C++

Patrick Marais Simon Perkins

Computer Science Department
University of Cape Town

February 5, 2024

Outline

C++ History

- Developed by Bjarne Stroustrop around 1979.
- “C With Classes”.
- Goal: Object Orientation to C without sacrificing performance, portability or **low-level functionality**.
- OO component refined over two decades, adding virtual functions, operator overloading, multiple inheritance, and exception handling.
- **Generics** (Templates). Powerful but difficult.
- `http://www.cplusplus.com/info/history/`

C++ Design Goals

- From Wikipedia
- C++ is designed to:
  - be a statically typed, general-purpose language
  - support multiple programming styles
    - (procedural programming, data abstraction, object-oriented programming, and generic programming)
  - give the programmer choice.
  - be compatible with C, and as efficient and portable.
  - function without a sophisticated programming environment
- Avoids platform specific, non-general features
- No overhead for unused features (the "zero-overhead principle")

1983 - Bjarne Stroustrup bolts everything he's ever heard of onto C to create C++. The resulting language is so complex that programs must be sent to the future to be compiled by the Skynet artificial intelligence. Build times suffer. Skynet's motives for performing the service remain unclear but spokespeople from the future say "there is nothing to be concerned about, baby," in an Austrian accented monotones. There is some speculation that Skynet is nothing more than a pretentious buffer overrun.

C++ vs Java

- Java ByteCode vs C++ Compiled Binaries (Portability vs Speed).
- Java enforces a strong OO paradigm, C++ does not.
    - OO Discipline vs many different coding styles.
    - Modern C++ code utilises a combination of **imperative**, **OO**, **generics** and **functional**.
- C++ has a pre-processor for specifying code inclusion and compiler directives.
- C++ is the older language and sports more complex features (operator overloading, multiple inheritance, templates)
    - C++ Templates far more advanced than Java Generics.
- C++ is still the industry standard for performant code.
    - Databases, Webservers and Games are more likely to be written in C++.

Useful online references

- Good references:
  - `http://www.cppreference.com`
  - `http://isocpp.org/tour`
  - `http://cplusplus.com`
- `http://www.learncpp.com`
  - good online course. covers material in similar order to this course.

C++ Hello World

```
#include <iostream> // Include IO stream library header

// Command line arguments
// argc: Number of parameters
// argv: Array of pointers to chars (strings).
int main(int argc, char * argv[])
{
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

- main function is entry point to program

C++ Program Structure

- C++ program usually composed of *header* (.h) and *source* (.cpp) files.
- Headers (.h) **declare** the existence of code elements.
  - Function signatures. Name, variables, return variables.
  - Class structure. Methods and members.
- Source files (.cpp) **define** the code.
  - Write code for functions and class methods
  - Compiled into **binary** object files (.o)
  - Source files include header files so that they can determine:
    - how to invoke functions and class methods in other object files.
    - how much memory a class occupies.
- Practically, included files inserted in place within .cpp file
  - and compiled.

C++ Program Structure

- Fibonnaci example
- We create
    - Driver source file containing **main** function. (fibdriver.cpp)
    - Header file containing function declaration (fib.h)
    - Source file containing function definition (fib.cpp)

C++ Structure Example

```
// Pre-processor directives that guard against
// redeclaration should fib.h be included
// again elsewhere.
#ifndef _fib_h
#define _fib_h

// A function DECLARATION,
// no function body.
int fib(int n);

#endif // matches the #ifndef
```

C++ Structure Example

```cpp
// Pre-processor directive including the DECLARATION
// of the fib function.
#include "fib.h"

// Function DEFINITION
int fib(int n)
{
  if(n <= 2)
    { return 1 };
  return fib(n-1) + fib(n-2);
}
```

C++ Structure Example

```cpp
#include <iostream> // Include I/O Stream library headers.
#include "fib.h"    // Include the DECLARATION of the fib function.

int main(void)
{
  int x;
  // Output to standard output
  std::cout << "Enter an integer: " << std::endl;
  // Read in an integer from standard input
  std::cin >> x;
  // Output the result
  std::cout << "fib(" << x << ") is " << fib(x) << std::endl;

  return 0; // Function needs a return value
}
```

Simple compile and link

- Compile source (.cpp) files to create binary code.
- **-c** compile to object file.

```
# Compile fib.cpp containing int fib(int n) function
# -c create binary object fib.o from fib.cpp
g++ fib.cpp -c
# fibdriver.cpp knows how to call the int fib(int n) function
# because #include "fib.h"
g++ fibdriver.cpp -c
```

- Link object files to create an executable
- Function calls between binary objects are “linked”.

```
g++ fibdriver.o fib.o -o fib
```

- **-o name** name of final executable

C++ Libraries

- C++ collects binary code into *libraries*.
- Unix libraries reside in /usr/lib and /usr/local/lib
  - **.so** (shared object) dynamic library.
  - **.a** (archive) static library.
- Binaries conform to Application Binary Interface (ABI)
  - Compiled for specific architectures. Intel, AMD, MIPS etc.
  - Faster than bytecode, although JIT
- **Declaration** (header file) needed by compiler to interpret library (**definition**).
  - Compiler needs to know HOW to call a given function. Arguments + return values.
  - e.g. #include <string> to use string class type.

C++ Libraries

- Directory structure

```
/usr/local
+-- mathlib
    +-- include
    |   +-- mathlib
    |       +-- bignumber.h
    +-- lib
        +-- libmathlib.so
```

- Header file tells compiler what bignumbers and multiply look like.

```
#include <mathlib/bignumbers.h>
...
bignumber huge_one, huge_two;
multiply(huge_one, huge_two);
```

- **-I directory** instructs compiler about header location.

```
# Compiler can now find bignumbers.h
g++ fib.cpp -c -I/usr/local/mathlib/include
```

C++ Libraries

- Directory structure

```
/usr/local
+-- mathlib
    +-- include
    |   +-- mathlib
    |       +-- bignumber.h
    +-- lib
        +-- libmathlib.so
```

- **libmathlib.so** contains code implementing bignumbers functionality.
- **-L directory** location. **-l libname** library

```
# Link against libmathlib.so
g++ fib.o fibdriver.o -o fib -L/usr/local/mathlib/lib -lmathlib
```

- Note (1) No lib prefix with flag. (2) libraries *after* object files.
- Tell unix about unusual shared library locations:

```
export LD_LIBRARY_PATH=/usr/local/mathlib/lib
```

More about linking

- **main** function in **fibdriver.cpp** files calls **fib** defined in **fib.cpp**.
- Compiled into .o binary code files.
- **main** binary code in **fibdriver.o** calls the **fib** binary code in **fib.o**.
- During linking phase, binary dependencies are **linked together**.
- **main** binary code knows how **fib** is called because it has seen the declaration in **fib.h**.
- Compiler may complain about *"undefined symbol"* or *"undefined reference"*.
- May also produce a **mangled**, somewhat decipherable name.
  - You're forgotten to link an object (.o) file, or library (.so).
  - Compiler can't find the binary for the function.

The Make Utility

- Projects consist of multiple source files and headers.
- changes may require recompile of the project.
- **make** automates the process of dependency checks.
- Configured with a **Makefile**, a collection of dependency rules.
- Makefile rules are of form:
  target: dependencies
  action
- after editing, type **make** and all will be consistent
  - **make** (use default Makefile) or **make –f** *makefilename*

The Makefile

- Example:

```
myexec: file1.cpp file2.cpp hdr.h
     g++ -o myexec file1.cpp file2.cpp -lm
```

- Builds **myexec** dependant on file1.cpp, file2.cpp and hdr.h
- .cpp and .h changes trigger rebuild
- Rebuild rule is given on the second line
- many different rule types etc, refer to man pages
- things to remember:
  - no spaces after return
  - tab to start rule (not a space etc!!)
- if your rules are wrong, the project code may be out of date

Makefile Variables

- Makefiles use variables to ease configuration
  - e.g. `CC=g++ # sets C++ compiler to g++`
- rules rewritten to use these variables

```
mytest: mytest.cpp
  $(CC) $(CCFLAGS) mytest.cpp -o mytest
```

- CCFLAGS has a default value (see man pages)
- "macros" or built in rules can be used to simplify e.g.

```
.cpp.o:
  $(CC) -C $<
```

  - the macro $< expands to a matching .cpp file
  - builds .o files using this build action for all .cpp files.
  - have default behaviour which we can override
- **.cc** is standard suffix, can add others: *.SUFFIXES*:

Other Makefile Info

- no space after line end
- tab for rule indent
- @ suppresses echo of rule (does not print out as it executes)
- if no target given, 1st rule made
- can make *target* (no dependency reqd)
    - # **make clean** (on command line)

```
# removes .o files
clean:
  rm *.o
```

- only changed files should rebuilt
- can insert files in a makefile with **include** statement

Canonical Makefile Example

```
CC=g++                              # Compiler
CUSTOMDIR=/home/alice/local         # Custom package location
LIBDIRS=-L$(CUSTOMDIR)/libs         # Library Locations
INCLUDES=-I$(CUSTOMDIR)/includes # Header File Locations
LIBS=-lalice                        # libalice.a
CXXFLAGS=$(INCLUDES) -Wall          # Headers + All Warnings
LDFLAGS=$(LIBDIRS) $(LIBS)          # Libs + their locations
TARGET=myprog                       # Name of executable
OBJECTS=f1.o f2.o f3.o              # Object files to build into exe
# Linking Rule
$(TARGET):  $(OBJECTS)
  $(CC) $(OBJECTS) -o $(TARGET) $(LDFLAGS):
  @cp $(TARGET) ./binaries
# Macro: automatically associate *.o with *.cpp
.cpp.o:
  $(CC) $(CXXFLAGS) -c $<

clean:
  @rm -f *.o
```

IDE's and Debuggers

- IDE's:
    - QtCreator
    - Eclipse CDT
    - Geany (Text Editor really)
- Compilers:
    - **g++** and **clang++** on linux.
    - **Minimalist GNU for Windows** (MingW) has **g++** . `http://nuwen.net/mingw.html`
    - **Visual Studio for Windows**.
- Debugging
    - g++/clang++. Compile with -g flag.
    - Then run GDB. QtCreator and Eclipse do this automagically.

The C++ Pre-processor

- No Java equivalent
- Modifies/processes source code prior to compilation
- pre-processor *directives* introduced by #
  - Preprocessor deals with all of these prior to compilation
  - Recursively processes include files, modifying source code passed to compiler
- Common uses:

```
#include <filename> // include files
#define MY_VALUE 1  // define macros
#pragma once        // set compiler behaviour
```

- Used to optimize, target platforms, compile only certain parts of code.
- use *#include file* to collect class/function declarations in one place C++
  - *source files* should contain (mainly) function **implementations**

Include Files

- The **#include** directive "inserts" the indicated file
  - At the point of the **#include**
  - Textual insertion which **modifies** the current file *before* compilation
- Include *header files*, for function prototypes, class defs etc
- Two include conventions:

```
#include <filename>
#include "filename"
```

- Version 1 searches default dirs (/usr/include)
- Version 2 searches explicit include dirs (-I/usr/local/matlib/include)
- If you include a file twice can give "redefinition" errors
  - use **#define** (see later)

Pre-processor Macros

- Define a *macro* with **#define**
- Can be "function" or constant :

```
#define MYINT    22
#define MYSQR(x) ((x) * (x))
```

  - MYINT replaced by 22 in C++ source
  - MYSQR(3) replaced by ((3)*(3)) in C++ source
- Convention is Upper-case Macro names
- Can set Macro values using compiler
  - g++ -DMYINT=22 ...
- Remember:
  - code replacement ONLY
  - final expression *must* preserve C++ syntax
- No semi-colon at end; to continue over lines, use \

```
#define MYLONGSTR "The quick brown fox\
jumped over the fence"
```

Conditional Macro Expansion

- conditionals: **#if, #ifdef, #ifndef**

```
#if MYVAL==4 // define f() for 4
string f(void) { return string("four"); }
#elif MYVAL==3 // define f() for 3
string f(void) { return string("three"); }
#else // define default f()
string f(void) { return string("fruit"); }
#endif
```

- use **#ifdef** or **#ifndef** to test if a macro has been defined
- can also **#undef** a defined macro
- Useful for writing multi-platform code:

```
#ifdef _USING_WINDOWS // windows specific code
string overlord(void) { return string("gates"); }
#elif _USING_MACOS //macos specific code
string overlord(void) { return string("jobs"); }
#endif
```

Header Files

- How to avoid multiple file inclusions:

```
//   Header file name: dog.h
#ifndef _DOG_H
#define _DOG_H
// stuff to include goes here, function declarations
void do_bark(dogtype dog);
#endif // Matches #ifndef _DOG_H
```

- If (in source file) we write:

```
#include "dog.h"
#include "dog.h"
```

- the second inclusion will do nothing, since the macro _DOG_H has been defined by the first inclusion.

Macro String Operations

- Stringizing: turning identifier into character string
  - **#define PRINT(y) std::cerr** << **#y "="** << **y** << **std::endl**
  - In code
    - PRINT(x); **std::cerr** << **"x="** << **x** << **endl;**
- Note: two strings next to each other are concatenated
- Token pasting: making new label/token from supplied tokens (tokens are NOT strings!)
- These operations are very useful for debugging code

```
#ifdef _DEBUG
#define INVOKE_FN(f)   debug_##f
#else
#define INVOKE_FN(f)   f
#endif
```

- In code: INVOKE_FN(F(3)); ⇒ **debug_F(3);** or **F(3);**
- depending on **_DEBUG** setting

Brief Explanation of code snippets

```
// A vector (resizable array) of ints.
// std::vector is templated with int variable
std::vector<int> intvec;
// std::cout is an output stream variable
// << adds data to the stream
// std::endl is a newline
std::cout << "5" << std::endl;
```

Simple C++ Types

- three kinds: simple, aggregate and class
- simple types are:
  - **char**: 8-bit integer value
  - **int**: standard system integer
  - **float**: system single precision float
  - **double**: system double precision float
  - **short**/ **long** / **long long**: short (half)/long (double) integer
- integral simple types can be **signed**/**unsigned**
  - **unsigned char c**: u_char may also be defined
  - **std::size_t**: standard unsigned integral type.
- simple type sizes are system dependent - **sizeof(type)**

```
cout << "System long size=" << sizeof(long) << " bytes.";
```

Integral Types

- 16-bit operating systems
  - int 16-bit, long 32-bit
- 32-bit OS
  - int & long 32-bit, long long 64-bit
- 64-bit OS
  - int 32-bit, long & long long 64 bit
- Modern alternative to **sizeof() – numeric_limits**.

```
#include <limits>
...
std::cout << "long size="
          << std::numeric_limits<long>::digits;
```

- Info on signed, integral, float parameters.
- **cstdint** header provides int32_t, int64_t, uint32_t, uint64_t types.

Scope

- What variable/function is a label bound to at this point in the code?
- Scope: classes, functions, code block.

```
class A { int a=1; }; // a in scope of class A
void func1(void) { int a=2; } // a in scope of func1
void func2(void) { int a=3; } // a in scope of func2
while(true) { int a=4; } // a in the scope of code block
```

- Variable is visible once defined
- Rule of thumb: Pairs of { } define new scope
- variable defined at **global** scope is visible everywhere
  - No Global scope in Java... and it's bad practice!
  - Every function/code in a source file can access a global variable!
- Labels **not** visible outside of scope:

```
{ int i = 3; i *= 2; }
// cannot see i here
```

Scope

- a variable defined in an enclosed scope hides one in outer scope:

```
int i = 10; // global/outer scope
for (int j = 0; j < 5; j++)
  { int i = j*2; cout << i << endl; }
```

  - local copy of **i** referenced; global unaffected
- you can access global scope variables using scope operator, **::**

```
int i = 10; // defined OUTSIDE  code/class
for (int j = 0; j < 5; j++)
  { int i = j*2; cout << ::i << endl;}
```

- ignore local version and print global version always
- Globals can cause confusing errors - try not to use them

Scope

- Scope is important!
- In C++, *lifetime of* **automatic** *variables is bounded to scope.*

```
void somefunction(void)
{
  anobject a; // Automatic variable

  {
    anobject b; // Automatic variable
    int c;      // Automatic variable
  } // Leaving scope, b and c are DESTROYED

} // Leaving function scope. a is DESTROYED
```

- *Resource Acquisition is Initialisation* (RAII)
- More to come...

Namespaces

- Large projects may have name “clashes”
- Overcome by qualifying each “label” in some way
- Solution: use a **namespace**
    - Labels in the namespace are “prefixed” with the namespace
    - No duplicate definitions within namespace.
    - Use scope resolution operator **::** to refer to namespace labels
- Syntax **: namespace id** { **// names** }

```
namespace project { int p1; }
namespace projectx { float p1; }
cout << project::p1 << projectx::p1;
```

- In fib.h

```
namespace maths { int fib(int n); }
```

- In fib.cpp

```
namespace maths {
int fib(int n) {
  if(n <=2) return 1
  else return fib(n-1)+fib(n-2)
} // close function brace
} // close namespace brace
```

- In fibdriver.cpp

```
int main(void) {
  std::cout << maths::fib(10) << std::endl;
}
```

- Works the same with classes.

Namespaces

- Nested namespaces

```
namespace foo { namespace bar { int a; } }
foo::bar::a = 5;
```

- Unnamed namespaces. Local to current compilation unit (.cpp file).

```
namespace { int a = 1; }
std::cout << ::a << std::endl;
```

- Don't mixed global namespace and unnamed namespace

```
namespace { int a = 1; }
int a = 2;
std::cout << a << std::endl; // Outputs 2
```

Namespaces

- Explicit qualification:

```
std::cout << project::p1 << std::endl;
```

- **using** *declaration* (one label only):

```
for (int i=0; i < 10; ++i) {
  using project::p1;
  std::cout << p1 << std::endl;
}
```

- Aliasing

```
namespace ublas = boost::numeric::ublas;
ublas::blastype a;
```

- Can use **using** *directive* (all labels part of current scope):

```
using namespace std; // everything now at current scope
cout << "foobar" << endl; // no std::cout or std::endl
// DON'T USE IN HEADERS -  exposes whole namespace in each included file.
```

- C++ standard namespace **std** contains system classes, global

Variable Qualifiers

- a variable can have following qualifiers:
  - **extern**: variable defined outside current scope
  - **static**: variable bound to class/function/file
  - **const**: the value cannot be changed after initialisation
  - **register**: suggests that compiler use CPU registers to store variable
  - **volatile**: variable protected from compiler optimisations

Extern Qualifier

- Refers to a variable declared in some other compilation unit.
- globals.h

```
#ifndef _GLOBALS_H
#define _GLOBALS_H
namespace corsairs { extern int long_john_silver; }
#endif
```

- globals.cpp

```
#include "globals.h"
namespace corsairs { int long_john_silver; }
```

- driver.cpp

```
#include "globals.h"
void main(void) { corsairs::long_john_silver = 5; }
```

- binary code in driver.o can access variable in globals.o

Static Qualifier

- **static** variables can be *local* or *global*
  - global bound to file; invisible outside file, global in file
  - local bound to class or function; persistent

```
// Only visible within source file. Can't extern it
static int k = 0;

int func(void)
{
  static int i = 0; // Set to 0 on initial execution
  return i++;
}
...
cout << func() << endl; // outputs 0
cout << func() << endl; // outputs 1
cout << func() << endl; // outputs 2
```

Const Qualifier in Function Variables

```
void func(const int arg)
{
  const int a = arg;
  const int b = 2;

  arg = 6 // Compiler complains
  a = 5;  // Compiler complains
  b = 3;  // Compiler complains

  // Alternative expression - Not convention
  int const c = 4;
}
```

Type Definitions

- Create new type name from old.
- Simpler code, less typing.
- Obeys scoping principles.
- Example 1:

```cpp
// type u_char is an unsigned char
typedef unsigned char u_char;
```

- Example 2: Long type names

```cpp
std::vector<float> vec;
typedef std::vector<float>::const_iterator it;
it i = vec.begin(); // I'm not typing that out again
```

C++11 - type related constructs

- Example 1: C++11 – **auto** keyword
- Deduce type of variable from expression on lhs

```
std::vector<float> vec;
// Didn't even have to type
// the iterator type out (even) once
auto i = vec.begin();
```

- Example 2: C++11 – **decltype** keyword
- Deduce type of variable from supplied expression;

```
int an_int;
// Figure out the type of this
// variable at compile time.
decltype(an_int) another_int = 5;
```

Aggregate Types: structures

- groups data into a "record"
- ancestor of the **class**
  - All data and methods are **public**
  - For backward compatibility with C
  - Use a class if you really want a class!
- introduced by keyword **struct**:

```
struct DataEntry {
  int IdNumber;
  char name[40];
  char address[300];
}; // NB! semi-colon
```

- **DataEntry** is now a valid type:

```
DataEntry d1;
cout << "Name is: " << d1.name << endl;
d1.IdNumber = 1048576;
```

Structures

- can create *singleton* structures (no **struct** name):

```
struct { int a; } s1;
```

- unique instance
- Assignment operator does a shallow copy, byte-by-byte

```
DataEntry d1 = d2;
```

- pointers will not be accessed – shallow copy

Enumerations

- Not related to Java Enumeration
- Create a set of named integer constants:

```
enum name {label_1, ..., label_n};
```

- first integer will be zero, increment for each label
- can change integer mapping e.g.

```
enum DaysOfWeek {Sun=1,Mon,Tues,Wed,Thur,Fri,Sat};
```

- this is now a valid type:

```
DaysOfWeek dd;
if (dd == Fri) cout << "It's Friday!" << endl;
```

- Enumeration type is not **int**
- Enumeration scope is global or class
- For a class enum use :: outside of class, e.g

```
MyClass::DaysOfWeek x = MyClass::Sun;
```

Class Types

- Declared with class keyword (cf. Java)
- Declares new type in header file (person.h)

```
#ifndef PERSON_H
#define PERSON_H
namespace spooks { // in namespace
class person {
private:             // private members
  std::string n;
public:              // public members
  person(std::string name); // constructor
  void set_name(std::string name);
}; // NB! semi-colon
}  // end namespace
```

- C++ separates method code from class declaration
- Note ‘;’ after last bracket !!!
- Access applies to all following members, until changed

Class Types

- Implement methods in source file (person.cpp)

```
#include "person.h" // incl class declaration
// implementation of methods
namespace spooks {
person::person(std::string name) : n(name) {}   // constructor
void person::set_name(std::string name) { n = name; } // member fn
} // end namespace
```

- Scope operator :: and class name associates declaration + definition

```
// file: driver.cpp
#include "person.h"
int main(void) {
  spooks::person X("Fox Mulder"); // create instance of person
  X.set_name("Dana Scully");      // apply method to it. operator.
  return 0;
}
```

- Note the namespace qualification.

Variable Initialisers

- Simple Variables

```
float a = 0.4534534e-10;      // simple vars
int b[5] = { 0, 1, 2, 3, 4 }; // arrays
```

- Structure: field by field:

```
struct Name { char a; int numbers[3]; float t; };
Name tt = {'A', {1,2,3}, 0.5};
```

- initialised in order; brackets for multi-value fields:

```
int myarray[3][3][2] = {
  { {1,2}, {3,4}, {5,6} },
  { {7,8}, {9,10},{11,12} },
  { {13,14}, {15,16}, {17,18} }
};
```

- Only works for Plain Old Data (POD).
- Class types initialised with a *constructor*.

C++11 Initializer Lists

- Extends Initializer Lists to work with non-POD constructs.

```
std::vector<int> a = { 0, 1, 2, 3, 4 };
```

- instead of

```
std::vector<int> a;
a.push_back(0);
a.push_back(1);
```

- Constructor takes a **std::initializer_list<type>** variable

```
class MyList {
  public:
    MyList(const std::initializer_list<int> & rhs)
    {
      for(auto i = rhs.begin(); i != rhs.end(); ++i)
        { /* construct MyList */ }
    }
}
```

Type Conversion

- C++ strongly typed language (unlike C...)
- automatic or explicit *type casts*.
- automatic casts: expressions/assignments, function params, class etc
- Explicit:

```
float x = 4.0f;
int i = (int)(x) + 2; // old style
int j = int(x) + 2;   // new style
```

- old style for e.g. **unsigned char**, **long long**
- type conversion may be unsafe: compiler tries to limit this
- e.g. cast **int** to **short** and back to **int** looses 2 bytes of data

C++11 constexpr

- Always had constant expressions e.g. $3 + 4$
- Now can have “constant” functions

```
constexpr int get_five(void) {return 5;}
// Create an array of 12 integers.
int some_value[get_five() + 7];
```

- Some Limitations
  - Must return non-void
  - Can't declare variables or new types in body
- Evaluated at *compile time*

C++ I/O

C++ I/O

C++ Input and Output

- Console I/O, via global objects **cout**, **cerr**, **clog** and **cin**.

```
#include <iostream>
std::string s; double d;
std::cout << "Hello world " << s << ' ' << d << std::endl;
std::cin >> s >> std::ws /* consume ws */ >> d; // Read string+double
```

- File I/O via instantiated **ofstream** and **ifstream** objects.

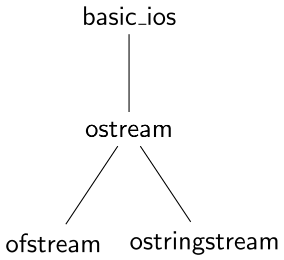
```
#include <fstream>
std::ofstream out("output.txt"); std::ifstream in("input.txt");
out << "Hello world " << s << ' ' << d << std::endl;
in >> s >> std::ws /* consume ws */ >> d; // Read string+double
```

- Memory I/O via **ostringstream** and **istringstream**.

```
#include <sstream>
std::ostringstream oss; std::istringstream iss("FooBar 1.234");
oss << "Hello world " << s << ' ' << d << std::endl;
iss >> s >> std::ws >> d;            // Get string+double from iss
std::cout << oss.str() << std::endl; // Print the oss' string.
```

I/O Stream Hierarchy

- http://en.cppreference.com/w/cpp/io
- **ofstream** + **ostringstream** inherit from **ostream**.



- base class **ostream &** can bind to **ofstream/ostringstream** variables.
- Similarly with input classes.

I/O Stream Operators

- I/O Operators
  - << appends data to stream object.
  - >> removes data from stream object.
- Can be **overloaded** for custom types.

```
class point { public: int x; int y; };
```

- overload stream output << operator. **ostream** is base class.

```
ostream & operator<<(ostream & out, const point & p)
  { out << p.x << ' ' << p.y; return out; }
```

- overload stream input >> operator. **istream** is base class.

```
istream & operator>>(istream & in, point & p)
  { in >> p.x >> std::ws /* consume ws */ >> p.y; return in; }
```

- streams can now output and input point

```
point origin;
cin >> origin;  // >> overloaded for point
cout << origin; // << overloaded for point
```

Console I/O

- via methods and overloaded operators (<<,>>)
- Example of simple I/O:

```
int i; float f;
// Add a string to cout
cout << "Enter an integer and a float: ";
cin >> i >> ws >> f; // Remove an int and a float from cin
```

- **cout** writes to stdout, **cin** reads from stdin (normally)
- **cerr** is mapped to stderr (no redirection)
- the I/O operators (<<,>>) write/read text data

```
cout << "\tApple\n";        // ASCII escape codes
cout << hex << 45 << endl; // hex and endl I/O manipulators
cout.setf(ios::left | ios::fixed); // Status Flags
```

- Special characters embedded in text string
- can cause newline, carriage return, “alarm”, backspace etc.
- some examples:
  - **n,**\t: newline and tab
  - \**"**: double quote
  - \NNN,\xNNN: print char with octal/hex code NNN
- examples:
  - **cout << "The terminal will beep now \a!";**
  - **cout << "Insert a \t tab or \t two";**

Manipulators

- Manipulators more powerful than escape codes
- http://www.cplusplus.com/reference/iostream/manipulators/
- affects behaviour of stream e.g. change printing precision
- examples:

```
cout << "the end-of-line manip" << endl;
cout << "write numbers as hex" << hex << 45;
cin >> a >> ws >> b; // ws consumes whitespace
// #include <iomanip> for setprecision
cout << scientific << setprecision(8) << 1.342355;
```

- can write own manipulators
- can test stream for errors e.g.

```
if (!cin) { panic(); }
if (cin.eof()) { panic(); }
```

Reading Console Data

- >> operator is *overloaded* to support different data types.
- examples:

```
string mystring;
cin >> mystring; // Reads till whitespace
float f;
cin >> f; // Read in a float. e.g. try "1.45e-8"
```

- many other methods available - special formatting etc

```
int i;
cin >> hex >> i; // Read in hex. Try 0xAB
cin >> octal >> i; // Read in octal. Try 054
```

Reading Lines of Console Data

- **getline** is an *extremely* useful function.

```
#include <string>     // Needed for string objects
#include <iostream>   // Need for cout and cin

void main(void) {
  std::string s;
  std::getline(std::cin, s,'\n'); // Reads till '\n'.
  std::cout << s << std::endl;    // Output the input
}
```

- Watch for dodgy input

```
int i; float f;
// Assume user types "4 4.2 hello world"
cin >> i >> f; // Consumes the "4 4.2"
cin >> i >> f; // Tries to consume "hello world". FAILS
```

Reading from cin

- Are we there yet? **cin.eof()** tests for end of input.
- Ctrl D signals EOF for input data.
- Example:

```
vector<string> items;
string s;
while (!cin.eof()) {
  cin >> s >> ws;
  items.push_back(s);
}
```

- Note: **eof()** only true when input buffer empty
- The **ws** manipulator consumes white space

Memory-based I/O

- Format strings or read from strings in memory.
- Useful for converting non-string types to string
- Use **stringstream** preferably
    - **strstream** for old **char** * strings. deprecated + buggy.
    - **istringstream**/ **ostringstream** depending on input/output
- Example:

```
#include <sstream>
int i1, i2; float f1; string str;
string input = "hello 1 2 2.3";
istringstream is(input);
is >> str >> i1 >> i2 >> f1;
```

File-based I/O

- two basic types: input and output file streams
- simple file I/O: **fstream**, **ifstream**, **ofstream**
- easier to use most appropriate one:

```
#include <fstream>
ifstream myfile;
myfile.open("file.dat");
if (!myfile)
  { cerr << "File open failed!"; }
```

- Create and open at same time

```
ifstream myfile("file.dat");
myfile.close(); // close the file
```

- Always close files (destructors tho)

Text File I/O

- overloaded operators (<<,>>) available; derived from **ios**
- example of reading from file (text):

```
int i;
while (!myfile.eof()) {
  myfile >> i >> ws;
  cout << "The next data item is " << i << endl;
}
```

- text must have ints; other input ignored
- type of input determined by variable
- white space separates data items (use **ws**!)

Binary File I/O

- Example of reading in binary data:

```
#include <iostream>
#include <fstream>
...
const int ARRAY_SZ=40;
char array[ARRAY_SZ];
ifstream myfile("binary.dat", ios::binary);
while (!myfile.eof()) {
  int n = myfile.read(array, ARRAY_SIZE);
  cout << "Data: ";
  for(int i=0; i<n; ++i) { cout << array[i]; }
  cout << endl;
}
...
outfile.write(array, ARRAY_SIZE); // Binary output
```

Files: some comments

- there are many possible file error conditions
- a class enumeration, **ios::io_state** contains these:
  - **badbit, eofbit, failbit** etc
- status functions check and set these “stream status bits”
- for file open modes, **ios::open_mode**:
  - **trunc, in, out, app, binary, ate**
- for specialised streams, default values used
- direct (random access) of files is possible:
  - tell()/seek())
- every file has a stream pointer, indicating current position