Tutorial – C++

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Content

Good Program

The greatest obstacle to learning to write good programs is a firm belief that (by Bjarne Stroustrup)

- it is among the most difficult of technical skills
- it is a skill that requires some rare talent
- it is done by socially inept young men in total isolation, mostly by night
- it is mostly about building violent video games
- it is a skill that requires mastery of advanced mathematics
- it is an activity completely different from everyday ways of thinking
- it is something that doesn't help people

Hello World!

- Enter the source code with your favorite editor (Emacs for me) and save to file `hello.cpp`.

```
// my first C++ program
#include <iostream>
using namespace std;

int main() {
  cout << "Hello World!" << endl;
}
```

- Fire up the command line, compile with `g++ hello.cpp`. An executable named `a.out` appears in the same directory. Run it with `./a.out`.

Hello World! – Line 0

```
// my first C++ program
```

Comments are stripped by *cpp* (c preprocessor).

- Line comment. Start with //, everything after it is ignored.

```
// comment
int main() {                    // comment
}
```

- Block comment. Start with /* and end with */. Nested comment is not allowed. Everything in between is ignored.

```
/* comment */
int main(/* comment */) { /* comment */
}
```

Hello World! – Line 1

```
#include <iostream>
```

Preprocessor directives precede by a hash sign #. They are interpreted by the preprocessor. This line

- Search for the file `iostream` in standard search path, if not specified otherwise (with `-I`).

  ```
  /usr/local/include
  libdir/gcc/target/version/include
  /usr/target/include
  /usr/include
  ```

- If found, copy everything to current file. Otherwise error.

Hello World! – Line 2

```
using namespace std;
```

Everything declared in name space `std` is available, e.g., `cout` which is declared in `iostream`.

```
namespace std {
  // ...
  ostream cout;
  // ...
}
```

`namespace` is used to prevent name collision and global namespace polluting.

Hello World! – Line 3

```
// empty line
```

Spaces are allowed in C++ anywhere between basic units. Different styles require different spaces for indentation, spaces around function names, etc. E.g., K&R, Stroustrup Style, Google C++ Style Guide and etc.
The International Obfuscated C Code Contest (IOCCC) is a good example of space usage.

Hello World! – Line 4

```
int main()
```

This line initiates the declaration of a function. Essentially, their definition is introduced with a succession of

- a return type, e.g., `int`,
- a function name, e.g., `main`, and
- a pair of parentheses `()`, optionally including parameters.

`main` function is the *entry point*. At most one definition is allowed. No `main` function in libraries.

Hello World! – Line 4 & 6

```
{}
```

- The brace at line 5 `{` indicates the beginning of the main function and `}` the end of it.
- Everything between brace pair is the *body* of the function.
- Brace pair is used to delimit *scope* in C++.

Hello World! – Line 5

```
cout << "Hello World!" << endl;
```

This line is a C++ statement. Usually they are executed in the same order they appear unless Level 3 optimization (-O3)is on.

- cout is character/console output.
- « is insertion operator. Insert what follows into the output, i.e. print to the console.
- The statement ends with SEMICOLON (;).

Environment

Platform By default I recommend *nix OS, choose whatever distribution you like, e.g. Ubuntu, CentOS.

- Linux machine in Computer Lab and Davis Hall 123. You may SSH to machines in Davis Hall 123.
- Otherwise, do the configuration yourself.

IDE Integrated Development Environment, e.g., Visual Studio, CodeBlocks, Eclipse and etc., is not recommended for programming assignment.

Compiler I use `g++ 5.2.1` for `C++`. So feel free to use any of the `C++11` features in the programming assignment.

General

The ISO C++ standard defines two kinds of entities:

- *Core language features*, such as built-in types (e.g., `char` and `int`) and loops (e.g., `for`-statements and `while`-statements)
- *Standard-library components* (in namespace `std`), such as containers (e.g., `vector` and `map`) and I/O operations (e.g., « and `getline()`).
  The standard-library components are perfectly ordinary C++ code provided by *every* C++ implementation.

Content

Variable

Fundamental types include

Character Represents a single character, e.g., `char`.

Numeric integer Stores *exact* integer value. They vary in signedness and size, e.g., `int`, `long long`.

Floating-point Represent real values, such as 3.14 or 0.01, with different levels of precision, e.g., `float`, `double`.

Boolean Represent one of two states. e.g., `bool`.

C++ standard library (STL) provides extension such as `std::string`, `std::iterator` and etc.

Literals

Literals are constants refer to fixed values that the program may not alter.

- Integers. E.g., 212 `signed int` in decimal, 215u `unsigned int` in decimal, 0xFEEL `signed long` in hexadecimal, 066 `signed int` in octal.
- Floating points. E.g., 3.1415, 3.1e-2, 1.f (note the dot in between).
- Boolean literals, `true` and `false`. 0 is `false` and all others are `true`.

Arithmetic

These are the basic arithmetic operators in C++.

```
int main() {
  int x = 3, y = 4;
  x + y;                          // plus
  +x;                             // unary plus
  x - y;                          // minus
  -x;                             // unary minus
  x * y;                          // multiply
  x / y;                          // divide
  x % y;                          // remainder (modulus, integer only)
}
```

Comparison

These are the basic comparison operators.

```
int main() {
  int x = 4, y = 3;
  x == y;                          // equal (BE CAREFUL for float)
  x != y;                          // not equal
  x < y;                           // less than
  x > y;                           // greater than
  x <= y;                          // less than or equal
  x >= y;                          // greater than or equal
}
```

Logic

Logic operators include

```
int main() {
  int x = 3, y = 1;
  x & y;                    // bitwise AND
  x | y;                    // bitwise OR
  x ^ y;                    // bitwise XOR
  ~x;                       // bitwise complement
  x && y;                   // logic AND
  x || y;                   // logic OR
}
```

- A bitwise logical operator yields a result of their operand type for which the operation has been performed on each bit.
- The logical operators && and || simply return `true` or `false` depending on the values of their operands.

Assignment

```
int main() {
  int a = 5;                    // OK, a lvalue, 5 rvalue
  a + 3 = 4;                    // WRONG, a + 3 rvalue
  int a = 5.243;                // narrowing conversion
}
```

L-value Expressions that refer to a memory location, potentially allowing new values to be assigned. An lvalue may appear as either the left-hand or right-hand side of an assignment.

R-value That are not L-values. They may appear only on the right hand side of the assignment.

Conversion

C++ performs all *meaningful conversions* between the basic types so that they can be mixed freely.

Usual Arithmetic Conversions Conversions used in expressions aim to ensure that expressions are computed at the *highest precision* of its operands.

```
int main() {
  int x = 3;
  double b = 4;
  char c = 'a';
  x + b;                    // x is converted to double
  c + x;                    // c is converted to int
}
```

Variable Initialization

- Don't introduce a name until you have a suitable value for it.
- =-form is traditional and dates back to C. If in doubt, use the general {}-list form, which saves you from conversions that lose information, i.e., narrowing conversion.
- Types may be deduced from the initializer.

```
int main () {
  int a = 5;            // initial value: 5
  int b(3);             // initial value: 3
  int c{2};             // initial value: 2
  int result;           // initial value undetermined
  auto d = a;           // d is integer 5, deduced type: int
  decltype(c) e;        // integer value undetermined

  int e{7.2};           // ERROR, narrowing conversion
  double f{'a'};        // OK
}
```

Shorthand Operators

C++ provides concise and convenient shorthand operators to modify variables. All arithmetic and bitwise logic operators may be combined with =.

```
int main() {
  int x = 4, y = 3;
  x += y;                          // x = x + y;
  ++x;                             // x = x + 1; use x;
  x++;                             // use x; x = x + 1;
  --x;                             // x = x - 1; use x;
  x--;                             // use x; x = x - 1;
  x *= y;                          // x = x * y;
  x |= y;                          // x = x | y;
}
```

Variable Scope

Like most PL's, C++ uses *lexical scoping* or *static scoping*. A declaration introduces its name into a scope.

Local Scope *Local Name* is declared in a function or lambda. Scope extends from the point of declaration to the end of the block where it is declared.

Class Scope *Member name* (class member name) is defined in a class outside any function, lambda, or enum class. Scope extends from the opening { of its enclosing declaration to the end of that declaration.

Namespace Scope *Namespace member name* is defined in a namespace outside any function, lambda, class, or enum class. Scope extends from the point of declaration to the end of its namespace.

A name not declared inside any other construct is called a global name and is said to be in the global namespace.

Variable Scope Example

An object *must* be constructed/initialized before it is used and will be destroyed at the end of its scope.

```
struct foo {
  int count;                        // member name
};

int g = 4;                          // global

int main(int argc) {                // main is global, argc is local
  int a = 3;                        // local, visible in main
  {
    int b = 4;                      // local visible inside {}
  }
  int g = 3;                        // local g, shadows global g
  ::g = 40;                         // refer to global g
  c = 4;                            // ERROR!!
}
```

Constants

`const` "I promise not to change this value". Primarily used to specify interfaces.

`constexpr` "To be evaluated at compile time". Used primarily to specify constants, to allow placement of data in read-only memory.

```
constexpr int square(int n) { return n * n; }

int main() {
  const int a = 3;                  // a is named constant
  int b = 3;                        // b is not constant

  constexpr int c = 1.4 * square(a); // OK
  constexpr int d = 1.4 * square(b); // ERROR
  const int e = 1.4 * square(b);     // OK
}
```

Array

```
int main() {
  int a[3]{1,2,3};              // array of 3 integers
  char b[4];                    // array of 4 characters
  int size = 10;
  int b[size] = {1,2,3};        // runtime-sized array c++14
  int c[3][4];
}
```

- In declarations, `[ ]` means "array of".
- All array are 0-based. So `a[0]` is the first element and `a[2]` is the last element.

Multi-Dimensional Array

In memory, milti-dimensional array is actually a long 1D array, in stored row-major order.

Syntax Sugar

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

int main () {
  const int W = 3;
  const int H = 2;
  int a[H][W] = {1, 2, 3, 4, 5, 6};

  for (int i = 0; i < H; ++i)
    for (int j = 0; j < W; ++j)
      cout << a[i][j] << endl;
}
```

Pseudo Multi-Dimensional

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

int main () {
  const int W = 3;
  const int H = 2;
  int a[H * W]{1, 2, 3, 4, 5, 6};

  for (int i = 0; i < H; ++i)
    for (int j = 0; j < W; ++j)
      cout << a[i * W + j] << endl;
}
```

Character Array

Strings in C/C++ are represented by `char` array. By convention, *null character*, '\0', is used as terminator, signaling the end of string.

```
#include <iostream>
#include <cstdio>
int main() {
  char foo[10] = "1234";
  std::cout << foo << std::endl;
  char bar[10] = "1234\0abcd";
  printf("%s\n", bar);
}
```

Pointer

- Pointer type stores the *address* of memory location, Pointer may change its value, i.e., point to another memory location.
- Reference type is *auto de-referenced* pointer. But its memory address may not change.

```
int main() {
  int a = 3;
  int* b = &a;
  int& c = a;
  // a, b, c share a memory location
  (*b) = 4;
  c = 40;
}
```

Pointer Cont'd

Prefix unary * means "contents of", and prefix unary & means "address of". In declarations, operators (such as & , * and []) are called *declarator operators*.

- When we don't have an object to point to or
- if we need to represent the notion of "no object available" (e.g., for an end of a list)

The pointer is set the value `nullptr` ("the null pointer"). There is only one `nullptr` shared by all pointer types.

```
int main() {
  int* a = nullptr;
  double* b = nullptr;
  int x = nullptr;                // ERROR
}
```

Null Pointer

In old code, 0 or `NULL` is typically used. However, using `nullptr` eliminates potential confusion between integers, e.g., 0 or `NULL`, and pointers, e.g., `nullptr` which is of type `std::nullptr_t`.

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

int main() {
  cout << (0 == nullptr) << endl;
  cout << (NULL == nullptr) << endl;
  cout << (0 == NULL) << endl;

  int a = NULL;                    // warning
  int x = nullptr;                 // ERROR
}
```

Dynamic Memory

- Allocating memory on heap with `new`.
  - allocating blocks of memory and
  - invoking constructor.
- De-allocating memory with `delete`.
  - invoking destructor and
  - returning memory to system.

```
int main() {
  int* arr;
  arr = new int [10];
  delete[] arr;

  int* val;
  val = new int (3);
  delete val;
}
```

Content

Conditional Branch `if else`

The condition is not necessary Boolean type. It may also be types that could be *implicitly* cast to Boolean, e.g., char, integer and etc. If no curly bracket, only the first statement block belongs to `if`.

```
int main() {
  int a = 0;
  if (a) a = 100;
  else if (3 == a) a = 4;

  if (a)
    a -= 1;
  a -= 1;
}
```

Conditional Branch `switch case`

The condition is expected to be integer-compatible.

```
int main() {
  int a = 2;
  switch (a) {
  case 1: { a -= 1; break; }
  case 2: a -= 2; /* missing break */
  case 3: a -= 3; break;
  default: a -= 0;
  }
}
```

Remember the `break` statement at the end of every `case` unless intended otherwise. Curly bracket for each `case` body is optional.

Unconditional Branch goto

Avoid goto. The only useful case I know is deeply nested loop.

```
int main() {
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < i; ++j) {
      for (int k = 0; k < 100; ++k)
        if (5 == k) goto end;
    }
  }

 end:
  ;
}
```

Loop for

- Initial expression is evaluated once before the loop.
- Termination condition is evaluated before every loop.
- Iteration expression is evaluated after every loop.

```
int main() {
  int b = 10;
  for (int i = 0; i < b; ++i) {
    b -= 2;
    int c;   // visible only inside for
  }
}
```

Loop Range for

Since C++11, a new for loop syntax is available.

```
#include <iostream>

int main() {
  int a[3]{1, 2, 3};
  for (int i : a) {
    std::cout << i << std::endl;
  }
}
```

Loop while

The two loop structures are equivalent:

- `while(cond) { }`
- `for(; cond; ) { }`

```
int main() {
  int a = 3;
  while (a > 0) --a;

  int b = 4;
  do {
    --b;
  } while (b > 0);

  int c = 4;
  while (--c > 0) /* empty */ ;
}
```

Content

Function

Function consists of return type, function name, argument list and function body.

```
int foo(int a, int b = 3) {
  return a + b;
}
int bar(int a, int b = 3, int c) { // WRONG
  return 0;
}
int fun();
int main() {
  int b = foo(3);
  int c = foo(3, 4);
  fun();
}
int fun() { }
```

Function Side Effect

C++ allows functions to have *side effects*.

```
#include <iostream>
int f0(int a) { a -= 1; return a; }
int f1(int& a) { a -= 1; return a; }
int f2(int* a) { *a -= 1; return a; }

int main() {
  int a = 3;
  f0(a);      // value of a?
  f1(a);      // value of a?
  f2(&a);     // value of a?
  std::cout << f1(a) << std::endl;
  std::cout << a << std::endl;
}
```

Lambda Function

It is introduced in C++11.

Lambda function Constructs a closure: an unnamed function object capable of capturing variables in scope.

```
#include <functional>
#include <iostream>
int main() {
  int a = 10;
  std::function<int(int)> func = [&](int p) { return (a += p); };
  std::cout << func(33) << std::endl;
  std::cout << a << std::endl;
}
```

Overloading

Tow different functions may have the same name if their parameter are different

- different number of parameters, or
- any of their parameters are of a different type.

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

int add(int a, int b) { return a + b; }
float add(float a, int b) { return a + b; }

int main() {
  cout << add(3, 4) << endl;
  cout << add(2.3f, 4) << endl;
  cout << add(2.3, 4) << endl;  // ERROR, ambiguous
  cout << add(2.3, 3.4) << endl; // ERROR, ambiguous
}
```

Preprocessor Directives

- Preprocessor directives are lines preceded by a hash sign #.
- The preprocessor process these directives before compilation.
- Backslash \ may be used to extend a directive to multiline.

Directives – Macro

#define and #undef.

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

#define SIZE 3
#define max(a, b) (a) > (b) ? (a) : (b)
#define str(x) #x
#define glue(a,b) a ## b

int main() {
  int v[SIZE];
  max(3, 4);
#undef SIZE
  char m[SIZE];                    // ERROR

  glue(c, out) << str(Hello\nWor
                  ld!) << endl;
}
```

Directives – Conditional Inclusion

`#ifdef`, `#ifndef`, `#if`, `#endif`, `#else` and `#elif`.

```
int main() {
#ifdef SIZE
  int v[SIZE];
#else
  int v[10];
#endif
}
```

is equivalent to

```
int main() {
  int v[10];
}
```

Directive – Line Control

```
int main() {
#line 11 "Test Message"
  int 3a;
}
```

Generates error message

```
Test Message: In function ‘int main()’:
Test Message:11:7: error: ...
```

Without `#line`

```
test.cpp: In function ‘int main()’:
test.cpp:3:7: error: ...
```

You don't actually need to set this directive manually.

Directive – Error

```
#if __cplusplus > 10
#error A C++ compiler is required!
#endif

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

This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter.

Directive – Inclusion

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

int main() { }
```

- The first header, between angle-brakets, is system wide header files, i.e., those in system search path.
- The second header, between double quotes, is searched locally then system wide if failed.

Directive – Pragma

This directive is used to specify diverse options to the compiler. Compiler *silently* ignores this directive if no support.

```
#ifndef FOO_H_
#define FOO_H_

#prama once                          // include only once

class Foo { };

#endif /* FOO_H_ */
```

Predefined Macros

The following macros are set by the compilers.

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

int main() {
  cout << __LINE__ << endl
       << __FILE__ << endl
       << __DATE__ << endl
       << __TIME__ << endl
       << __cplusplus << endl;
}
```

Content

Enumeration

```
int main() {
  enum class Color {Red, Yellow, Green};
  enum class TrafficLight {Red, Yellow, Green};

  Color color = Color::Red;
  TrafficLight tl = TrafficLight::Red;
}
```

Enumerators, e.g., `Red`, resides in the scope of their `enum class`.
`Color::Red` is different from `TrafficLight::Red`.
Primarily used to make code more readable and less error-prone.
By default, the integer values of enumerators starts with 0 and increases by one for each additional enumerator.

enum class vs enum

enum class is strongly typed.

```
int main() {
  enum class Color1 {Red, Yellow, Green};
  enum class Color2 {Red, Yellow, Green};
  enum       Color3 {Red, Yellow, Green};

  Color1 x = Red;                 // ERROR
  Color1 y = Color2::Red;         // ERROR
  Color1 z = 2;                   // ERROR
  Color3 a = 2;                   // ERROR
  int i = Color1::Red;            // ERROR
  int j = Color3::Red;            // OK
  int k = Red;                    // OK, Color3::Red
}
```

enum class Operation

By default, enum class supports assignment, initialization and comparisons. Additional operation may be defined.

```
#include <iostream>
using namespace std;
enum class Color {Red, Yellow, Green};
Color& operator++(Color& t) {
  switch (t) {
  case Color::Green: return t = Color::Yellow;
  case Color::Yellow: return t = Color::Red;
  case Color::Red: return t = Color::Green;
  }
}
int main() {
  Color c = Color::Red;
  cout << static_cast<std::underlying_type<Color>::type>(c) << endl;
}
```

Structures

- General syntax for declaration and intialization
- Main usage, i.e., collect data
- Access fields

See the accompany code and comments.

Unions

Union A struct in which all members are allocated at the same address so that the it occupies only as much space as its largest member. The most recent written member is *active*. Only one member is active at a time.

It's *undefined behavior* to read from the member of the union that wasn't most recently written. Many compilers implement, as a non-standard language extension, the ability to read inactive members of a union.

Classes

- Basic syntax for declaration and definition
- Techniques and concepts, e.g., abstract/concrete class, inheritance, interface, etc.
- Operator overloading
- Separate compiling, modularity
- Demo Complex Class

Content

Standard Library Components

- run-time language support, e.g., for allocation and run-time type information.
- The C standard library, e.g., `cstdio`, `cmath`.
- strings, support for international character sets and localization
- regular expression
- I/O streams
- a framework of containers, e.g., `vector`, `map` and algorithms, e.g., `find()`, `sort()`.
- support for numeric computation
- support for concurrent programming
- support for template metaprogramming
- smart pointer

Use Standard Library

Every standard library is provided through some standard header.

```
#include <vector>                    // vetor container
#include <string>                    // string support

int main() {
  std::vector<int> v0{1, 2, 3};
  std::vector<std::string> v1{"string", "string"};
}
```

The standard library is defined in a namespace called `std`. To use standard library utilities, the `std::` prefix can be used.
It is generally in poor taste to dump every name from a namespace into the global namespace.

Old C Standard Library

Headers from the C standard library, such as `<stdlib.h>` are provided, albeit discouraged.
For each such header there is also a version with its name prefixed by `c` and the `.h` removed.

```
<stdio.h>  => <cstdio>
<stdlib.h> => <cstdlib>
```

This version places its declarations in the `std` namespace.

```cpp
#include <cmath>

int main() {
  std::sqrt(3);
}
```

Strings

STL <string> provides common string operations.

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

int main() {
  string s{"12345678"};
  cout << s.substr(1, 3) << endl;

  s.replace(0, 1, "abcd");
  cout << s << endl;

  cout << s + "hello" << endl;
  cout << s.c_str() << endl;
}
```

Regular Expression

STL <regex> provides regular expression support.

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

int main() {
  regex pat{"\\(\\d{3}\\) \\d{3}-\\d{4}"};
  if (regex_match("(334) 333-0610", pat))
    cout << "Match" << endl;

  pat = R"(\b(sub)([^ ]*))";
  string s{"this subject has a submarine as a subsequence"};
  for (sregex_iterator i(s.begin(), s.end(), pat);
       i != sregex_iterator{}; ++i)
    cout << i->str() << endl;
}
```

I/O Streams

The I/O stream library provides formatted and unformatted buffered I/O of text and numeric values.

- `ostream` converts typed objects to a stream of characters bytes.
- `istream` converts a stream of characters bytes to typed objects.

Output

In <ostream> , the I/O stream library defines output for every built-in type.

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

int main() {
  cout << 3 << 3.3 << 'a' << "hello" << bool << endl;
}
```

The operator «, i.e., "put to", is used as an output operator on objects of type ostream. By overloading this operator, we can serialize user-defined objects.
Predefined ostream objects:

- cout standard output stream, connects to stdout.
- cerr standard stream to report errors, connects to stderr.

Input

In `<istream>` , the I/O stream library defines input for every built-in type.

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

int main() {
  int a;
  double b;
  char c;
  cin >> a >> b >> c;
}
```

The operator », i.e., "get from", is used as an input operator. By default, a whitespace character, e.g., a space or a newline, delimits one read. `getline()` is used get a whole line, *including* the terminating newline character.

I/O State

An iostream has a state that we can examine to determine whether an operation succeeded.

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

int main() {
  int i;
  while (cin >> i) {
    cout << i << endl;
  }
}
```

In general, the I/O state holds all the information needed to read or write, such as formatting information, error state (e.g., end-of-file), and what kind of buffering is used.

I/O of User-Defined Type

```
#include <iostream>
using namespace std;
struct Point {
  int x, y;
};
ostream& operator<<(ostream& os, const Point& p) {
  os << '(' << p.x << ',' << p.y << ')';
}
istream& operator>>(istream& is, Point& p) {
  is >> p.x >> p.y;
}
int main() {
  Point p;
  cin >> p;
  cout << p << endl;
}
```

File Stream

In <fstream> , the STL provides streams to and from a file:

- `ifstream` read from a file
- `ofstream` write to a file
- `fstream` two way stream, both read from and write to a file

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

int main() {
  ofstream out("file.txt");
  ifstream in("file.txt");
}
```

The usage of `ofstream` and `ifstream` is same as `cout` and `cin`.

String Stream

In <sstream>, the STL provides streams to and from a string:

`istringstream` read from a string

`ostringstream` write to a string

`stringstream` read from and write to a string

```
#include <sstream>
#include <iostream>
using namespace std;
int main() {
  stringstream s;
  s << 3 << 'a' << "abd";
  cout << s.str() << endl;
  int a;
  char b;
  string c;
  s >> a >> b >> c;
  cout << a << b << c << endl;
}
```

Stream Consideration

1. Streams are are *type-safe*, *type-sensitive*, and *extensible*
2. It is, however, horribly *slow*.
3. Not fast and succinct as `printf` and `sprintf`.

```cpp
#include <cstdio>    // printf
#include <iostream>  // cout
#include <iomanip>   // setprecision, fixed
using namespace std;

int main() {
  cout << setprecision(2) << fixed << 1.23456 << "\n";
  printf("%.2f\n", 1.23456);
}
```

Personally, I would fall back to `printf` and `sprintf` whenever possible.

Containers

Various data structures are implemented.

- `<vector>` dynamic array
  - `<map>` dynamic sorted key-value pair
  - `<set>` mathematical set
- `<unordered_map>` hashmap
- `<unordered_set>` hashset
  - `<queue>` priority_queue, normal queue
  - `<deque>` double-linked queue
  - `<stack>` stack

Algorithms

All routines are included in <algorithm>.

- Non-modifying sequence operations, e.g., `all_of`, `any_of`, `count`, `find`, `find_first_of`, etc.
- Modifying sequence operations, e.g., `copy`, `copy_if`, `remove_if`, `swap`, `reverse`, etc.
- Partitioning operations, e.g., `partition`, `stable_partition`.
- Sorting operations, e.g., `sort`, `is_sorted`, `partial_sort`, `stable_sort`, and etc.
- Binary search operations (on sorted ranges), `lower_bound`, `upper_bound`, `binary_search`, etc.

Algorithms Cont'd

- Set operations (on sorted ranges), `merge`, `inplace_merge`, `set_difference`, etc.
- Heap operations, `is_heap`, `make_heap`, `push_heap`, `pop_heap`, `sort_heap`, etc.
- Minimum/maximum operations, `max`, `min`, `minmax`, `lexicographical_compare`, `next_permutation`, etc.

Utilities

- Dynamic memory management. Smart pointers (e.g., `unique_ptr`, `shared_ptr`), allocators (e.g., `std::allocator`).
- Pair and tuples. `pair`, `tuple` and etc.
- `bitset`, manipulate bits.

Numerics

- In <cmath>, standard mathematical functions include `sqrt()`, `log()`, `sin()` and etc. for type `float`, `double` and `long double`.
- Complex numbers, in <complex>.
- Random numbers, in <random>.
- Numeric limits, in <numeric_limits>. E.g., `std::numeric_limits<int>::max()`.

Concurrency

The standard library directly supports concurrent execution of multiple threads in a single address space.

Summary

- Core languages features
- STL libraries

For more information about C++. Online resources include

- http://cppreference.com
- http://www.cplusplus.com/