o streams objects keep internally -at least- one internal position:
- ifstream, like istream, keeps an internal get position with the location of the element to be read in the next input operation.
- ofstream, like ostream, keeps an internal put position with the location where the next element has to be written.
- Finally, fstream, keeps both, the get and the put position, like iostream.

tellg() and **tellp()**
These two member functions with no parameters return a value of the member type streampos, which is a type representing the current get position (in the case of tellg) or the put position (in the case of tellp).
seekg() and **seekp()**
These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:
seekg (position);
seekp (position);

Text files

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin", ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n"
  return 0;
}
```

Notice the type we have used for variables begin and end:
streampos size;

File Input/Output

Binary

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- File streams include two member functions specifically designed to read and write binary data sequentially: **write** and **read**
- **write** is a member function of ostream (inherited by ofstream).
- **read** is a member function of istream (inherited by ifstream)

```
write ( memory_block, size );
read ( memory_block, size );
```

File Input/Output

Binary

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- File streams include two member functions specifically designed to read and write binary data sequentially: **write** and **read**
- **write** is a member function of ostream (inherited by ofstream).
- **read** is a member function of istream (inherited by ifstream)

```
write ( memory_block, size );
read ( memory_block, size );
```

File Input/Output

Binary

The following member functions exist to check for specific states of a stream (all of them return a bool value):

- File streams include two member functions specifically designed to read and write binary data sequentially: **write** and **read**
- **write** is a member function of ostream (inherited by ofstream).
- **read** is a member function of istream (inherited by ifstream)

```
write ( memory_block, size );
read ( memory_block, size );
```

Text files

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin", ios::in|ios::
      binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory"
        ;

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

for loop

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
    // whatever here...
}
```

for loop

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

for loop

Custom Count Down

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

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

While Loop

Institute of Space Technology

Custom Count Down

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

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

do-while loop

Institute of Space Technology

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

break statement

Institute of Space Technology

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

continue statement

Institute of Space Technology

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5)
        continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

go-to statement

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
mylabel:
  cout << n << ", ";
  n--;
  if (n>0) goto mylabel;
  cout << "liftoff!\n";
}
```

Switch statement

```
#include <iostream>
int x;
using namespace std;
int main ()
{
    cout<<"Enter value of x:\t";
    cin>>x;
switch (x) {
  case 1:
    cout << "x is 1\n";
    break;
  case 2:
    cout << "x is 2\n";
    break;
  default:
    cout << "value of x unknown\n";
  }
}
```

functions

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2)
       << '\n';
  cout << "The third result is " << subtraction (x,y)
      << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

functions

Default Values

```
// default values in functions
#include <iostream>
using namespace std;

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12) << '\n';
  cout << divide (20,4) << '\n';
  return 0;
}
```

functions

```cpp
// declaring functions prototypes
#include <iostream>
using namespace std;

void odd (int x);
void even (int x);

int main()
{
  int i;
  do {
    cout << "Please, enter number (0 to exit): ";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int x)
{
  if ((x%2)!=0) cout << "It is odd.\n";
  else even (x);
}

void even (int x)
{
  if ((x%2)==0) cout << "It is even.\n";
  else odd (x);
}
```

functions

Recursivity

```
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Scope

global & local variable

```
// inner block scopes
#include <iostream>
using namespace std;

int main () {
  int x = 10;
  int y = 20;
  {
    int x;   // ok, inner scope.
    x = 50;  // sets value to inner x
    y = 50;  // sets value to (outer) y
    cout << "inner block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
  }
  cout << "outer block:\n";
  cout << "x: " << x << '\n';
  cout << "y: " << y << '\n';
  return 0;
}
```

Scope

namespace

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

namespace first
{
  int x = 5;
}

namespace second
{
  double x = 3.1416;
}

int main () {
  {
    using namespace first;
    cout << x << '\n';
  }
  {
    using namespace second;
    cout << x << '\n';
  }
  return 0;
}
```

Scope

namespace

```
// static vs automatic storage
#include <iostream>
using namespace std;

int x;

int main ()
{
  int y;
  cout << x << '\n';
  cout << y << '\n';
  return 0;
}
```

Array

string, dimensional

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
  char answer1 [80];
  string answer2;
  cout << question1;
  cin >> answer1;
  cout << question2;
  cin >> answer2;
  cout << "Hello, " << answer1;
  cout << " from " << answer2 << "!\n";
  return 0;
}
```

Array

string, dimensional

```
// strings and NTCS:
#include <iostream>
#include <string>
using namespace std;

int main ()
{
 har myntcs[] = "some text";
string mystring = myntcs;  // convert c-string to s
cout << mystring;          // printed as a library
cout << mystring.c_str();  // printed as a c-strin
}
```

Structure

Basic

```
// example about structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;  int year;} mine, yours;

void printmovie (movies_t movie);

int main ()
{  string mystr;
  mine.title = "2001 A Space Odyssey";
  mine.year = 1968;
  cout << "Enter title: ";
  getline (cin,yours.title);
  cout << "Enter year: ";
  getline (cin,mystr);
  stringstream(mystr) >> yours.year;
  cout << "My favorite movie is:\n ";
  printmovie (mine);
  cout << "And yours is:\n ";
  printmovie (yours);
  return 0;}

void printmovie (movies_t movie)
{  cout << movie.title;
  cout << "  (" << movie.year << ")\n";}
```

Structure

Pointers

```
// array of structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;   int year; } films [3];

void printmovie (movies_t movie);

int main ()
{  string mystr;
  int n;
  for (n=0; n<3; n++)
  {     cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;   }
  cout << "\nYou have entered these movies:\n";
  for (n=0; n<3; n++)
    printmovie (films[n]);
  return 0;}

void printmovie (movies_t movie)
{  cout << movie.title;
  cout << " (" << movie.year << ")\n";}
```

Structure

Pointers

```
// pointers to structures
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

struct movies_t {
  string title;
  int year;};

int main ()
{  string mystr;
  movies_t amovie;
  movies_t * pmovie;
  pmovie = &amovie;
  cout << "Enter title: ";
  getline (cin, pmovie->title);
  cout << "Enter year: ";
  getline (cin, mystr);
  (stringstream) mystr >> pmovie->year;
  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";
  return 0;}
```

Institute of Space Technology

THE END

Thank you

Email