CS2040C Data Structures and Algorithms

More on C++

Lecture Overview

- Object Oriented Features in C++
  - Class and Object
  - Inheritance
  - Template Class
- Useful C++ Libraries
  - String
  - Stream
  - Standard Template Library

Object Oriented Languages

Definition and Motivation

Object Oriented Languages

- All programming languages like C, C++, Java etc has an underlying **programming model**
  - Also known as **programming paradigms**
- Programming Model tells you:
  - How to organize the information and processes needed for a solution (program)
  - Allows/facilitates a certain way of thinking about the solution
  - Analogy: it is the “world view” of the language
- Popular programming paradigms:
  - **Procedural** : C, Pascal
  - **Object Oriented**: Java, C++
  - etc

Bank Account : A simple illustration

- Let's look at C implementation of a simple bank account
- A bank account contains:
  - *Account Number* : integer
  - *Balance* : double (should be >= 0)
- Basic operations:
  - *Withdrawal*
  - *Deposit*
- Using structure is the best approach in C

Bank Account : C Implementation

```
typedef struct {
    int acctNum;
    double balance;
} BankAcct;
```

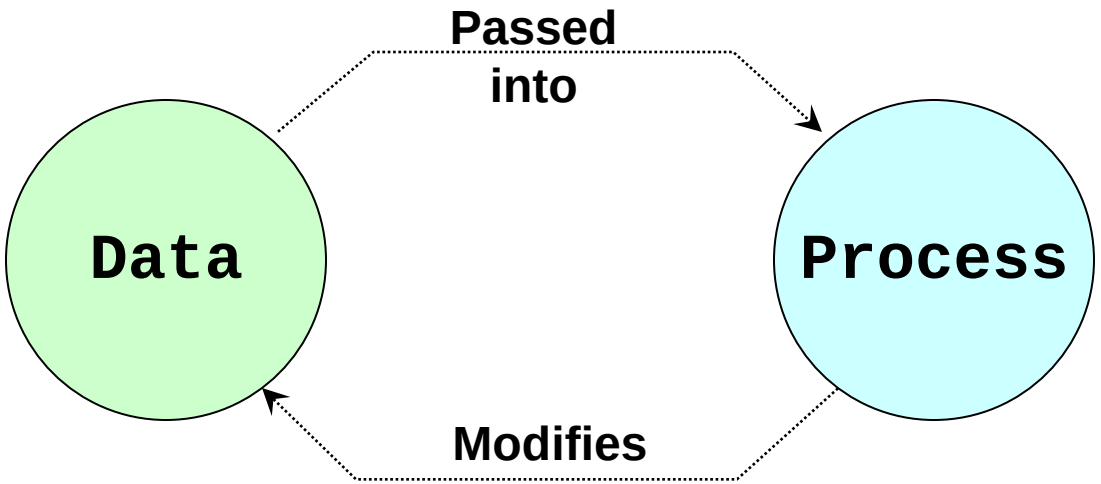
Structure to hold information for bank account

```
void initialize( BankAcct* baPtr, int anum)
{   baPtr->acctNum = anum;
    baPtr->balance = 0;
}

int withdraw( BankAcct* baPtr, double amount)
{   if (baPtr->balance < amount)
       return 0;          //indicate failure
    baPtr->balance -= amount;
    return 1;             //success
}

void deposit( BankAcct* baPtr, double amount)
{  ... Code not shown ... }
```

Functions to provide basic operations

Bank Account : C Implementation

- C treats the data (structure) and process (function) as separate entity:



Procedural Languages

- C is a typical procedural language
- Characteristics of procedural languages:
  - Data and Process are separated
  - Data and Process are “passive”
    - The caller initiates the execution
  - User must make sure the data and process are used correctly
    - No good way to prevent intentional / accidental wrong use

Procedural Languages

Correct use of `BankAcct` and its operations

```
BankAcct ba1;

initialize(&ba1, 12345);
deposit(&ba1, 1000.50);
withdraw(&ba1, 500.00);
withdraw(&ba1, 600.00);
     ...
```

Wrong and malicious exploits of `BankAcct`

```
BankAcct ba1;

deposit(&ba1, 1000.50);

initialize(&ba1, 12345);
ba1.acctNum = 54321;

ba1.balance = 10000000.00;
       ...
```

Forgot to initialize

Account Number should not change!

Balance should be changed by authorized operations only

Procedural Languages

- Disadvantages of procedural languages:
  - Hard to protect data from “unauthorized” modification
  - Hard to debug
  - Hard to expand / modify
    - How to introduce a new type of bank account (e.g. current account)?
      - Without affecting the current implementation
      - Without recoding the common stuff

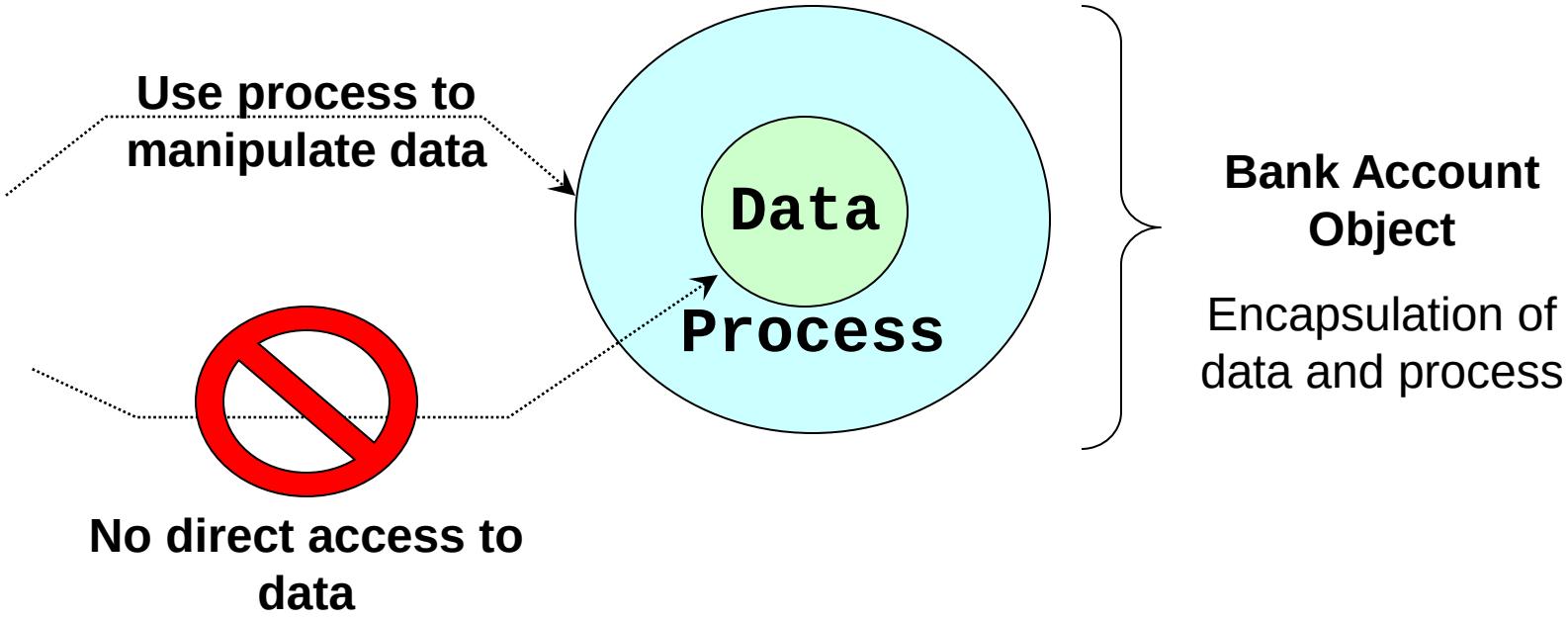
- Bottom line:
  - Usually fast to code and efficient in execution
  - Less overhead when designing

Object Oriented Languages

- Main features:
  - **Encapsulation**
    - Group data and associated processes into a single package
    - Hide internal details from outsider
  - **Inheritance**
    - A meaningful way of extending current implementation
    - Introduce logical relationship between packages
  - **Polymorphism**
    - Behavior of the processes changes according to the actual type of data

Bank Account : OO Implementation

- A conceptual view of equivalent object-oriented implementation for the Bank Account



C++: Object Oriented Features

What makes C++ Object Oriented

Encapsulation in C++: Classes

- A package of data and processes is known as a **class** in C++
- A **class** is a user defined **data type**
- Variables of a class are called **objects**
- Each class contains:
  - **Data:** each object has an independent copy
  - **Functions:** process to manipulate data in an object
- Terminology:
  - Data of a class: member data (**attributes**)
  - Functions of a class: member functions (**methods**)

Accessibility of attributes and methods

- Data and methods in a class can have different level of accessibilities (visibilities)
- **`public`**
  - Anyone can access
  - Usually intended for methods only
- **`private`**
  - Only object of the same class can access
  - Recommended for all attributes
- **`protected`**
  - Only object of the same class or its children can access
  - Recommended for attributes/methods that are common in a “family”

Bank Account: C++ Implementation

```
class BankAcct {

private:
  int _acctNum;
  double _balance;

public:
  int withdraw( double amount )
  {   if (_balance < amount)
         return 0;
      _balance -= amount;
      return 1;
  }

  void deposit( double amount )
  {  ... Code not shown ... }

};
```

All attributes/methods from this point onward are private

Convention: Prefix a " _ " for attributes name. For easy identification.

Note the parameter. Data of bank account is no longer passed in.

Bank Account: Class and Object

- The class declaration defines a **new data type**
  - No actual variables are allocated!
- To have a variable of a class:
  - Create (instantiate) **object**
- The distinction between class and object
  - Similar to structure declaration and structure variable in C
  - Analogy: class == blue print, object == actual house
- To access **public** data or method of an object
  - Use the "." dot operator
  - Similar to structure access in C

Bank Account: Example usage

```
// BankAcct class declaration from slide 16

int main( )
{
    BankAcct ba1;

    ...
    ba1.deposit(1000);
    ba1.withdraw(500.25);


    ba1._acctNum = 1357;
    ba1._balance = 10000000;

}
```

Question: How to initialize?

Interacts with object using **public methods**

Error: Outsider cannot access **private attributes**

Constructors

- The previous implementation for bank account is incomplete
  - account number and balance are not initialized
- Each class has one or more specialized methods known as **constructor**
  - Called **automatically** when an object is created
- **Default constructor**
  - Takes in no parameter
  - Automatically provided by the compiler if programmer does not define **any constructor method**
- **Non-default constructor**
  - Can take in parameter
  - Can have multiple different constructors

Bank Account: Two Example Constructors

```
class BankAcct {

     ..............

public:
  BankAcct( int aNum )
  {  _acctNum = aNum;
     _balance = 0;
  }


  BankAcct( int aNum, double amt )
  : _acctNum( aNum), _balance( amt )
  {

  }

     ..............
};
```

Constructor method has the same name as the class with **no return type**

Alternative syntax to initialize object attributes. Known as **initialization list**. Only valid in constructor method.

Bank Account: Example usage 2

```
// BankAcct class declaration from slide 20

int main( )
{
    BankAcct ba1( 1234 );

    BankAcct ba2( 1235, 500.00 );

    BankAcct ba3;

}
```

Make use of 1st constructor

Make use of 2nd constructor

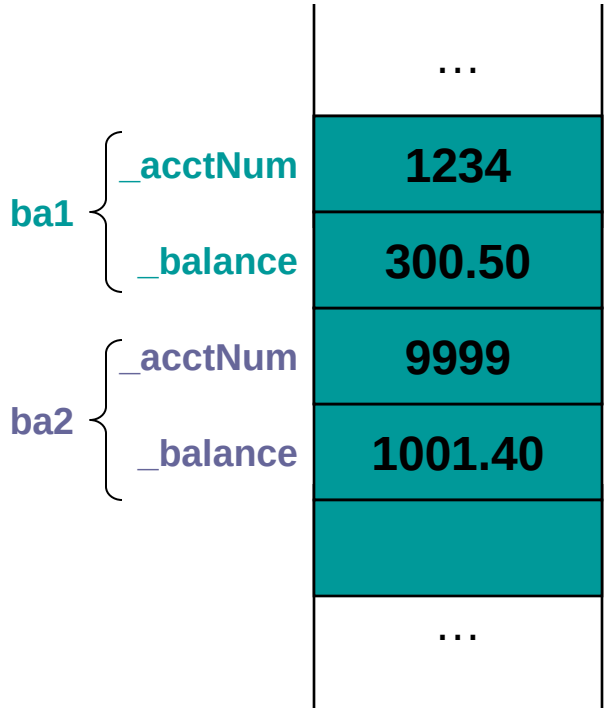
Error: default constructor no longer valid

- If programmer defines extra constructors:
  - Compiler no longer provides the default constructor
  - Programmer has to define default constructor if it is useful

Object: Memory Snapshot

- An in-depth look at method execution
  - Memory snapshot is provided for better understanding

```
class BankAcct {
//... other code not shown ...
int withdraw( double amount )
{   if (_balance < amount)
          return 0;
      _balance -= amount;
      return 1;
}};

int main( )
{   BankAcct ba1( 1234, 300.50 );
    BankAcct ba2( 9999, 1001.40 );

    ba1.withdraw(100.00);
    ba2.withdraw(100.00);
}
```

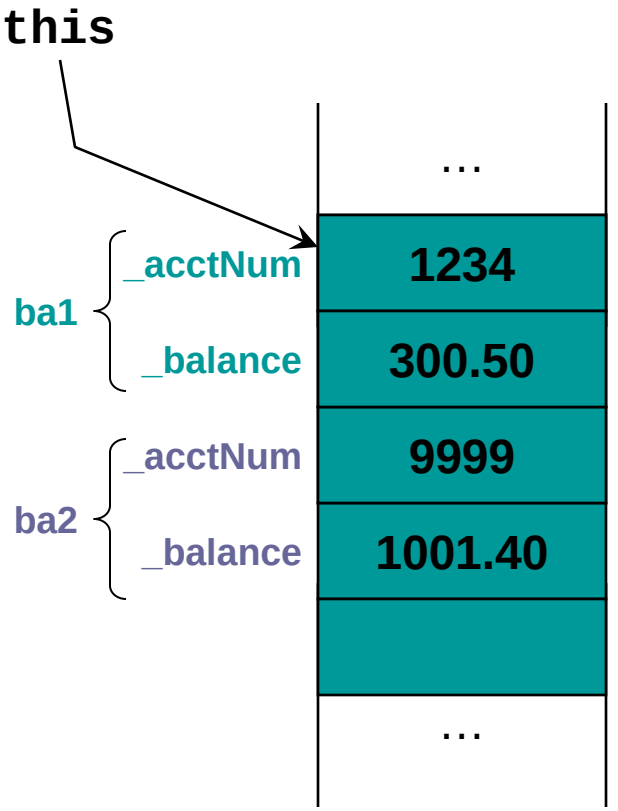


Object: What is "**`this`**"

- A common confusion:
  - How does the method "know" which is the "object" currently executing?
  - E.g. when **`withdraw( )`** accesses the attribute **`_balance`**, which **`_balance`** is used?
- Whenever a method is called, e.g. **`ba1.withdraw()`**, a **pointer to the calling object** is set automatically
  - Given the name **"this"** in C++, meaning "this particular object"
- All attributes/methods are then accessed implicitly through this pointer

Object: What is "**this**"

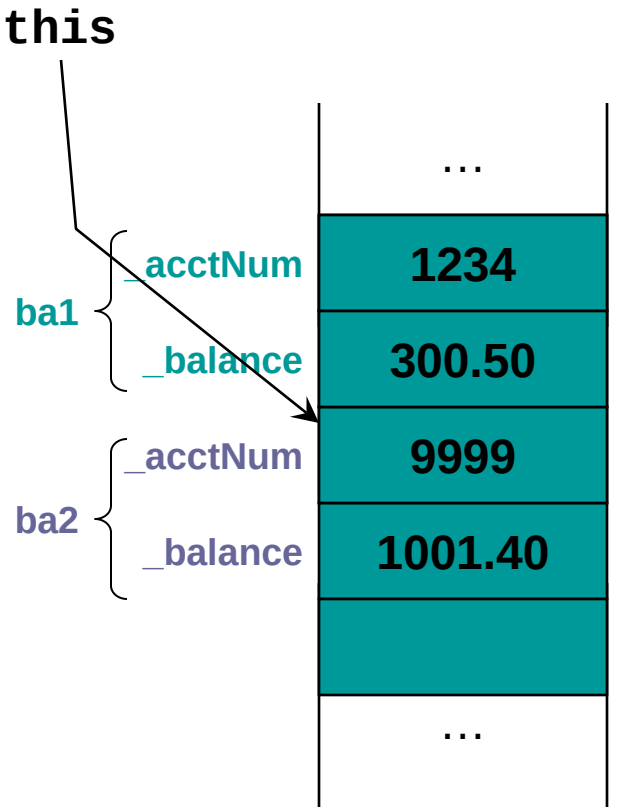
```
class BankAcct {
//... other code not shown ...
int withdraw( double amount )
{
      if (_balance < amount)
         return 0;
      _balance -= amount;
      return 1;
}
};

int main( )
{   BankAcct ba1( 1234, 300.50 );
    BankAcct ba2( 9999, 1001.40 );

    ba1.withdraw(100.00);
    ba2.withdraw(100.00);
}
```



At this point

Object: What is "**this**"

```
class BankAcct {
//... other code not shown ...
int withdraw( double amount )
{
      if (_balance < amount)
         return 0;
      _balance -= amount;
      return 1;
}
};

int main( )
{   BankAcct ba1( 1234, 300.50 );
    BankAcct ba2( 9999, 1001.40 );

    ba1.withdraw(100.00);
    ba2.withdraw(100.00);
}
```



At this point

Object: Passed by value

- Objects are **passed by value** (similar to structure)

```
// BankAcct class definitions
void transfer(BankAcct& fromAcct,
              BankAcct& toAcct, double amt)
{
   fromAcct.withdraw(amt);
   toAcct.deposit(amt);
}

int main()
{   //Simple testing on object passing
    BankAcct ba1(1234,1000.00), ba2(1235, 2000.00);
    transfer(ba2, ba1, 500.00);
}
```

Note that the Bank Accounts are passed by reference.

Question: What if we remove the "**&**"?

- Additionally, objects tend to contain lots of attributes
  - Recommended to **pass all objects by reference**
  - Caution: Any function/method that modifies the object will affect the actual parameter!

Destructor

- Destructor is a specialized method of a class
  - Called automatically when
    - Object of the class goes out of **scope**
    - Object of the class gets deleted explicitly
- Destructor should be defined for classes that
  - Are allocated memory dynamically
  - Requested system resources (e.g. file)
- Syntax for destructor:
  - Method with same name as the class:
    - Prefixed by ~
    - Empty parameter list and no return type
  - Only one per class
- If destructor is not implemented:
  - A default destructor will be given automatically

Portion of code delimited by curly braces **{ }**

Destructor: An Example

```
/* class Simple ➔ */

void f()
{  Simple s(999);
   cout << "End of f()\n";
}

int main()
{   Simple s(123), *sptr;

    if (true) {
        Simple s(456);
    }

    f();

    sptr = new Simple(789);
    delete sptr;

    cout << "End of main\n";
}
```

B

A

C

```
class Simple{
private:
   int _id;
public:
   Simple(int i):_id(i){
       cout << _id << " alive!!\n";
   }
   ~Simple(){
       cout << _id << " died!!\n";
   }
};
```

Output:

```
123 alive!!
456 alive!!
456 died!!
999 alive!!
End of f()
999 died!!
789 alive!!
789 died!!
End of Main
123 died!!
```

A

B

C

Life of an Object

- Allocation ("Birth"):
  - When:
    - Object declaration or ***new*** keyword is used
  - Steps:
    - The object is allocated in memory
    - Constructor of the object is called
- Alive:
  - After the construction is performed successfully
  - Object ready to be used
- Deallocation ("Death"):
  - When:
    - Object goes out of scope or ***delete*** keyword is used
  - Steps:
    - Destructor of the object is called
    - The memory occupied by the object is returned to the system

Self-test: Accessibility

- Write a new method **`richerThan( )`** for the bank account:
  - Takes in another account as parameter
  - Compares the balance
    - Returns true if balance is more than the other account

```
class BankAcct {

private:
  //...same...

public:
  //...other methods not shown
  bool richerThan(BankAcct otherAcct)
  {
        //... Try it out ...

  }
};
```

Inheritance

Like father, like son

Inheritance: Motivation

- It is common to find several classes that share many attributes and methods
- For example, let's define a *saving account* class, which contains:
  - **Data** : *account number, balance, interest rate*
  - **Process**: *withdraw, deposit, pay_interest*
- It is clear that:
  - Saving account shares > 50% code compared to bank account
- Should we just cut and paste the code?

Inheritance: Motivation

- Duplicating code is undesirable:
  - Hard to maintain
    - Need to correct all copies if error is found
    - Need to update all copies if modification is needed
    - etc
- Also, since the classes are logically unrelated:
  - Other code that works on one class cannot work on the other
  - Example:

```
void transfer( BankAcct& fromAcct,
               BankAcct& toAcct, double amt );
```

will **not** work on saving account objects (compilation error)

Inheritance: Motivation

- Object oriented languages allow **inheritance**
  - Derive a new class from another class
  - The new class **inherits** most of the attributes and methods from the other class

- Terminology:
  - If **`class B`** is derived from **`class A`**, then
    - **`class B`** is called a **child (sub-class)** of **`class A`**
    - **`class A`** is called a **parent (super-class)** of **`class B`**

Saving Account: Inheritance example

```
class BankAcct {
protected:                  //changed from private
  int _acctNum;
  double _balance;
     ... ...
};

class SavingAcct : public BankAcct {
protected:
  double _rate;     //interest rate

public:
   SavingAcct( int anum, double rate )
      :BankAcct( anum )
   {
      _rate = rate;
   }
   void payInterest( )
   {
      _balance += _balance * _rate;
   }
};
```

To indicate inheritance

Note that there is **no** declaration for account number and balance

Special syntax for initializing base class or object member

Observations

- Use of inheritance greatly reduces the amount of redundant coding
  - No definition of account number and balance
  - No definition of withdraw and deposit

- Improve maintainability:
  - E.g. If the **`withdraw( )`** function is modified in **`class BankAcct`**, no changes is needed in **`class SavingAcct`**
  - The code on **`class BankAcct`** remains untouched
    - Other programs using **`BankAcct`** are not affected

Saving Accounts: Sample Usage

```
// BankAcct class and SavingAcct class definitions
void transfer(BankAcct& fromAcct,
              BankAcct& toAcct, double amt)
{
   fromAcct.withdraw(amt);
   toAcct.deposit(amt);
}

int main( )
{
    BankAcct ba1( 1234, 500.00 );
    SavingAcct sa1( 8888, 0.025 );

    sa1.deposit(1000.00);

    sa1.payInterest();

    transfer(ba1, sa1, 100.00);
}
```

A simple function for illustration purpose

Saving Account object

Inherited method

New method

Question: Will it work??

Super class and sub class

- An added advantage for inheritance is that:
  - Whenever a super class object is expected, a sub class object **is acceptable!**
  - E.g. the last line of the previous slide:
    - The function **`transfer()`** expects **`BankAcct`** object, but it is **ok** to pass in a **`SavingAcct`** object instead
  - Hence, all existing functions that work with the super class objects will work on sub class objects with **no modification**!
- Analogy:
  - We can drive a car
  - Honda is a car (Honda is a subclass of car)
  - We can drive a Honda

Pitfalls and Rules of thumb

- Beware:
  - Do not overuse inheritance
  - Do not overuse the **`protected`** keyword
    - Make sure it is something inherent for future sub class
- To determine whether it is correct to inherit:
  - Use the “is-a” rules of thumb
    - If “B is-a A” sounds right, then B is a subclass of A
  - Frequently confused with the “has-a” rule
    - If “B has-a A” sounds right, then B should have an A attribute

Rules of thumb: “is-a” and “has-a”

```
class BankAcct {
    ... ...
};

class SavingAcct : public BankAcct {
     ... ...

};
```

Inheritance: Saving Account IS-A Bank Acccount

```
class BankAcct {
    ... ...
};

class Person {

     BankAcct _customerAcct;
};
```

Attribute: Person HAS-A Bank Acccount

Templates

One definition, many data types

Generic functions and classes

- Generic processes:
  - Processes that are applicable to a wide range of data types
  - Example:
    - Choose the maximum element out of two elements
    - The solution is similar for integer, character, floating point or anything that can be compared using the “<” operator
- In C: Need to write separate versions of code
- In C++: **Template** can be used
  - Code is written **once** only
  - Data types can be specified later during actual usage
  - Any data types, including user defined types, can be used

Example: function **maximum( )**

- An implementation for integers:

```
int maximum( const int& left, const int& right)
{
      return (left > right)? left : right;
}
```

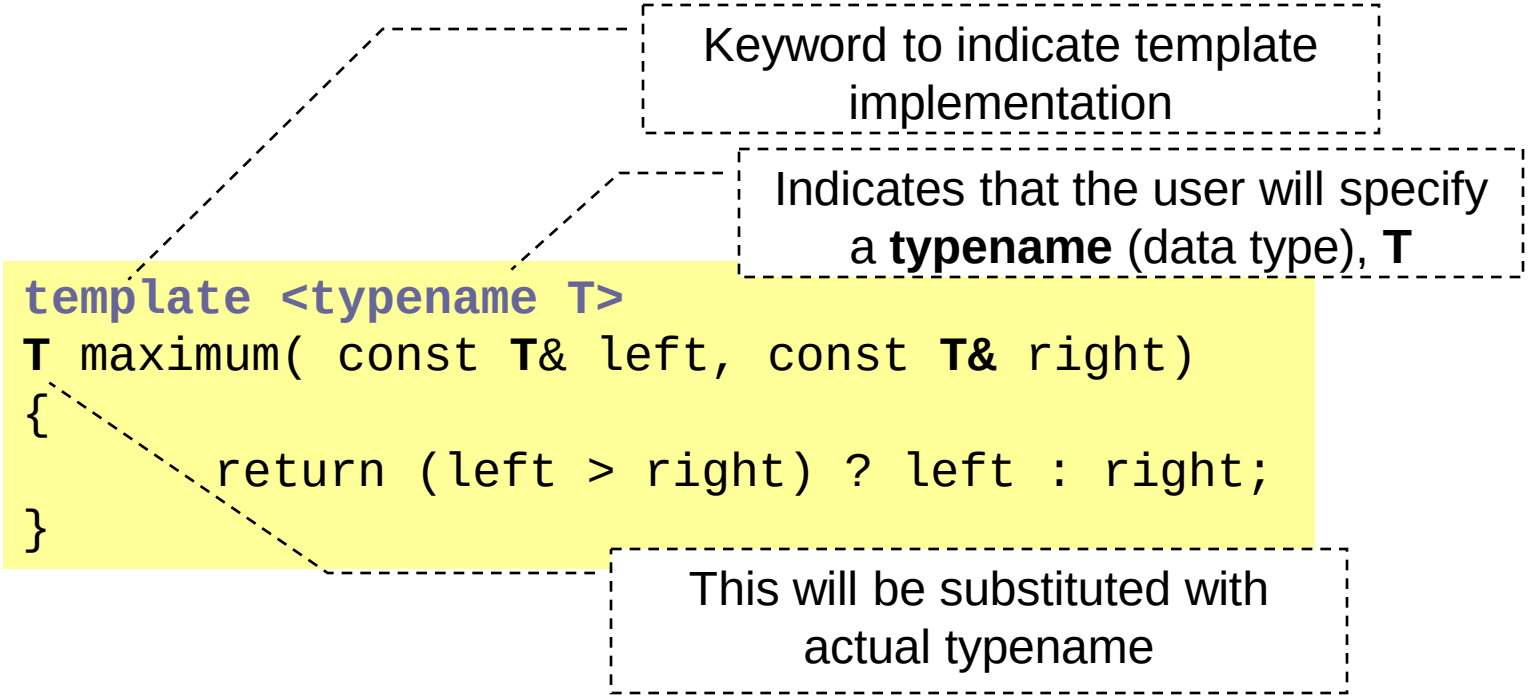
- The implementation above may not work correctly for other data types

```
float maxFloat;
maxFloat = maximum(3.14159, 0.1);
```

Error: What's the problem?

Template: function **maximum( )**

- Template implementation:

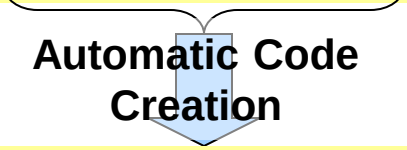


- **typename T** means that **T** is a variable that stores a data type (type name)
- Unlike **int T**, where **T** is a variable that stores **values** of integer type

Template: Usage example

- User can make use of the template by **explicitly** specifying the data type:

```
char maxChar = maximum<char> ('a', '!');
int maxInt = maximum<int> (6, -1);
float maxFloat = maximum<float>(3.1415, 0.1);
```




```
float maximum( const float& left, const float& right)
{
      return (left > right) ? left : right;
}
```

- Compiler **creates** actual code **automatically** with the type substituted **during compilation time** for template code
- Quick check:
  - How many versions of the maximum( ) function are there in the above example?

Template: Usage Example 2

- If the **typename** for the template can be deduced by the usage, the user can omit the explicit **typename**

```
char maxChar = maximum('a', '!');
int maxInt = maximum(6, -1);
float maxFloat = maximum(3.1415, 0.1);
```

Implicit typename

- The compiler can easily deduce the typename for each of the usages above
- Quick Check:
  - How many versions of the maximum( ) function are there in the above example?
  - How about the following?

```
float maxFloat = maximum(3.1415, 1234);
```

Example: A pair of integers

- The following class is useful for keeping track of pair of integers
  - E.g. 2D Coordinates (3, 5), (8,10) etc

```
class Pair {
private:
  int _first;                 // first value
  int _second;                // second value

public:
  Pair(int a, int b) : _first(a), _second(b) {}

  int getFirst() const { return _first; }
  int getSecond() const { return _second; }
};
```

- Could be useful for other data types as well
  - Pair of strings e.g. (“Potter”, “Harry”)
  - Pair of floating point number e.g. (1.34, -2.45), etc

Template: class Pair

```
template <typename T>
class Pair {
private:
  T _first;           // first value
  T _second;          // second value

public:
  Pair(T a, T b) : _first(a), _second(b) {}

  T getFirst() const { return _first; }
  T getSecond() const { return _second; }
};
```

- Similar to declaration of template function
  - Introduce a typename variable (e.g. T)
  - Use this typename variable for data that requires different datatype depends on usage

Template: Pair Usage Example

- For template class, type name declaration must be **explicit**

```
Pair<int> intPair(4, 6);

Pair<float> coordinate(1.23, -2.54);

Pair<char*> name("Harry", "Potter");
```

- Implicit type name is **invalid** for template class

```
Pair intPair(4, 6);                    Error!!

Pair coordinate(1.23, -2.54);          Error!!

Pair name("Harry", "Potter");          Error!!
```

Multiple Typename

- The class Pair can be even more general by allowing different types for its two elements
  - Example:
    - (“Harry Potter”, 18); //name and age
    - (“Optimus Prime”, BankAcct(9999, 1.50)) //name and bank account
    - etc

```
template <typename T1, typename T2>
class Pair {
private:
  T1 _first;        // first value
  T2 _second;             // second value

public:
  Pair(T1 a, T2 b) : _first(a), _second(b) {}

  T1 getFirst() const { return _first; }
  T2 getSecond() const { return _second; }
};
```

Useful Libraries in C++

Life and time savers…

Overview of C++ Standard Library

- String
- Stream
- Standard Template Library (STL)
  - Container
  - Iterator
  - Algorithm
- etc

String in C++

An object-oriented implementation of string

String in C++

- C++ provides an object oriented implementation for string
  - Header file: **`#include <string>`**
- Main features:
  - Operators:
    - "**=**" : Assign value to a string object
    - "**>**", "**<**", "**==**" : Comparison between string objects
    - "**+**" : Defined as string concatenation
    - "**`[index]`**" : Access character at position **`index`**
  - Constructors:
    - Default constructor: empty string
    - Cstring constructor: take in a c-style string
  - Methods:
    - **`size()`** or **`length()`**: gives number of characters in the string
    - **`substr(pos, nChar)`**: get the substring of **`nChar`** length starting from position **`pos`**
    - **`at(pos)`**: get the character at position **`pos`**

String in C++: Example 1

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

using namespace std;

int main()
{
  string str1;
  string str2(”xyz”);

  str1 = ”abc”;
  cout << ”S1 = ” << str1 << endl;
  cout << ”S2 = ” << str2 << endl;
  cout << ”S1 + S2 = ” << str1 + str2 << endl;
  cout << ”S2 + S1 = ” << str2 + str1 << endl;

  if (str1 > str2)
     cout << "S1 > S2" << endl;
  else
     cout << "S1 <= S2" << endl;
}
```

`str1` is an empty string

`str2` is initialized with the string “xyz”

Use “=“ to assign a string to **`str1`**

Output:

```
S1 = abc
S2 = xyz
S1 + S2 = abcxyz
S2 + S1 = xyzabc

S1 <= S2
```

String in C++: Example 2

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

using namespace std;

int main()
{
 char cstr[12] = "abcd";
 string str1(cstr);
 string str2("efgh");
 string str3;

 str3 = str1 + str2;

 cout << str3 << endl;
 cout << str3.size() << endl;
 cout << str3[4] << endl;
 cout << str3.at(4) << endl;
 cout << str3.substr(2,5) << endl;
}
```

Can use c-style string to initialize a string object

Addition returns a newly concatenated string

Output:

```
abcdefgh
8
e
e
cdefg
```

String in C++: Input/Output

- Use the insertion operator << to output string objects
- Use the extraction operator >> to input string objects
  - ignore the initial white spaces
  - return whatever read before the next white space
  - i.e. capable of reading a single word only
- To read a whole sentence:
  - return whatever read before the new line character

```
string str;
getline(cin, str);


char cstr[80];
cin.getline(cstr, 80);
```

Becareful of the syntax.

Size is not needed as string object are extendible

Provided for comparison purpose. Note the difference in getline( )

String in C++: Tokenizer

- Use the built in method

```
int find_first_of(const string& dstr, int pos = 0);
```

  - **dstr** : a string of possible delimiter characters
  - **pos** : start looking from position pos, with a default of 0
  - return the position if found; return **string::npos** if not found

```
string str = "One#Two$Three$";
unsigned int tStart = 0, tEnd = 0;

tEnd = str.find_first_of("#$");

while( tEnd != string::npos ) {
   cout << str.substr(tStart, tEnd - tStart) << endl;
   tStart = tEnd + 1;
   tEnd = str.find_first_of("#$", tStart);
}

if (tStart < str.size())
   cout << str.substr(tStart) << endl;
```

tStart = Starting position
tEnd = Ending position
str[tStart to tEnd] = token

pos not provided. Default of zero is used

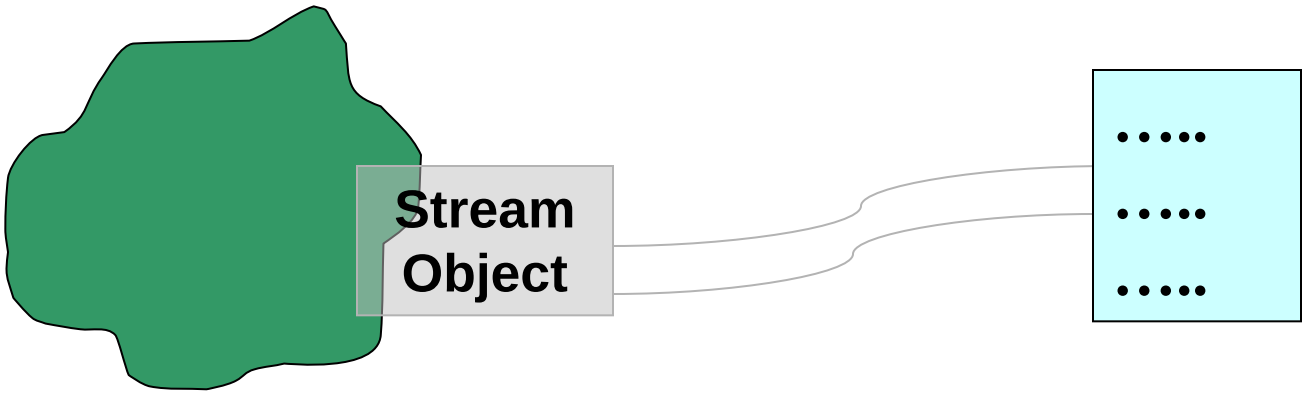
Start looking from this position.

Input/Output Stream

Managing input and output in C++

Standard C++ IO Stream

- C++ provides an standardized conceptual model to manage all kinds of input and output
- A ***stream object*** wraps around any actual device and provide facility for input/output



Executing Process
i.e. your program

Actual Devices
e.g. file, keyboard, monitor

Standard Stream: input stream

- **Header File:**
  - **`#include <iostream>`**
- **`istream cin`**
  - Built-in input stream variable
  - Default is the keyboard
  - Input using extractor operator **>>**
  - Converts input to the correct data type automatically

```
int x;
int y;
double z;

cin >> x >> y >> z;
```

Sample User Input:

```
10 10 3.45
```

Standard Stream: output stream

- **Header File:**
  - **`#include <iostream>`**
- **`ostream cout`**
  - Built-in output stream variable
  - Default is the console display
  - Output using inserter operator **<<**

```
int x = 20;
double y = 3.14;
string s = ”Hello”;


cout << x << ” ” << s << ” ” << y << endl;
```

Output:

```
20 3.14 Hello
```

Output Stream: Manipulator

- The behavior of the standard output stream can be modified using **IO Manipulator**
- Header File:

```
#include <iomanip>
```

- Usage:

```
cout << manipulator;
cout << data;
```

- Common manipulator:
  - **`endl`** : Flush the output
  - **`setprecision(n)`** : Set number of significant digits to **`n`**
  - **`setw(n)`** : Set field width to **`n`**
  - **`boolalpha`** : prints boolean value as "**`true`**"/"**`false`**"
  - Others can be found in Carrano's Book pg 827

Output Stream: Manipulator

```
double d = 3.141592;
bool b = true;

cout << d << endl;

cout << setprecision(3);
cout << d << endl;

cout << setw(10) << 1234 << endl;
cout << 1234 << endl;

cout << b << endl;
cout << boolalpha << b << endl;
```

Output:

```
3.14159

3.14

      1234
1234

1
true
```

- Note that most manipulators only affect the **next output**
  - E.g. see the **`setw(10)`** above
  - Need to set again for subsequent output

File Stream

Recording and Reading information

File Stream

- **Header File:**
  - **`#include <fstream>`**
- **`ifstream`**
  - Input file stream
- **`ofstream`**
  - Output file stream
- Opening a file stream:

```
ifstream inFile( ”input.txt” );
```

OR

```
ifstream inFile;
inFile.open( ”input.txt”);
```

  - Similar for `ofstream`
- Closing a file stream:

```
inFile.close( );
```

Similar for `ofstream`

File Stream: Example 1

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

int main () {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  int x;

  while (readFile >> x)
      writeFile << x  << "*";
  writeFile << endl;

  readFile.close();
  writeFile.close();
  return 0;
}
```

in.txt:

```
1 2 3
 4 5
6     7     8
```

out.txt:

```
1*2*3*4*5*6*7*8*
```

- Behavior of the >> extractor:
  - Skip all white spaces (blank, tab, newline)
  - Stop when:
    - Out of data
    - Data is of the wrong type

File Stream: Example 2

```
/... Similar to previous example .../

int main () {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  char x;

  while (readFile >> x)
      writeFile << x  << "*";
  writeFile << endl;

/... Similar to previous example .../
}
```

in.txt:

```
1 2 3
 4 5
6     7     8
```

out.txt:

```
1*2*3*4*5*6*7*8*
```

- The behavior of the >> extractor is the same even when reading for **characters** instead of **integers**
- To read every characters including white spaces
  - Use the **`get( )`** method

File Stream: Example 3

```
/... Similar to previous example .../

int main () {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  char x;

  while (readFile.get(x) )
      writeFile << x  << "*";
  writeFile << endl;

/... Similar to previous example .../
}
```

in.txt:

```
1 2 3
 4 5
6     7     8
```

out.txt:

```
1* *2* *3*
* *4* *5*
*6* * * *7* * * *8*
*
```

- Be careful when reading:
  - Make sure you use the correct operation is used
  - Most common problems in file reading

File Stream: Example 4

```
/... Similar to previous example .../

int main () {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  string x;

  while (readFile >> x)
      writeFile << x  << "*";
  writeFile << endl;

/... Similar to previous example .../
}
```

in.txt:

```
1 2 3
 4 5
6     7     8
```

out.txt:

```
1*2*3*4*5*6*7*8*
```

- As discussed before, extractor read only a **single word** for string object
- If whole sentence is needed:
  - Use **`getline( )`** method

File Stream: Example 5

```
/... Similar to previous example .../

int main () {
  ifstream readFile("in.txt") ;
  ofstream writeFile("out.txt");
  string x;

  while ( getline(readFile,x) )
      writeFile << x  << "*“ << endl;
  writeFile << endl;

/... Similar to previous example .../
}
```

in.txt:

```
1 2 3
 4 5
6     7     8
```

out.txt:

```
1 2 3*
 4 5*
6     7     8*
```

- Note that newline characters are **not** stored for string objects

File Stream: Example 6

```
/... Similar to previous example .../

int main () {
  ifstream readFile("test.txt") ;
  int i;
  string x;

  readFile >> i;
  getline(readFile,x);

  cout << "i: " << i << endl;
  cout << "x: " << x << endl;
/... Similar to previous example .../
}
```

test.txt:

```
1
4 5 6
```

output:

```
i: 1
x:
```

- Be careful when mixing **>>** and **`getline()`**
  - **>>** reads only required data, newline character is left untouched!
  - **`getline()`** picks up anything on a line, even if it is just a single new line character

File Stream: Example 6 (corrected)

```
/... Similar to previous example .../

int main () {
  ifstream readFile("test.txt") ;
  int i;
  string x;

  readFile >> i;
  getline(readFile, x);
  getline(readFile,x);

  cout << "i: " << i << endl;
  cout << "x: " << x << endl;
/... Similar to previous example .../
}
```

test.txt:

```
1
4 5 6
```

output:

```
i: 1
x: 4 5 6
```

- Simple remedy:
  - Use additional **`getline()`** to discard left over newline characters

cout and **cin**

- Reminder:
  - **cout** behaves similarly as an object of **ofstream**
  - **cin** behaves similarly as an object of **ifstream**
  - Examples in this section are applicable to **cout** and **cin** as well!
- E.g. Earlier code can be used for **cin** and **cout**:

```
int main () {
   string x;

   while (cin >> x)
       cout << x  << "*";
   cout << endl;

}
```

String Stream

Reading and writing to string

String Stream

- Stream objects have the nice ability to convert data to the required type automatically
- Is it possible to provide that functionality for string objects as well?
  - Convert string into other data type
    - Similar to C-Style string functions **`atoi()`**, **`atof( )`**
  - Convert other data type into string
    - Similar to C-Style string function **`sprintf( )`**
- C++ provides String Stream for the above functionalities

String Stream: Input String Stream

- **Header File:**
  - **`#include <sstream>`**
- **`istringstream`**
  - String stream for input
  - Can be constructed from raw string e.g. “Hello” or string object

```
istringstream instr(”Now 2.14 30”);

string s;
double d;
int x;

instr >> s;
instr >> d;
instr >> x;
```

s = ”Now”
d = 2.14
x = 30

String Input Stream: Example

- Useful when the input does not follow a pattern
- Example:
  - The input file stores coefficients of equations ax^2+bx+c 1 line per equation, it may look as follow:

    2 3 // linear equation 2x + 3

    4 5 6 // quadratic equation $4x^2 + 5x + 6$
  - If we use `inputFile >> i;` to get each number
    - This skips over any white spaces (including newline)
    - Hence, no way to distinguish this input with

      2 3 4 // quadratic equation $2x^2 + 3x + 4$

      5 6 // linear equation 5x + 6
- How to solve the problem?

String Input Stream: Example

- Basic Idea:
  - Read one line at a time
  - Make use of string input stream to read individual numbers

```
ifstream inputFile( ”coefficient.dat”);
string s;

while ( getline(inputFile,s) ) {
    istringstream is(s);
    int i = 0;
    int coef[3];
    while (is >> coef[i])
        ++i;

    Question: How to determine whether it’s
    quadratic or linear?
}
```

Read one line

Attach a string stream object to the input line

Get individual number from the string stream object

String Stream: Output String Stream

- **Header File:**
  - **`#include <sstream>`**
- **`ostringstream`**
  - String stream for output
  - Use << insertor to place information into a string output stream object
  - Use str( ) method to get a string object back from the string output stream object

```
string s = ”Now”;
double d = 3.14;
int x = 30;
ostringstream outstr;

outstr << s << ” ” << d << ” ” << x << endl;
string t = outstr.str();

cout << t;
```

Output:
```
Now 3.14 30
```

String Output Stream: Example

- Useful when we need to convert and pack several data into one string
  - The string can then be output to screen/file, or further processing
- Good practice:
  - Provide a **`toString()`** method for your own classes
  - Returns a string that contains useful information about the object
- Example (**`BankAcct Class`**):

```
class BankAcct {
...
public:
string toString( ) {
   ostringstream os;
       os << ”Acct No: ” << _acctNum;
       os << ” Balance: ” << _balance;
       return os.str();
}
...
};
```

String Output Stream: Example (cont)

- Sample Usage:
  - With the **`toString()`** method, we can conveniently prints the information to screen or to a file

```
#include "BankAcct.h"

int main( )
{   BankAcct ba1( 1234, 300.50 );
    BankAcct ba2( 9999, 1001.40 );
    string acctInfo;
    ofstream outFile("Account.txt");

    acctInfo = ba1.toString();
    cout << acctInfo << endl;
    outFile << acctInfo << endl;

    cout << ba2.toString() << endl;
    outFile << ba2.toString() << endl;
    ... ...
}
```

Get the information as a nice string

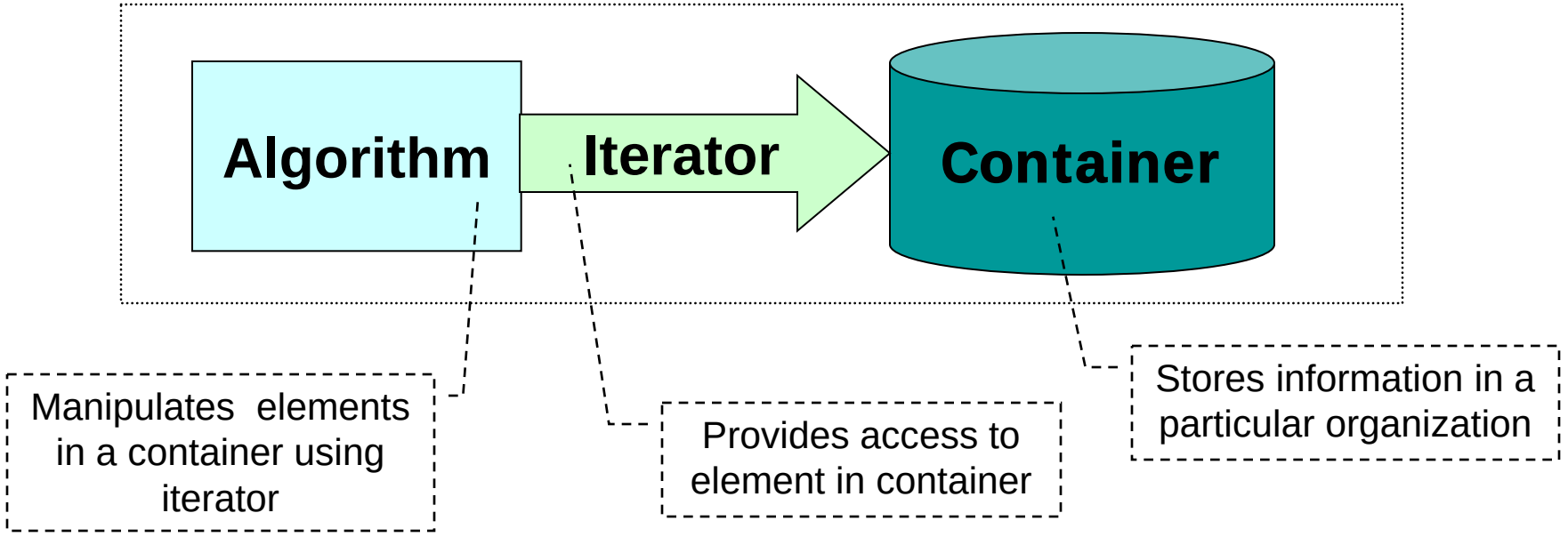
Can output to screen or file

Can make use of the method directly

Standard Template Library
STL

Standard Template Library (STL)

- A major part of the C++ standard library
- Consists of three components:
  - Container
  - Iterator
  - Algorithm
- Defined as template class
- Relationship between STL components:



STL Containers

- Containers:
  - Object that contains other objects
  - Represent general **data structures** in computing
  - Most of the data structures covered in this course are available as STL containers ☺
- Main features:
  - Template class
    - Can be used for built-in and user defined data types
  - All containers supports a set of general methods
    - size( ) : number of elements
    - empty() : is the container empty?
    - etc
  - Specialized methods are defined for individual container classes

STL Containers

- **Vectors**
- **Double-Ended Queues**
- **Lists**
- **Priority Queues**
- **Queues**
- **Stacks**
- Sets
- Multisets
- Maps
- Multimaps

STL Vector

- Header File:

```
#include <vector>
```

- Defined as template class

```
vector<int> intVector;
```

- Stores contiguous elements as an array
  - i.e. object oriented implementation of an array
- Advantages:
  - Fast insertion and removal of at the end of vector
  - Support dynamic number of elements
  - Automatic memory management
- Vector is the simplest STL container class, and in many cases the most efficient

STL Vector: Constructors

vector<T> *v* `vector<int> intV;`	*v* is an empty vector of type *T* **`intV`** is an empty integer vector
vector<T> *v(n)* `vector<int> intV( 10 );`	*v* is a vector of type *T* with *n* elements initialized with default constructor **`intV`** has 10 integers initialized to 0
vector<T> *v(n, tObj)* `vector<BankAcct> acctV(5, BankAcct(4444));`	*v* is a vector of type *T* with *n* copies of *tObj* **`acctV`** has 5 **`BankAcct`** objects, each with account number of 4444

STL Vector: Constructors (con't)

vector<T> *v(v2)*	*v* is an exact copy of vector *v2*. *v2* should have compatible type as *v*

`vector<int> intV;`
`...//add elements to intV`
`Vector<int> intV2( intV )`
	intV2 is an exact copy of **intV**

vector<T> *v(sIter,eIter)*	*v* is a vector of type *T* with elements copied from a container, starting at *sIter* iterator (inclusive) and stops at the *eIter* iterator (exclusive).
`int ia[6] = {1,2,3,4,5,6};` `vector<int> intV(ia,ia+3);`	**intV** has the first three elements copied from the array **ia**

STL Vector: Commonly used methods

`size( )`	returns the number of items
`empty( )`	true if the vector has no elements
`clear( )`	removes all elements
`at(`*`n`*`) or [`*`n`*`]`	returns an element at position *n*
`front()`	returns a reference to the first element
`back()`	returns a reference to the last element
`pop_back()`	removes the last element
`push_back( `*`e`*` )`	add element *e* to the end

- Other less frequently used methods can be found on online references

STL Vector: Commonly used methods

- Iterator related methods:

`begin()`	returns an iterator to the first element
`end()`	returns an iterator to the "end" of container
`erase( iter )`	removes element indicated by iterator *iter*
`insert(iter, e)`	inserts element *e* before the element indicated by iterator *iter*
`insert(iter, n, e)`	inserts *n* copies of element *e* before the element indicated by iterator *iter*

STL Vector: Example

```
#include <vector>
...
int main()
{

    vector<int> intV;

    cout << "intV size = " << intV.size() << endl;
    for (int ix = 0; ix != 5; ++ix)
          intV.push_back(ix);
    cout << "intV size = " << intV.size() << endl;

    if (!intV.empty()) {
        cout << "intV = [ ";
        for (int ix = 0; ix != intV.size(); ++ix)
            cout << intV[ix] << " ";
        cout << "]" << endl;
    }

    intV.pop_back();
    cout << "intV size = " << intV.size() << endl;
    ...
}
```

output:

```
intV size = 0
intV size = 5
intV = [ 0 1 2 3 4 ]
intV size = 4
```

STL Iterator

- Iterator is an abstraction:
  - Resembles a *pointer* that points into an *array*
- Iterator can be used to access and manipulate elements in a container:
  - Elements are accessed in a sequence regardless of actual organization
- Allows the programmer to define common operations (algorithm) for container without worrying about the underlying details
  - Some of these common operations are implemented as STL Algorithm

Operations on pointer

```
int a[] = {1,2,3,4,5,6,7,8,9};
int *p;

for (p = a; p != a+9; ++p) {
    cout << *p << endl;
}
```

A pointer can be

1. initialized to point to the beginning of the container (the array)
2. compared with another pointer to see whether it has come to the end of the container
3. incremented (++) to point to the next element in the container
4. dereferenced (*) to access the element in the container

Operations on Iterator

- Let **`iter`** be an iterator of a container

`*iter`	Accesses the item pointed by the iterator
`iter++` or **`++iter`**	Moves the iterator to point to the next item in the container
`iter--` or **`-- iter`**	Moves the iterator to point to the previous item in the container
`iter1 == iter2`	Returns true when both iterators point at the same item in the container
`iter1 != iter2`	Returns true when the two iterators do not point at the same item in the container

Iterator

- All container classes provide their own iterators
  - Declaration:

    ```
    container::iterator iterator_variable;
    ```

  - Example:

    ```
    vector<int>::iterator myIter;
    ```

- All container classes define following methods
  - **`begin()`** returns an iterator that points at the beginning of the container
  - **`end()`** returns an iterator that points at one element pass the end of the container
    - Usually used as a termination condition for loops

Iterator: Example

```
#include <vector>
...
void print_vector( vector<int>& iV )
{
    vector<int>::iterator iter;

    cout << "V = [ ";
    for (iter = iV.begin(); iter != iV.end(); ++iter)
            cout << *iter << " ";
    cout << "]" << endl;
}

int main()
{
    vector<int> intV;

    for (int ix = 0; ix != 5; ++ix)
          intV.push_back(ix);
    print_vector(intV);
}
```

output:

```
V = [ 0 1 2 3 4 ]
```

Iterator: Example (con't)

```
#include <vector>
...
void print_vector( vector<int>& iV ) {...}

int main()
{
    vector<int> intV;
    for (int ix = 0; ix != 5; ++ix)
          intV.push_back(ix);
    print_vector(intV);

    vector<int>::iterator myIter = intV.begin();

    intV.insert(myIter, 123); //caution! see next slide
    print_vector(intV);

    // Continue on next slide
```

output:

```
V = [ 0 1 2 3 4 ]
V = [ 123 0 1 2 3 4 ]
```

Iterator: Example (cont)

```
    //continue from previous slide …

    myIter = intV.begin();  //Important: reset myIter
    myIter++;
    intV.erase(myIter);
    print_vector(intV);

    myIter = intV.begin();   //Reset!
    cout << *myIter << endl;

}
```

output:

```
... ...
V = [ 123 1 2 3 4 ]
123
```

- Most built-in methods (e.g. `insert()`, `erase()`) invalidates the iterator after the operation:
  - Make sure you “reset” the iterator before the next usage!

Summary

- Class and Object
- Inheritance
- Generic Function and Class
- C++ Standard Library

References

- Carrano's Book (Data Abstraction and Problem Solving with C++)
  Appendix C: C++ Header Files and Standard Functions
  Appendix E: Standard Template Library
- C++ Online Reference
  http://www.cplusplus.com/reference/