SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30008 Data Structures and Patterns
Lecture 2 Introduction to C++ Part 2

Coding Conventions

- Coding conventions establish guidelines for a specific programming language that recommend programming style, practices, and methods for each aspect of a program written in this language.
- Coding conventions are only applicable to the human maintainers and peer reviewers of a software project.
- Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual.
- Coding conventions are not enforced by compilers.

CamelCase

- CamelCase (or medial capitals) is a practice of writing names of variable or functions with some inner uppercase letters to denote embedded words:
  - InputStreamReader: character input stream
  - getEncoding: getter method for data encoding
  - MyIntegerArray: array variable
  - CreateWindowEx: Windows function

CamelCase

- Two popular variants:
  - Pascal InfixCaps - the first letter should be a capital, and any embedded words in an identifier should be in caps, as well as any acronym that is embedded.
  - Java - variables are mixed case with a lowercase first letter, methods should be verbs, in mixed case with the first letter lowercase, and classes should be nouns, in mixed case with the first letter of each internal word capitalized.

Borland-Inspired Style Guide Elements

- Field names should start with the letter ‘f’.
- Local variable names should start with the letter ‘l’.
- Parameter names should start with the letter ‘a’.
- Global variable names should start with the letter ‘g’.

Which Coding Standard To Use?

- Coding conventions are not enforced by compilers.
- Not following some or all of the rules has no impact on the executable programs created from the source code.
- Code standards facilitate program comprehension and make program maintenance easier.
- Every organization may enforce some sort of coding standards as part of organization's quality assurance process

Now back to C++ Classes…

What is a “friend”?

URL: https://www.youtube.com/watch?v=WCFGNdXSzus

Operator Overloading

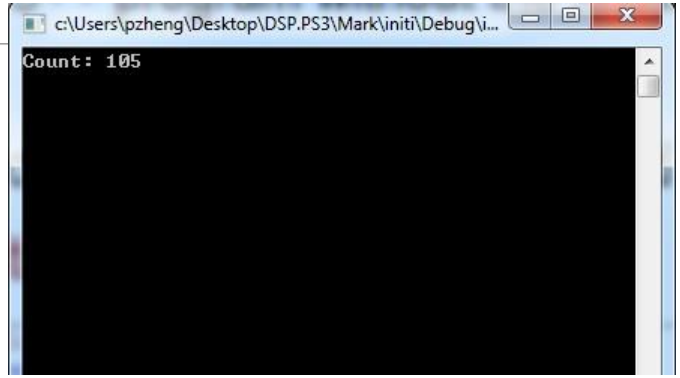
- Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviours depending on the types of its arguments.
- Operator overloading is usually only syntactic sugar. It can easily be emulated using function calls.

```
ClassName operator - (ClassName c2)
{
    ... .. ...
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... .. ....
    result = c1-c2;
    ... .. ...
}
```

Why Operator overloading

- You may write any C++ program without the knowledge of operator overloading.
- However, operator operating are profoundly used by programmers to make program intuitive.

```
1  #include <iostream>
2  using namespace std;
3
4  class Test
5  {
6     private:
7        int count;
8     public:
9         Test(): count(5){}
10        void operator ++()
11        {
12           count = count+100;
13        }
14        void Display() { cout<<"Count: "<<count; }
15 };
16
17 int main()
18 {
19     Test t;
20     // this calls "function void operator ++()" function
21     ++t;
22     t.Display();
23     getchar();
24     return 0;
25 }
```



Operator Overloading

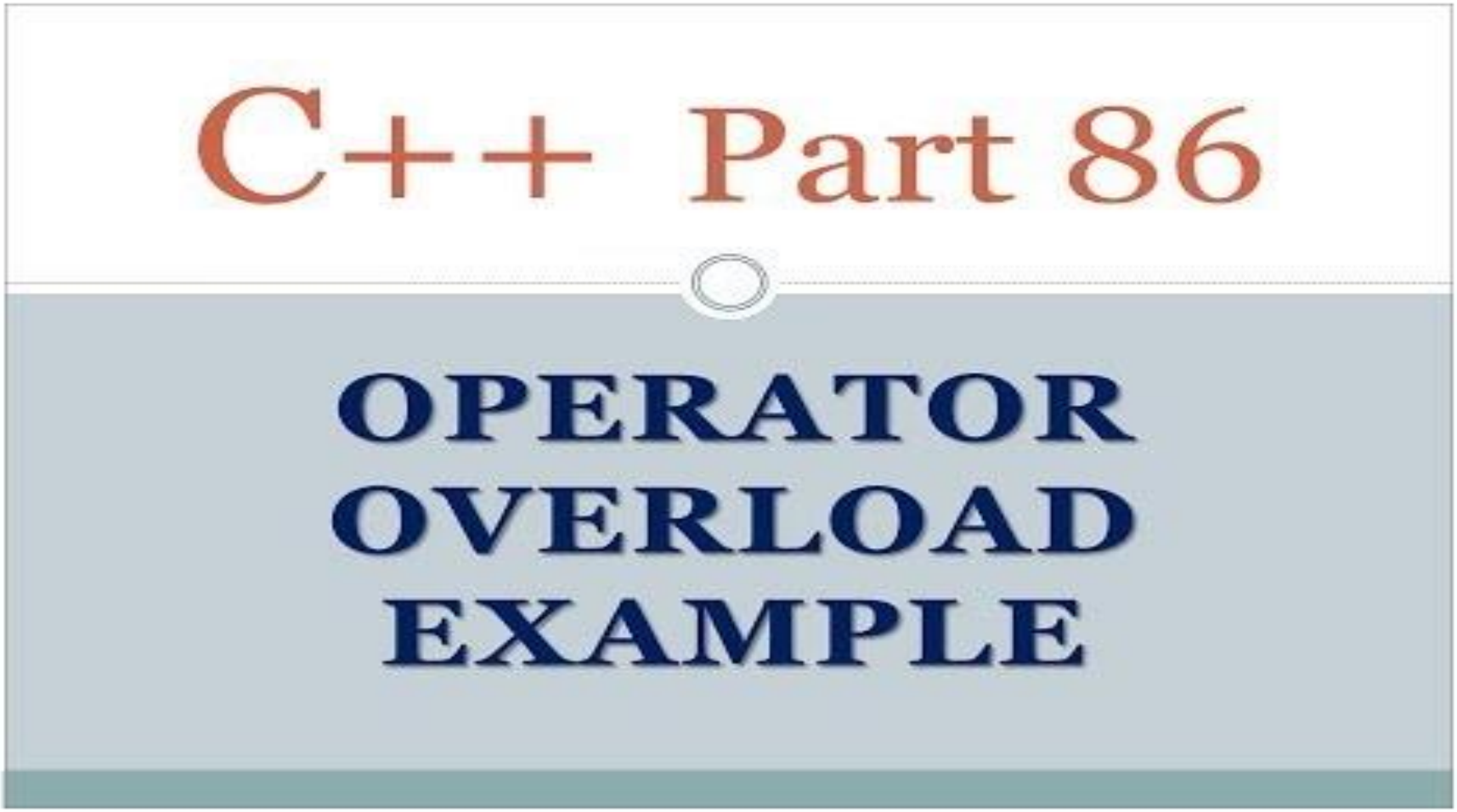
- C++ supports operator overloading.
- Overloaded operators are like normal functions, but are defined using a pre-defined operator symbol.
- You cannot change the priority and associativity of an operator.
- Operators are selected by the compiler based on the static types of the specified operands.

The Equivalence Operator ==

- The Boolean operator == defines a structural equivalence test for Book objects.
- We use `const` references for the Book arguments to pass Book objects by reference rather than copying their values into the stack frame of the operator ==.

```
// friend
bool operator==( const Book& aLeft, const Book& aRight )
{
  return aLeft.fUnitsSold == aRight.fUnitsSold &&
         aLeft.fRevenue == aRight.fRevenue &&
         aLeft.hasSameISBN( aRight );
}
```

== Operator Overload Example



URL: https://www.youtube.com/watch?v=UTcY6akR38I

The insertion and extraction operators

- The **insertion** (<<) **operator**, which is preprogrammed for all standard C++ data types, sends bytes to an output stream object. **Insertion operators** work with predefined "manipulators," which are elements that change the default format of integer arguments.
- The **extraction operator** (>>), which is preprogrammed for all standard C++ data types, is the easiest way to get bytes from an input stream object. Formatted text input **extraction operators** depend on white space to separate incoming data values.
- Reference

http://faculty.cs.niu.edu/~hutchins/csci241/io-op.htm

The Input Operator >>

Book.cpp

```
// friend
istream& operator>>( istream& aIStream, Book& aItem )
{
  double lPrice;

  aIStream >> aItem.fISBN >> aItem.fUnitsSold >> lPrice;
  // check that the inputs succeeded
  if ( aIStream )
    { aItem.fRevenue = aItem.fUnitsSold * lPrice; }
  else
    { // reset to default state
      aItem = Book();
    }
  return aIStream;
}
```

Line: 1 Column: 1 C++

Return reference to input stream.

The Output Operator <<

Book.cpp

```
// friend
ostream& operator<<( ostream& aOStream, const Book& aItem )
{
  aOStream << aItem.fISBN << "\t" << aItem.fUnitsSold << "\t"
           << aItem.fRevenue << "\t" << aItem.getAveragePrice();
  return aOStream;
}
```

Line: 39 Column: 7 C++

Return reference to output stream.

The overloaded >> and << operators used in main()

ReadWriteBooks.cpp

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

using namespace std;

int main()
{
    Book lBook;

    cin >> lBook;          // read book data
    cout << lBook << endl; // write book data

    return 0;
}
```

Line: 15 Column: 1 C++ Tab Size: 4 main

Terminal

```
Sela:HIT3303 Markus$ ./ReadWriteBooks
0-201-70353-X 4 24.99
0-201-70353-X   4       99.96   24.99
Sela:HIT3303 Markus$
```

Overloading stream insertion (<<) and extraction (>>) operators in C++

- In C++, stream insertion operator “<<” is used for output and extraction operator “>>” is used for input.
- We must know the following things before we start overloading these operators.
  1) cout is an object of ostream class
  2) cin is an object of istream class
  3) These operators must be overloaded as a global function. And if we want to allow them to access private data members of class, we must make them a **friend**.

Why overloaded as global?

- In operator overloading, if an operator is overloaded as member, then it must be a member of the object on left side of the operator.
- For example, consider the statement “ob1 + ob2” (let ob1 and ob2 be objects of two different classes).
- To make this statement compile, we must overload ‘+’ in class of ‘ob1’ or make ‘+’ a global function.

Why overloaded as global?

- The operators ‘<<' and '>>' are called like 'cout << ob1' and 'cin >> ob1'.
- If we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time.
- Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

An example (In “Complex” class)

```
#include
<iostream>  using
namespace std;
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {  real = r;  imag = i; }
    friend ostream & operator <<
(ostream  &out, const Complex &c);
    friend istream & operator >>
    (istream &in,
Complex &c);
};
```

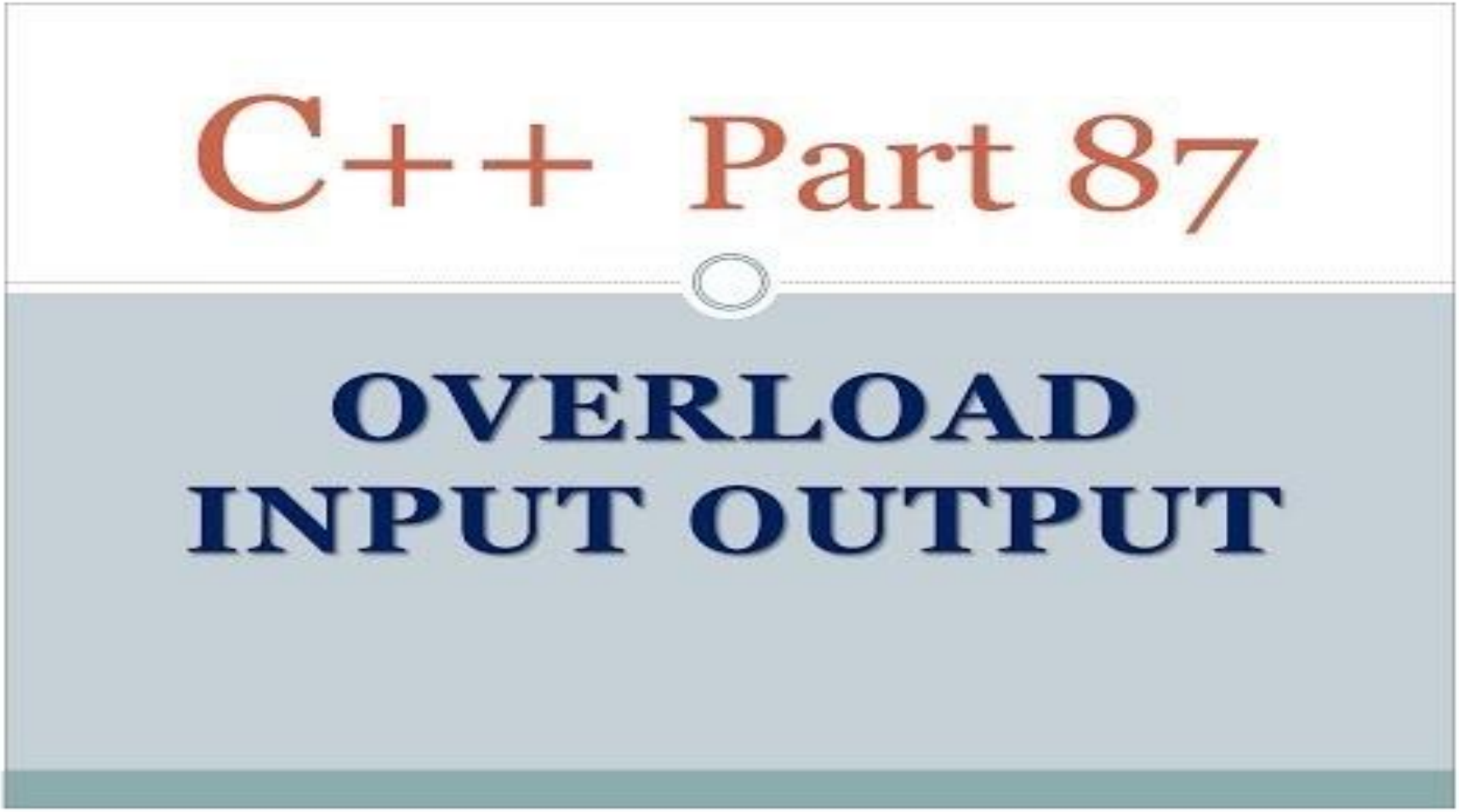
```
ostream & operator << (ostream &out,
const  Complex &c)
{
    out << c.real;
    out << "+i" << c.imag <<
    endl;  return out;
}

istream & operator >> (istream
&in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary
    Part ";  in >> c.imag;
    return in;
}
```

An example (In main() function)

```
int main()
{
    Complex c1;
    cin >> c1;
    cout << "The Complex object is ";
    cout << c1;
    return 0;
}
```

'cin>>c1' actually means "operator>>(cin,c1)" which is read as "Extraction operator is called and cin and c1 is passed as an argument".

>> and << Operator Overload Example



URL: https://www.youtube.com/watch?v=1X_QJAmdb-c

Class Book - Member Operator +=

Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

The Member Operator +=

The operator += is defined as a member of class Book!

Book.cpp

```
// member operator
Book& Book::operator+=( const Book& aRHS )
{
  fUnitsSold += aRHS.fUnitsSold;
  fRevenue += aRHS.fRevenue;
  return *this;
}
```

Line: 39 Column: 7 C++ Tab Size: 4 std

Return reference to receiver.

The Overloaded Operator +

AddSales.cpp

```cpp
// overloaded operator
Book operator+( const Book& aLeft, const Book& aRight )
{
  Book Result( aLeft );

  Result += aRight;
  return Result; // return by value
}
```

Line: 14 Column: 1 C++ Tab Size: 4 std

- operator+ : (Book,Book)→ Book

The member operator (+) used in main()

AddSales.cpp

```
1  #include <iostream>
2  #include "Book.h"
3
4  using namespace std;
5
6  // overloaded operator
7  Book operator+( const Book& aLeft, const Book& aRight )
8  {
9    Book Result( aLeft );
10
11   Result += aRight;
12   return Result; // return by value
13 }
14
15 int main()
16 {
17     Book lBook1, lBook2;
18
19     cin >> lBook1 >> lBook2;          // read books
20     cout << lBook1 + lBook2 << endl;  // write sales data
21
22     return 0;
23 }
24
```

Line: 7 Column: 52 C++ Tab Size: 4 std

Terminal

```
Sela:HIT3303 Markus$ ./AddSales
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
0-201-78345-X   5       110     22
Sela:HIT3303 Markus$
```

Class Book - Member Functions

Book.h

```
#ifndef BOOK_H_
#define BOOK_H_

#include <iostream>

class Book
{
 private:
   std::string fISBN;
   unsigned fUnitsSold;
   double fRevenue;

 public:
   Book() : fUnitsSold(0), fRevenue(0.0) {}
   Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
   Book( std::istream& aIStream ) { aIStream >> *this; }

   Book& operator+=( const Book& aRHS );

   friend bool operator==( const Book& aLeft, const Book& aRight );
   friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
   friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );

   double getAveragePrice() const;
   bool hasSameISBN( const Book& aRHS ) const;
};

#endif /* BOOK_H_ */
```

The Member Functions

Book.cpp

```
// member function
double Book::getAveragePrice() const
{
    return fRevenue / fUnitsSold;
}

// member function
bool Book::hasSameISBN( const Book& aRHS ) const
{
    return fISBN == aRHS.fISBN;
}
```

Line: 11 Column: 2 C++ Tab Size: 4 std

Automatic type conversion

Private member variables are visible within the scope of class Book.

AddSalesSecure (+ operator being used)

AddSalesSecure.cpp

```
int main()
{
    Book lBook1, lBook2;

    cin >> lBook1 >> lBook2;                // read books

    if ( lBook1.hasSameISBN( lBook2 ) )     // security check
    {
        cout << lBook1 + lBook2 << endl;    // write sales data
        return 0;
    }
    else
    {
        cerr << "Error, data must refer to same ISBN!" << endl;
        return 0;
    }
}
```

Line: 32 Column: 1 C++ Tab Size: 4 main

AddSalesContinuously (+=, >> and << operators being used)

AddSalesContinuously.cpp

```
6   int main()
7   {
8       Book lTotal, lCurrent;
9
10      if ( cin >> lTotal )                // is there data to process?
11          while ( cin >> lCurrent )       // continue with transactions
12          {
13              if ( lTotal.hasSameISBN( lCurrent ) )
14                  lTotal += lCurrent;
15              else
16              {
17                  cout << lTotal << endl;
18                  lTotal = lCurrent;
19              }
20              cout << lTotal << endl;
21          }
22      else
23      {
24          cerr << "Error, no data!" << endl;
25          return -1;
26      }
27      return 0;
28  }
29
```

Line: 28 Column: 1 C++ Tab Size: 4 main

Terminal

```
Sela:HIT3303 Markus$ ./AddSalesContinuously
0-201-78345-X 2 25.00
0-201-78345-X 3 20.00
0-201-78345-X   5       110     22
0-201-78345-Y 10 15.00
0-201-78345-X   5       110     22
0-201-78345-Y   10      150     15
Sela:HIT3303 Markus$
```

^D

A Small Note: Inlining

- C++ offers function inlining that may reduce the calling overhead associated with a function.
- The basic idea is to save time at a cost in space. **Inline functions** are a lot like a placeholder. Once you define an **inline function**, using the '**inline**' keyword, whenever you call that **function** the compiler will replace the **function** call with the actual code from the **function**.
- **However:**
  - Inlining increases the code size and sometimes may not be applicable (e.g., recursive or virtual functions).
  - Inlining can give better cache performance, but too many inlined functions can result in a large code size and page faults, hence defeating the original aim of inlining. (Inlining is not useful for embedded systems, where large binaries are not preferred.)
  - The compiler may choose to ignore inline requests.

Inlined Member Functions

```
6   class Book
7   {
8    private:
9      std::string fISBN;
10     unsigned fUnitsSold;
11     double fRevenue;
12
13   public:
14     Book() : fUnitsSold(0), fRevenue(0.0) {}
15     Book( const std::string& aBook ) : fISBN(aBook), fUnitsSold(0), fRevenue(0.0) {}
16     Book( std::istream& aIStream ) { aIStream >> *this; }
17
18     Book& operator+=( const Book& aRHS );
19
20     friend bool operator==( const Book& aLeft, const Book& aRight );
21     friend std::istream& operator>>( std::istream& aIStream, Book& aItem );
22     friend std::ostream& operator<<( std::ostream& aOStream, const Book& aItem );
23
24     double getAveragePrice() const;
25     bool hasSameISBN( const Book& aRHS ) const;
26  };
```

For inlining to work the functions need to be defined in the header file (.h), that is, within the class specification.

BookInline.h

```
24
25  class Book
26  {
27
28      ...
29
30      // member function
31      inline double getAveragePrice() const
32      {
33          return fRevenue / fUnitsSold;
34      }
35
36      // member function
37      inline bool hasSameISBN( const Book& aRHS ) const
38      {
39          return fISBN == aRHS.fISBN;
40      }
41  };
42
```

Line: 20 Column: 1 C++ Tab Size: 4 Book

‘this’ pointer in C++

- The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.
- ‘this’ pointer is a constant pointer that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

A static member is a member of the class which is shared by all objects of the class. No matter how many objects of the class are created, there will only be one copy of the static member.

‘this’ pointer in C++

- There are several situations that we use “this” pointer

1)When local variable’s name is same as member’s name

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

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
       // The 'this' pointer is used to retrieve the object's x
       // hidden by the local variable 'x'
       this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};

int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

‘this’ pointer in C++

- 2) To return reference to the calling object

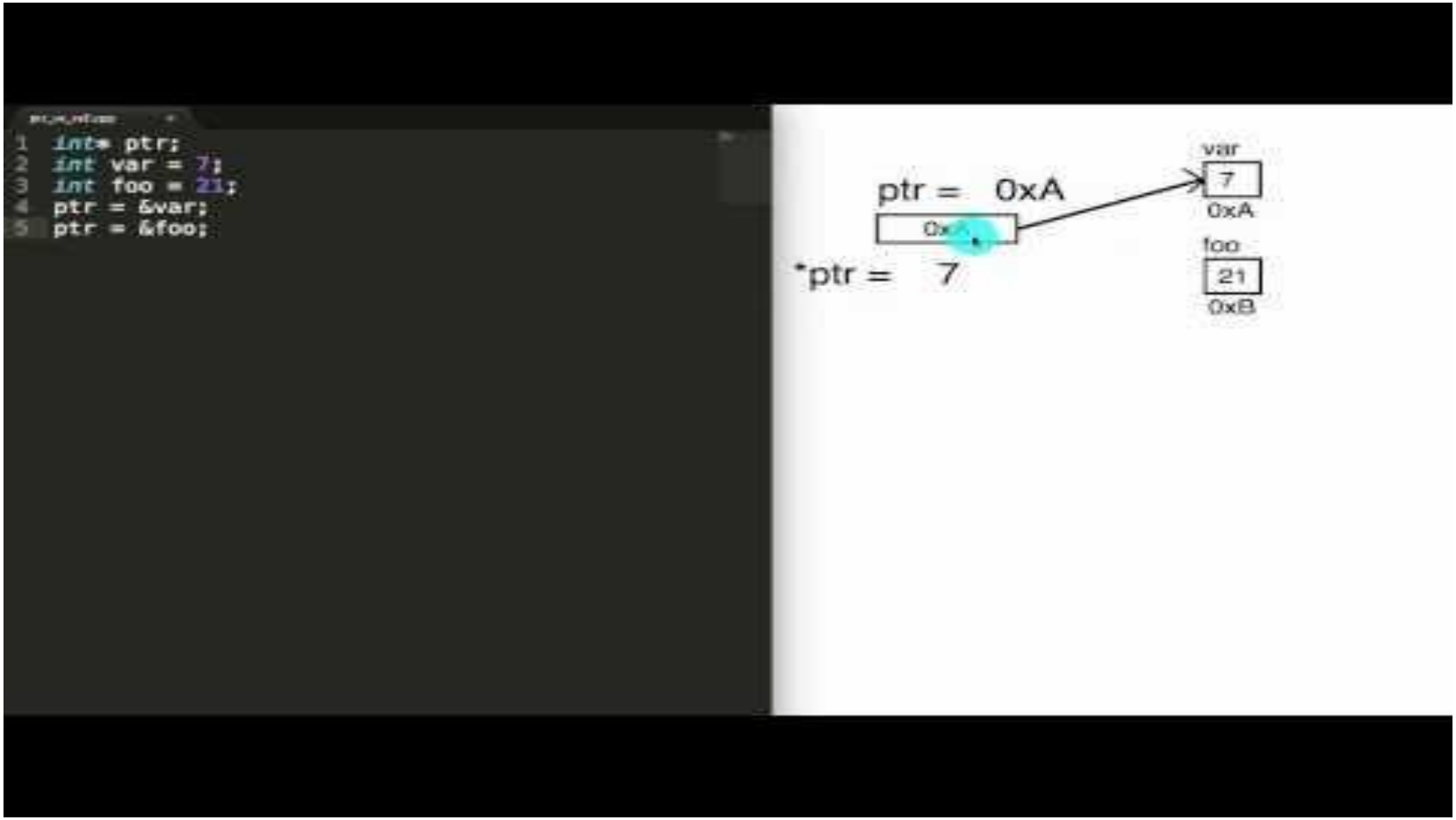
```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

How do we use “this”?

```
public:
    void printThisPointer() const
    {
        cout << this << endl;
    }
};

void main()
{
    Cow betsy;
    cout << &betsy << endl;
    betsy.printThisPointer();
    Cow georgy;
    cout << &georgy << endl;
    georgy.printThisPointer();
    Cow
}
```

URL: https://www.youtube.com/watch?v=I2-RmR7JRdc

Difference between a pointer and a reference



URL: https://www.youtube.com/watch?v=sxHng1iufQE

Difference between pass by value, pass by reference and pass by pointer

```
void passByRef(int & ref);
void passByPtr(int * ptr);

int main()
{
    int x = 5;
    printf("x = %i\")
    return 0;
}

void passByVal(int val)
{
    val = 10;
    printf("val = %i\n", val);
}

void passByRef(int & ref)
{
    ref = 20;
    printf("ref = %i\n", ref);
}

void passByPtr(int * ptr)
{
    *ptr = 30;
    printf("*ptr = %i\n", *ptr);
}
```

```
PaulProg: ls
passby.cpp
PaulProg:
```

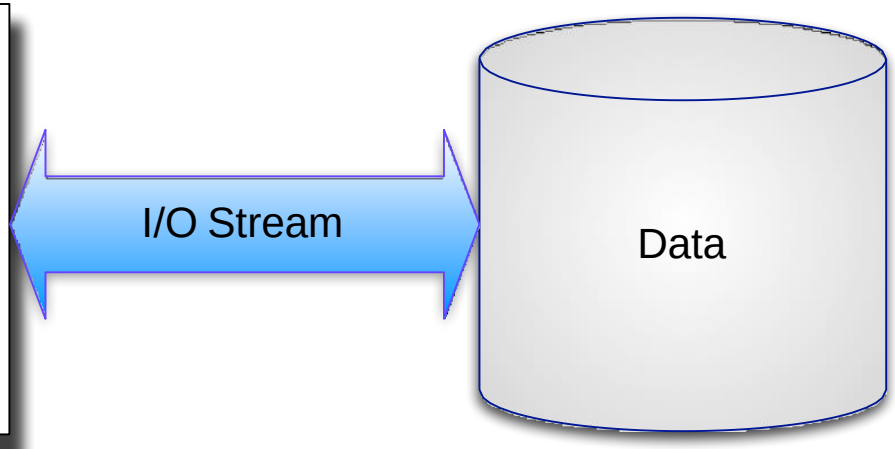
URL: https://www.youtube.com/watch?v=UWYnUGnChhA

SWINBURNE UNIVERSITY OF TECHNOLOGY

Data Input/Output

I/O Streams

```
#include <iostream>

int main()
{


    return 0;
}
```

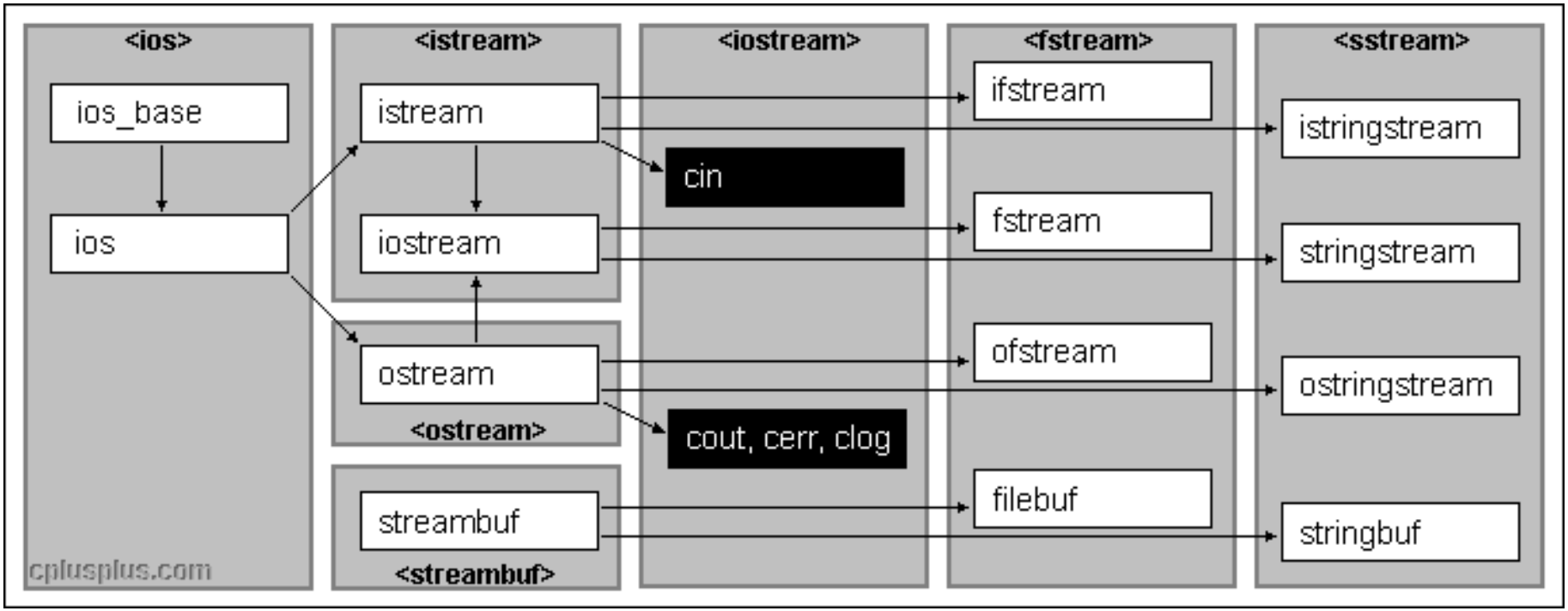


- In C++, input or output, independent of the type of I/O medium, are mapped into logical **data streams** with common properties.
- Two forms of mapping are supported: **text streams** and **binary streams**.

I/O Media

- Streams can be associated with
  - **Physical devices** (e.g., console - cin, cout)
  - **Files** (e.g., coefficients.txt, sales.dbf)
  - **Structured storage** (e.g., int values[10])

Input/Output library



You can read more about the Input/Output library here:
http://www.cplusplus.com/reference/iolibrary/

A Program that uses the C++ I/O library.

SimpleIO.cpp

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

int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

Line: 14 Column: 1 C++ Tab Size: 4 main

C:\WINDOWS\system32\cmd.exe

```
Enter two numbers:
7
12
The sum of 7 and 12 is 19
Press any key to continue . . .
```

- cin and cout represent the standard input stream and the standard output stream in every C++ program.

The Standard Input Stream cin

- **cin** is an object of class istream that represents the **standard input stream**. It corresponds to **stdin** in C.
- **cin** is a globally visible object that is readily available to any C++ compilation unit (i.e., a .cpp-file) that includes the **library iostream**.
- **cin** receives **input** either from the keyboard or a stream associated with the standard input stream.
- We use the **operator >>** to fetch formatted data or use the methods read of get to retrieve unformatted data from the standard input stream.

The Standard Output Stream cout

- **cout** is an object of class **ostream** that represents the **standard output stream**. It corresponds to **stdout** in C.
- **cout** is a globally visible object that is readily available to any C++ compilation unit (i.e., a .cpp-file) that includes the library **iostream**.
- **cout sends data** either to the console (as text) or a stream associated with the standard output stream.
- We use the **operator <<** to push formatted data or use the methods write or put to send unformatted data to the standard output stream.

Chaining Input and Output

output - Notepad

File Edit Format View Help

```
Enter two numbers:
The sum of 7 and 12
```

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

int main()
{
    cout << "Enter two numbers:" << endl;
    int v1, v2;
    cin >> v1 >> v2;
    cout << "The sum of " << v1 << " and " << v2
         << " is " << v1 + v2 << endl;
    return 0;
}
```

Ln: 14 Column: 1 Tab Size: 4 main

input - Notepad

File Edit Format View Help

```
7
12
```

Standard Streams Summary

- In C++, there are three standard text streams:
  - **cin** - text input stream
  - **cout** - text output stream
  - **cerr** - secondary error text output stream

Input Stream Variations

- **istream** objects can be used to **read** and interpret **input** from sequences of characters. istream provides functions to perform input operations divided in two main groups: formatted input and unformatted (raw) input.
- **ifstream** provides an interface to **read** data from **files** as input streams. objects of type ifstream maintain private memory to perform buffered I/O.
- **istringstream** provides an interface to manipulate **strings** as **input streams**. istringstream objects provide a memory stream to exchange data between two stream-based data endpoints.

Output Stream Variations

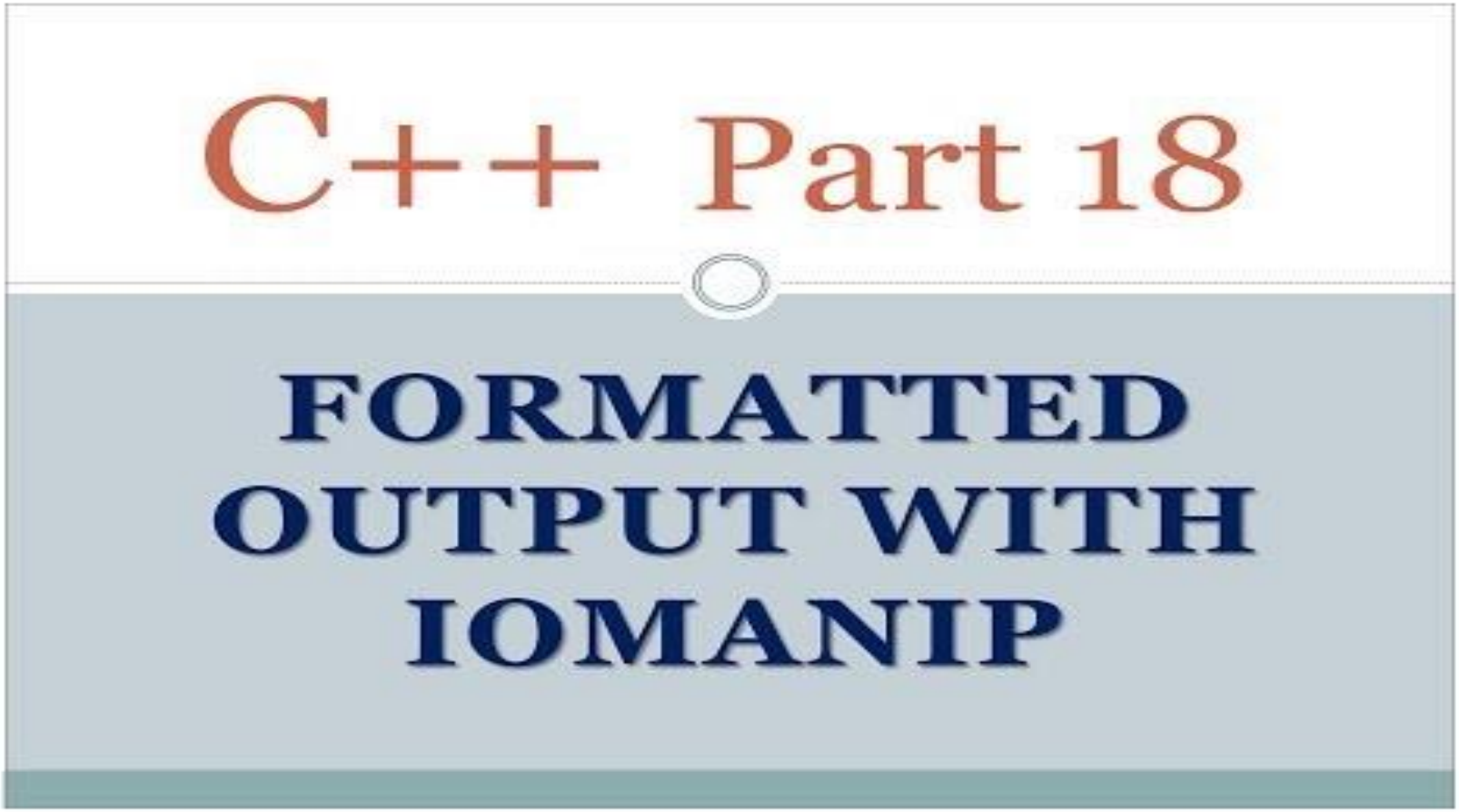
- **ostream** objects can be used to **write** and format **output** as sequences of characters. ostream provides functions to perform output operations divided in two main groups: formatted output and unformatted (raw) output.
- **ofstream** provides an interface to **write** data to **files** as output streams. objects of type ofstream maintain private memory to perform buffered I/O.
- **ostringstream** provides an interface to manipulate **strings** as **output streams**. ostringstream objects provide a memory stream to exchange data between two stream-based data endpoints.

iomanip library (When to use it?)

- iostream has a 6 digit precision as a default, therefore even if the digit is 1234.12345, it will be rounded off and displayed as 1234.12. By using iomanip, we can set the digit precision to be higher.
  - We can do this by adding #include <iomanip> and using cout << setprecision(10);

- iomanip also allows us to set the field width for better, more consistent spacing for rows of data.
  - We can do this using setw(int) to set the size we want.

iomanip library (When to use it?)



URL: https://www.youtube.com/watch?v=FkBMLIL6llo

End of Introduction to C++ Part 2

Now let's proceed with Lab 2 ☺