

Introduction to C++

Leo Liberti

DIX, École Polytechnique, Palaiseau, France

II semester 2009/10

Preliminary remarks

Teachers

Leo Liberti: liberti@lix.polytechnique.fr.

Office: LIX 412-29 (prefab). Telephone: 4138

Jean-François BIASSE: biasse@lix.polytechnique.fr

Aim of the course

Teach the basics of the C++ language

Means

Mixed lecture/practical teaching style

Develop a complex C++ application

<http://www.lix.polytechnique.fr/~liberti/teaching/c++/dmap-08/>

Course structure

Timetable

Lectures: Mondays 14:00-15:45(36) and 16:00-17:45(36)

Examination

23/03/09 practical + oral

Course material

- Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading (MA), 1999
- Stephen Dewhurst, *C++ Gotchas: Avoiding common problems in coding and design*, Addison-Wesley, Reading (MA), 2002
- Herbert Schildt, *C/C++ Programmer's Reference*, 2nd edition, Osborne McGraw-Hill, Berkeley (CA)

Course contents

Syllabus

1 Preamble

- Course

2 Introduction

- Programming Languages
- C++ Language basics
- Syntax
- Basic Linux development tools

3 Classes

- Basic class semantics
- Input and output
- Inheritance and polymorphism

4 Templates

- User-defined templates
- Standard Template Library

Generalities

Definitions

- *Program*: set of instructions that can be interpreted by a computer
- *Instructions*: well-formed sequences of characters (syntax)
- *Interpretation*: sequence of operations performed by the computer hardware (semantics)
- *Programming language*: set of rules used to form valid instructions
- *Algorithm*: a program which terminates (though sometimes find “non-terminating algorithm” with abuse of notation)

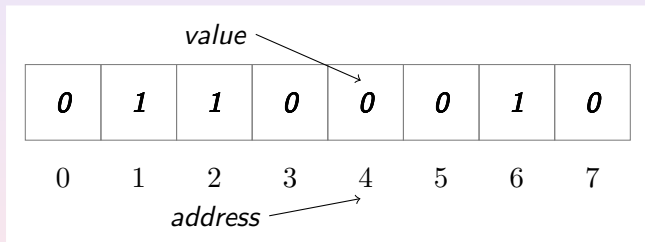
Operations

Valid computer operations

- *Input*: transfer data from external device to processor
- *Output*: transfer data from processor to external device
- *Storage*: transfer data from processor to memory
- *Retrieval*: transfer data from memory to processor
- *AL operation*: perform arithmetic/logical operation on data
- *Test*: verify condition on data and act accordingly
- *Loop*: repeat a sequence of operations

Memory

- Usual representation for memory: indexed array of cells where values can be stored



- unit of measure: **bit** (Binary digIT) — can hold a 0 or a 1
- 8b (bit) = 1B (byte), 1024 B = 1 KB (Kilobyte), 1024 KB = 1MB
- sometimes find 1KB = 1000 B and 1MB = 1000 KB

Creating and using data

- **Declaration:** the compiler is told about a new symbol and its type `void myFunction(void);`
- **Definition:** a segment of memory is associated to a symbol (called *variable name*) `char varName;`

varName →

1	1	0	1	0	1	1	0
0	1	2	3	4	5	6	7

- **Assignment:** a value is stored in the memory associated to the variable name

`char varName(9);`

varName →

0	0	0	0	0	1	0	1
0	1	2	3	4	5	6	7

`varName = 100;`

varName →

0	1	1	0	0	1	0	0
0	1	2	3	4	5	6	7

- **Deallocation:** the variable name is discarded and the associated memory is considered free

Basic data types

- boolean value: `bool` (1 bit), `true` or `false`
- ASCII character: `char` (1 byte), integer between -128 and 127
- integer number:
 - `int` (usually 4 bytes), between -2^{31} and $2^{31} - 1$
 - `long` (usually 8 bytes)
 - can be prefixed by `unsigned`
- floating point: `double` (also `float`, rarely used)
- arrays:

```
typeName variableName[constArraySize] ;
```

```
char myString[15];
```

- pointers (a pointer contains a memory address):

```
typeName * pointerName ;    char* stringPtr;
```

Declaration, Assignment, Test, AL Operators

- declaration: `typeName variableName ;` `int i;`
- assignment: `variableName = expression ;` `i = 0;`
- test:

```
if ( condition ) {
    statements ;
} else {
    statements ;
}
```

```
if ( i == 0 ) {
    i = 1;
} else if ( i < 0 ) {
    i = 0;
} else {
    i += 2;
}
```

- logical operators: and (&&), or (||), not (!)

```
condition1 logical_op condition2 ;
```

```
if (!(i == 0 || (i > 5 && i % 2 == 1))) { ...
```

- arithmetic operators: +, -, *, /, %, ++, --, +=, -=, *=, /=, ...

Loops

- loop (while):

```
while ( condition ) {
    statements ;
}
```

```
while (i < 10) {
    i = i + 1;
}
```

- loop (for):

```
for ( initial_statement ; condition ; itn_statement ) {
    statements ;
}
```

```
for (i = 0; i < 10; i++) {
    std::cout << "i = " << i << std::endl;
}
```

Functions

- function declaration:

```
typeName functionName(typeName1 argName1, ...) ;  
  
double performSum(double op1, double op2);
```

- function call:

```
varName = functionName(argName1, ...) ;  
  
double d = performSum(1.0, 2.1);
```

- return control to calling code: *return value ;*

```
double performSum(double op1, double op2) {  
    return op1 + op2;  
}
```

Functions: Argument passing

- Arguments are passed from the calling function to the called function in two possible ways:
 - by *value*
 - by *reference*
- Passing by value (default): the calling function makes a copy of the argument and passes the copy to the called function; **the called function cannot change the argument**

```
double performSum(double op1, double op2);
```

- Passing by reference (prepend a &): the calling function passes the argument directly to the called function; **the called function can change the argument**

```
void increaseArgument(double& arg) { arg++; }
```

Functions: Overloading

- Different functions with the same name but different arguments: *overloading*
- Often used when different algorithms exist to obtain the same aim with different data types

```
void getInput(int theInput) {  
    std::cout << "an integer" << std::endl;  
}  
void getInput(std::string theInput) {  
    std::cout << "a string" << std::endl;  
}
```

- Can be used in recursive algorithms to differentiate initialization and recursive step

```
void retrieveData(std::string URL, int maxDepth, Digraph& G,  
                bool localOnly, bool verbose);  
void retrieveData(std::string URL, int maxDepth, Digraph& G,  
                bool localOnly, bool verbose,  
                int currentDepth, VertexURL* vParent);
```

Pointers

- retrieve the address of a variable:

```
pointerName = &variableName ;
```

```
int* pi;  
pi = &i;
```

- retrieve the value stored at an address:

```
variableName = *pointerName ;
```

```
int j;  
j = *i;
```

- using pointers as arrays:

```
const int bufferSize = 10;  
char buffer[bufferSize] = "J. Smith";  
char* bufPtr = buffer;  
while(*bufPtr != ' '){  
    bufPtr++;  
}  
std::cout << ++bufPtr << std::endl;
```

Pointer warnings

- Pointers allow you to access memory directly, hence can be very dangerous
- Attempted memory corruption results in “segmentation fault” error and abort, or garbage output, or unpredictable behaviour
- Most common dangers:

- ① writing to memory outside bounds

```
char buffer[] = "LeoLiberti";  
char* bufPtr = buffer;  
while(*bufPtr != ' '){  
    *bufPtr = ' ';  
    bufPtr++;  
}
```

- ② deallocating memory more than once
- Pointer bugs are usually very hard to track

Indentation

- Not necessary for the computer
- **Absolutely necessary for the programmer / maintainer**
- After each opening brace { : new line and tab (2 characters)
- Each closing brace } is on a new line and “untabbed”

```
double x, y, z, epsilon;  
...  
if (fabs(x) < epsilon) {  
    if (fabs(y) < epsilon) {  
        if (fabs(z) < epsilon) {  
            for(int i = 0; i < n; i++) {  
                x *= y*z;  
            }  
        }  
    }  
}
```

Comments

- Not necessary for the computer
- **Absolutely necessary for the programmer / maintainer**
- One-line comments: introduced by //
- Multi-line comments: /* ... */
- Avoid over- and under-commentation
- Example of over-commentation

```
// assign 0 to x  
double x = 0;
```

- Example of under-commentation

```
char buffer[] = "01011010 01100100";  
char* bufPtr = buffer;  
while(*bufPtr &&  
      (*bufPtr++ = *bufPtr == '0' ? 'F' : 'T'));
```

Structure of a C++ Program I

```
/******  
* Name:          helloworld.cxx  
* Author:        Leo Liberti  
* Source:        GNU C++  
* Purpose:       hello world program  
* Build:         c++ -o helloworld helloworld.cxx  
* History:       060818 work started  
*****/  
  
#include<iostream>  
  
int main(int argc, char** argv) {  
    using namespace std;  
    cout << "Hello World" << endl;  
    return 0;  
}
```

Structure of a C++ Program II

- Each executable program coded in C++ must have one function called **main()**

```
int main(int argc, char** argv);
```

- The main function is the entry point for the program
- It returns an integer *exit code* which can be read by the shell
- The integer `argc` contains the number of arguments on the command line
- The array of character arrays `**argv` contains the arguments: the command `./mycode arg1 arg2` gives rise to the following storage:

`argv[0]` is a char pointer to the string `./mycode`

`argv[1]` is a char pointer to the string `arg1`

`argv[2]` is a char pointer to the string `arg2`

`argc` is an int variable containing the value 3

Structure of a C++ Program III

- C++ programs are stored in one or more text files
- Source files: contain the C++ code, extension `.cxx`
- Header files: contain the declarations which may be common to more source files, extension `.h`
- Source files are compiled
- Header files are included from the source files using the *preprocessor directive* `#include`

```
#include<standardIncludeHeader>
```

```
#include "userDefinedIncludeFile.h"
```

Development stages

- Creating a directory for your project(s) `mkdir directoryName`
- Entering the directory `cd directoryName`
- Creating/editing the C++ program
- Building the source
- Debugging the program/project
- Packaging/distribution (Makefiles, READMEs, documentation...)

Basic UNIX tools

- `cd directoryName` : change working directory
- `pwd` : print the working directory
- `cat fileName` : display the (text) file *fileName* to standard output
- `mv file position` : move *file* to a new *position*: e.g. `mv /etc/hosts .` moves the file `hosts` from the directory `/etc` to the current working directory (`.`)
- `cp file position` : same as `mv`, but copy the file
- `rm file` : remove *file*
- `rmdir directory` : remove an empty *directory*
- `grep string file(s)` : look for a *string* in a set of *files*: e.g. `grep -Hi complex *` looks in all files in the current directory (`*`) for the string `complex` ignoring upper/lower case (`-i`) and displays the name of the file (`-H`) as well as the line where the match occurs

Combining UNIX tools

- By default, unix tools send their output messages to the *standard output* stream (stdout) and their error messages to the *standard error* stream (stderr)
- Both streams can be redirected. E.g., to redirect both stdout and stderr, use:

```
sh -c 'command options arguments > outFile 2>&1'
```

- The output stream of a command can become the input stream of the next command in a chain:
e.g. `find ~ | grep \.cxx` finds all files with extension `.cxx` in all subdirectories of the home directory; the first command (`find`) sends a recursive list across subdirectories of the home directory (denoted by `~`) to stdout. This stream is transformed by the pipe character (`|`) in the standard input (stdin) stream of the following command (`grep`), which filters out all lines *not* containing `.cxx`.

Editing

- Traditional GNU/Linux text editor: `emacs`

`emacs programName.cxx`

- Many key-combination commands (try ignoring menus!)
- Legenda: C-key: CTRL+key, M-key: ALT+key (for keyboards with no ALT key or for remote connections can obtain same effect by pressing and releasing ESC and then key)
- Basics:
 - 1 C-x C-s: save file in current buffer (screen) with current name (will ask for one if none is supplied)
 - 2 C-x C-c: exit (will ask for confirmation for unsaved files)
 - 3 C-space: start selecting text (selection ends at cursor position)
 - 4 C-w: cut, M-w: copy, C-y: paste
 - 5 tab: indents C/C++ code
 - 6 M-x indent-region: properly indents all selected region

Building

- Computer needs instructions in machine language (executable), C++ is text
- The translation process from C++ to executable is called *building*
- Two stages:
 - ① *compilation*: production of an intermediate object file (.o) possibly with unresolved symbols
 - ② *linking*: resolve symbols by reading code from other objects, standard, and user-defined libraries
- Can perform both in one go with the GNU command `c++:`
`c++ -o helloworld helloworld.cxx`
- Can perform separately: `c++ -c helloworld.cxx`
(produces `helloworld.o`),
`c++ -o helloworld helloworld.o` (useful for combining multiple object files into one executable)

File types

- C++ *declarations* are stored in text files with extension `.h` (*header files*)
- C++ source code is stored in text files with extension `.cxx`
- Executable files have no extensions but their “executable” property is set to on (e.g. `ls -la /bin/bash` returns 'x' in the properties field)
- Each executable must have exactly *one* symbol `main` corresponding to the first function to be executed
- An executable can be obtained by *compiling* many source code files (`.cxx`), *exactly one of which* contains the definition of the function `int main(int argc, char** argv);`, and linking all the objects together
- Source code files are compiled into object files with extension `.o` by the command `c++ -c sourceCode.cxx`

Objects and symbols

- An object file (.o) contains a table of symbols used in the corresponding source file (.cxx)
- The symbols whose definition was given in the corresponding source file are *resolved*
- The symbols whose definition is found in another source file are *unresolved*
- Unresolved symbols in an object file can be resolved by *linking* the object with another object file containing the missing definitions
- An executable cannot contain any unresolved symbol
- A group of object files file1.o, ..., fileN.o can be linked together as a single executable file by the command `c++ -o file file1.o ... fileN.o` only if:
 - ① the symbol `main` is resolved exactly once in exactly one object file in the group
 - ② for each object file in the group and for each unresolved symbol in the object file, the symbol must be resolved in exactly one other file of the group

Debugging

- GNU/Linux debugger: `gdb`
- Graphical front-end: `ddd`
- Designed for Fortran/C, not C++
- Can debug C++ programs but has troubles on complex objects (use the “insert a print statement” technique when `gdb` fails)
- Memory debugger: `valgrind` (to track pointer bugs)
- In order to debug, compile with `-g` flag:

```
c++ -g -o helloworld helloworld.cxx
```
- More details during labs

Packaging and Distribution

- For big projects with many source files, a `Makefile` (detailing how to build the source) is essential
- Documentation for a program is **absolutely necessary** for both users and maintainers
- Better insert a minimum of help within the program itself (to be displayed on screen with a particular option, like `-h`)
- A `README` file to briefly introduce the software is usual
- There exist tools to embed the documentation within the source code itself and to produce `Makefiles` more or less automatically
- UNIX packages are usually distributed in tarred, compressed format (extension `.tar.gz` obtained with the command
`tar zcvf directoryName.tar.gz directoryName`

Classes: motivations

- 1 Problem analysis is based on data and algorithm break-down structuring \Rightarrow hierarchical design for data and algorithms
- 2 Fewer bugs if data inter-dependency is low \Rightarrow design data structure first, then associate algorithms to data (not the reverse)
- 3 Data structures are usually complex entities \Rightarrow need for sufficiently rich expressive powers for data design
- 4 Different data objects may share some properties \Rightarrow exploit this fact in hierarchical design

The class concept

Class

A *class* is a user-defined data type. It contains some data *fields* and the *methods* (i.e. algorithms) acting on them.

```
class TimeStamp {  
    public: // can be accessed from outside  
        TimeStamp(); // constructor  
        ~TimeStamp(); // destructor  
        long get(void) const; // some methods  
        void set(long theTimeStamp);  
        void update(void);  
    private: // can only be accessed from inside  
        long timestamp; // a piece of data  
};
```


Objects of a class

- An *object* is a piece of data having a class data type
- A class is declared, an object is defined
- In a program there can only be one class with a given name, but several objects of the same class
- Example:

```
TimeStamp theTimeStamp; // declare an object  
theTimeStamp.update(); // call some methods  
long theTime = theTimeStamp.get();  
std::cout << theTime << std::endl;
```

Referring to the current object

- Occasionally, we may want to know the address of an object within one of its methods
- Each object is endowed with the `this` pointer

```
cout << this << endl;
```

Constructors and destructors

- The class constructor defines the data fields and performs all user-defined initialization actions necessary to the object
- The class constructor is called only once when the object is defined
- The class destructor performs all user-defined actions necessary to object destruction
- The class destructor is called only once when the object is destroyed
- An object is destroyed when its *scope* ends (i.e. at the first brace `}` closing its level)

Lifetime of an object I

```
int main(int argc, char** argv) {
    using namespace std;
    TimeStamp theTimeStamp; // object created here
    theTimeStamp.update();
    long theTime = theTimeStamp.get();
    if (theTime > 0) {
        cout << "seconds from 1/1/1970: "
              << theTime << endl;
    }
    return 0;
} // object destroyed before brace (scope end)
```

Lifetime of an object II

Constructor and destructor code:

```
TimeStamp::TimeStamp() {  
    std::cout << "TimeStamp object constructed at address "  
        << this << std::endl;  
}  
TimeStamp::~TimeStamp() {  
    std::cout << "TimeStamp object at address "  
        << this << " destroyed" << std::endl;  
}
```

Output:

```
TimeStamp object constructed at address 0xbffff24c  
seconds from 1/1/1970: 1157281160  
TimeStamp object at address 0xbffff24c destroyed
```

Data access privileges

```
class ClassName {
    public:
        members with no access restriction
    protected:
        access by: this, derived classes, friends
    private:
        access by: this, friends
};
```

- a *derived* class is a class which inherits from this (see inheritance below)
- a function can be declared *friend* of a class to be able to access its protected and private data

```
class TheClass {
    ...
    friend void theFriendMethod(void);
};
```

Namespaces

- All C++ symbols (variable names, function names, class names) exist within a *namespace*
- The complete symbol is `namespaceName::symbolName`
- The only pre-defined namespace is the *global namespace* (its name is the empty string `::varName`)
- Standard C++ library: namespace `std` `std::string`

```
namespace WET {  
    const int maxBufSize = 1024;  
    const char charCloseTag = '>';  
}
```

```
char buffer[WET::maxBufSize];
```

```
using namespace WET;  
for(int i = 0; i < maxBufSize - 1; i++) {  
    buffer[i] = charCloseTag;  
}
```

Exceptions I

- Upon failure, a method may abort its execution
- We do not wish the whole program to abort
- Mechanism:
 - ❶ method *throws* an *exception*
 - ❷ caller method *catches* it
 - ❸ called method handles it if it can
 - ❹ otherwise it re-throws the exception
- Exceptions are passed on the method calling hierarchy levels until one of the method can handle it
- If exceptions reaches `main()`, the program is aborted

Exceptions II

Definition

An *exception* is a class. Exceptions can be thrown and caught by methods. If a method throws an exception, it must be declared:

returnType methodName(arguments) throw (ExceptionName)

- The `TimeStamp::update()` method obtains the current time through the operating system, which is outside the program's control
- `update()` does not know how to deal with a failure directly, as it can only update the time; should failure occur, control is delegated to higher-level methods

```
class TimeStampException {
public:
    TimeStampException();
    ~TimeStampException();
}
```

Exceptions III

```
void TimeStamp::update(void) throw (TimeStampException) {
    using namespace std;
    struct timeval tv;
    struct timezone tz;
    try {
        int retVal = gettimeofday(&tv, &tz);
        if (retVal == -1) {
            cerr << "TimeStamp::updateTimeStamp(): "
                 << "could not get system time" << endl;
            throw TimeStampException();
        }
    } catch (...) {
        cerr << "TimeStamp::updateTimeStamp(): "
             << "could not get system time" << endl;
        throw TimeStampException();
    }
    timestamp = tv.tv_sec;
}
```

Overloading operators in and out of classes I

- Suppose you have a class `Complex` with two pieces of private data, `double real;` and `double imag;`
- You wish to overload the `+` operator so that it works on objects of type `Complex`
- There are two ways: (a) declare the operator outside the class as a friend of the `Complex` class; (b) declare the operator to be a member of the `Complex` class

Overloading operators in and out of classes II

- (a) declaration:

```
class Complex {  
    public:  
        Complex(double re, double im) : real(re), imag(im) {}  
        ...  
        friend Complex operator+(Complex& a, Complex& b);  
    private:  
        double real;  
        double imag;  
}
```

definition (out of the class):

```
Complex operator+(Complex& a, Complex& b) {  
    Complex ret(a.real + b.real, a.imag + b.imag);  
    return ret;  
}
```

Overloading operators in and out of classes III

- (b) declaration:

```
class Complex {
public:
    Complex(double re, double im) : real(re), imag(im) {}
    ...
    Complex operator+(Complex& b);
private:
    double real;
    double imag;
}
```

definition (in the class):

```
Complex Complex::operator+(Complex& b) {
    Complex ret(this->real + b.real, this->imag + b.imag);
    return ret;
}
```

- `this->` is not strictly required, but it makes it clear that *the left operand is now the object calling the operator+ method*

The stack and the heap

- Executable program can either refer to near memory (the *stack*) or far memory (the *heap*)
- Accessing the stack is **faster** than accessing the heap
- The stack is **smaller** than the heap
- Variables are allocated on the stack `TimeStamp tts;`
- Common bug (but hard to trace): **stack overflow**
`char veryLongArray[1000000000];`
- Memory allocated on the stack is deallocated automatically at the end of the scope where it was allocated (closing brace `}`)
- Memory on the heap can be accessed through *user-defined memory allocation*
- Memory on the heap must be deallocated explicitly, otherwise *memory leaks* occur, exhausting all the computer's memory
- Memory on the heap **must not be deallocated more than once** (causes unpredictable behaviour)

User-defined memory allocation

- Operator new: allocate memory from the heap

```
pointerType* pointerName = new pointerType ;
```

```
TimeStamp* ttsPtr = new TimeStamp;
```

- Operator delete: release allocated memory

```
delete pointerName; delete ttsPtr;
```

- Commonly used with arrays in a similar way:

```
pointerType* pointerName = new pointerType [size] ;
```

```
double* positionVector = new double [3];
```

```
delete [] pointerName ; delete [] positionVector;
```

- **Improper user memory management causes the most difficult C++ bugs!!**

Using object pointers

- Suppose `ttsPtr` is a pointer to a `TimeStamp` object
- Two equivalent ways to call its methods:
 - 1 `(*ttsPtr).update();`
 - 2 `ttsPtr->update();`
- Prefer second way over first

Streams

- Data “run” through *streams*
- Stream types: input, output, input/output, standard, file, string, user-defined

```
outputStreamName << varName or literal ... ;
```

```
std::cout << "i = " << i << std::endl;
```

```
inputStreamName >> varName ;    std::cin >> i;
```

```
stringstream buffer;  
char myFileName[] = "config.txt";  
ifstream inputFileStream(myFileName);  
char nextChar;  
while(inputFileStream && !inputFileStream.eof()) {  
    inputFileStream.get(nextChar);  
    buffer << nextChar;  
}  
cout << buffer.str();
```

Object onto streams

- Complex objects may have a complex output procedure

- **Example:** we want to be able to say

```
cout << theTimeStamp << endl;
```

 and get

Thu Sep 7 12:23:11 2006 as output

- Solution: overload the << operator

```
std::ostream& operator<<(std::ostream& s, TimeStamp& t)
    throw (TimeStampException);
```

Object onto streams II

```
#include <ctime>

std::ostream& operator<<(std::ostream& s, TimeStamp& t)
    throw (TimeStampException) {
    using namespace std;
    time_t theTime = (time_t) t.get();
    char* buffer;
    try {
        buffer = ctime(&theTime);
    } catch (...) {
        cerr << "TimeStamp::updateTimeStamp(): "
              "couldn't print system time" << endl;
        throw TimeStampException();
    }
    buffer[strlen(buffer) - 1] = '\\0';
    s << buffer;
    return s;
}
```

Overloading the << and >> operators I

- How does an instruction like

```
cout << "time is " << theTimeStamp << endl; work?
```

- Can parenthesize is as

```
((cout << "time is ") << theTimeStamp) << endl;
```

to make it clearer

- Each << operator is a binary operator whose left operand is an object of type ostream (like the cout object); we need to define an operator overloading for each new type that the right operand can take
- Luckily, many overloadings are already defined in the Standard Template Library

Overloading the << and >> operators II

- The declaration to overload is:

```
std::ostream& operator<<(std::ostream& outStream,  
                        newType& newObject)
```

- To output objects of type TimeStamp, use:

```
std::ostream& operator<<(std::ostream& outStream,  
                        TimeStamp& theTimeStamp)
```

- Note: in order for the chain of << operators to output all their data to the same ostream object, each operator must return the same object given at the beginning of the chain (in this case, cout)
- In other words, each overloading must end with the statement `return outStream;` (notice `outStream` is the *very same name* of the input parameter — so if the input parameter was, say, `cout`, then that's what's being returned by the overloading)

Inheritance

- Consider a class called `FileParser` which is equipped with methods for parsing text occurrences like `tag = value` in text files
- We now want a class `HTMLPage` representing an HTML page with all links
- `HTMLPage` will need to parse an HTML (text) file to find links; these are found by looking at occurrences like `HREF="url"`
- It is best to keep the text file parsing data/methods and HTML-specific parts independent
- `HTMLPage` can *inherit* the public data/methods from `FileParser`:

```
class HTMLPage : public FileParser {...} ;
```

Nested inheritance

- Consider a corporate personnel database
- Need `class Employee;`
- Certain employees are “empowered” (have more responsibilities): need `class Empowered : public Employee;`
- Among the empowered employees, some are managers: need `class Manager : public Empowered;`
- Manager contains public data and methods from Empowered, which contains public data and methods from Employee

Nested inheritance II

```
class Employee {
public:
    Employee();
    ~Employee();
    double getMonthlySalary(void);
    void getEmployeeType(void);
};
```



```
class Empowered : public Employee {
public:
    Empowered();
    ~Empowered();
    bool isOverworked(void);
    void getEmployeeType(void);
};
```



```
class Manager : public Empowered {
public:
    Manager();
    ~Manager();
    bool isIncompetent(void);
    void getEmployeeType(void);
};
```


Hiding

Consider method `getEmployeeType`: can be defined in different ways for `Manager`, `Empowered`, `Employee`: *hiding*

```
void Employee::getEmployeeType(void) {  
    std::cout << "Employee" << std::endl;  
}  
void Empowered::getEmployeeType(void) {  
    std::cout << "Empowered" << std::endl;  
}  
void Manager::getEmployeeType(void) {  
    std::cout << "Manager" << std::endl;  
}
```

Nested inheritance and hiding

Examples of usage

```
Employee e1;
Empowered e2;
Manager e3;
cout << e1.getMonthlySalary(); // output the monthly salary
cout << e2.getMonthlySalary(); // call to the same fn as above
e1.getEmployeeType(); // output: Employee
e2.getEmployeeType(); // output: Empowered (call to different fn)
e3.getEmployeeType(); // output: Manager (call to different fn)
e3.Employee::getEmployeeType(); // output: Employee (forced call)
cout << e1.isIncompetent(); // ERROR, not in base class
```

Inheritance vs. embedding

- Consider example of a salary object:

```
class Salary {  
    Salary();  
    ~Salary();  
    void raise(double newSalary);  
    ...  
};
```

- Might think of deriving Employee from Salary so that we can say `theEmployee.raise()`; to raise the employee's salary
- Technically, nothing wrong
- Architecturally, very bad decision!
- Rule of thumb:
derive B from A only if B can be considered as an A
- In this case, better embed a Salary object as a data field of the Employee class

Polymorphism I

- Hiding provides *compile-time polymorphism*
- Almost always, this is **not** what is desired, and should be avoided!
- Want to be able to choose the class type of an object *at run-time*
- Suppose we want to write a function such as:

```
void use(Employee* e) {  
    e->getEmployeeType();  
}
```

and then call it using Employee, Empowered, Manager objects:

```
use(&e1); // output: Employee  
use(&e2); // output: Employee  
use(&e3); // output: Employee
```

- As far as use() is concerned, the pointers are all of Employee type, so wrong method is called

Polymorphism II

Run-time polymorphism can be obtained by declaring the relevant methods as virtual

```
class Employee {  
    ...  
    virtual void getEmployeeType(void);  
    ...  
};
```

```
class Empowered : public Employee {  
    ...  
    virtual void getEmployeeType(void);  
    ...  
};
```

```
class Manager : public Empowered {  
    ...  
    virtual void getEmployeeType(void);  
    ...  
};
```

```
use(&e1); // output: Employee  
use(&e2); // output: Empowered  
use(&e3); // output: Manager
```

Pure virtual classes

- Get objects to interact with each other: need *conformance* to a set of mutually agreed methods
- In other words, need an *interface*
- All classes derived from the interface implement the interface methods as declared in the interface
- Can guarantee the formal behaviour of all derived objects
- In C++, an interface is known as a *pure virtual class*: a class consisting only of method declarations and no data fields
- A pure virtual class has no constructor — no object of that class can ever be created (only objects of derived classes)
- A pure virtual class may have a virtual destructor to permit correct destruction of derived objects
- All methods (except the destructor) are declared as follows:

```
returnType methodName(args) = 0;
```
- All derived classes **must** implement all methods

Pure virtual classes

```
class EmployeeInterface {  
    public:  
        virtual ~EmployeeInterface() { }  
        virtual void getEmployeeType(void) = 0;  
};
```

```
class Employee : public virtual EmployeeInterface {...};  
class Empowered : public Employee, public virtual EmployeeInterface {...};  
class Manager : public Empowered, public virtual EmployeeInterface {...};
```

```
void use(EmployeeInterface* e) {...}  
...  
use(&e1); // output: Employee  
use(&e2); // output: Empowered  
use(&e3); // output: Manager
```

- Code behaves as before, but clearer architecture
- public virtual inheritance: avoids having many copies of EmployeeInterface in Empowered and Manager

Templates I

- *Situation*: action performed on different data types
- *Possible solution*: write many functions taking arguments of many possible data types.
- *Example*: swapping the values of two variables

```
void varSwap(int& a, int& b);  
void varSwap(double& a, double& b);  
...
```

- Potentially an unlimited number of objects \Rightarrow invalid approach
- Need for *templates*

```
template<class TheClassName> returnType functionName(args);
```

```
template<class T> void varSwap(T& a, T& b) {  
    T tmp(b);  
    b = a;  
    a = tmp;  
}
```


Templates II

Behaviour with predefined types:

```
int ia = 1;  
int ib = 2;  
varSwap(ia, ib);  
cout << ia << ", " << ib << endl; // output:  2, 1
```

```
double da = 1.1;  
double db = 2.2;  
varSwap(da, db);  
cout << da << ", " << db << endl; // output:  2.2, 1.1
```

Templates III

Behaviour with user-defined types:

```
class MyClass {
public:
    MyClass(std::string t) : myString(t) { }
    ~MyClass() { }
    std::string getString(void) { return myString; }
    void setString(std::string& t) { myString = t; }
private:
    std::string myString;
};
```

```
MyClass ma("A");
MyClass mb("B");
varSwap(ma, mb);
cout << ma << ", " << mb << endl; // output:  B, A
```

Internals and warnings

- Many hidden overloaded functions are created **at compile-time** (one for each argument list that is actually used)
- Very difficult to use debugging techniques such as breakpoints (which of the hidden overloaded functions should get the breakpoints?)
- **Use sparingly**
- But use the Standard Template Library as much as possible (already well debugged and very efficient!)

The STL

- Collection of generic classes and algorithms
- Born at the same time as C++
- Well defined
- Very flexible
- Reasonably efficient
- Use it as much as possible, do not reinvent the wheel!
- Documentation: <http://www.sgi.com/tech/stl/>
- Contains:
 - Classes: vector, map, string, I/O streams, ...
 - Algorithms: sort, swap, copy, count, ...

vector example

```
#include<vector>
#include<algorithm>
...
using namespace std;
vector<int> theVector;
theVector.push_back(3);
theVector.push_back(0);
if (theVector.size() >= 2) {
    cout << theVector[1] << endl;
}
for(vector<int>::iterator vi = theVector.begin();
    vi != theVector.end(); vi++) {
    cout << *vi << endl;
}
sort(theVector.begin(), theVector.end());
for(vector<int>::iterator vi = theVector.begin();
    vi != theVector.end(); vi++) {
    cout << *vi << endl;
}
```

map example

```
#include<map>
#include<string>
...
using namespace std;
map<string, int> phoneBook;
phoneBook["Liberti"] = 3412;
phoneBook["Baptiste"] = 3800;
for(map<string,int>::iterator mi = phoneBook.begin();
    mi != phoneBook.end(); mi++) {
    cout << mi->first << ": " << mi->second << endl;
}
cout << phoneBook["Liberti"] << endl;
cout << phoneBook["Smith"] << endl;
for(map<string,int>::iterator mi = phoneBook.begin();
    mi != phoneBook.end(); mi++) {
    cout << mi->first << ": " << mi->second << endl;
}
```