C++ for scientific computing

Geert Jan Bex

(geertjan.bex@uhasselt.be)

http://bit.ly/2P40p4L

Stay Connected to VSC

LinkedIn®

Introduction

Why C++?

- Industrial strength programming language
- General purpose
- Feature rich
  - object oriented
  - functional features
- Good standard library
- Excellent performance…
  - when used well
- However…
  - not that easy

> *Anybody who comes to you and says he has a perfect language is either naïve or a salesman.*
>
> — Bjarne Stroustrup

Scope

- Prerequisites
  - You are fluent in another programming language

This is not a training to teach you how to program!

- Limitations
  - subset of C++ most useful for scientific computation
    - data structures
    - numerics
    - data processing
    - algorithms

> *Within C++, there is a much smaller and cleaner language struggling to get out.*
>
> — Bjarne Stroustrup

Some history

- C++ created by Bjarne Stroustrup in 1983
- Many changes over the years
  - C++98: coming of age: ISO standardization
  - C++11: gets easier to use
  - C++14: fix things in C++11
  - C++17: new features
  - C++20: lots of new features, not fully supported yet
  - C++23: some new features, not fully supported yet
- Here, C++20 (a bit of C++23) + quite some STL

Presentation based on:

Bjarne Stroustrup , *A tour of C++,* Addison-Wesley, 2022

Typographical conventions

- Shell commands are rendered as

```
$ g++ -o hello.exe hello.cpp
```

  - Do *not* type `$`, it represents your shell prompt!
- Inline code fragments and file names are rendered as, e.g., `hello_world.cpp`
- Longer code fragments are rendered as

```
#include <iostream>
int main() {
    …
}
```

- Data files are rendered as

```
case dim temp
1 1 -0.5
2 1 0.0
…
```

fragment not shown

Syntax versus semantics

- syntax: form, grammar
    - correct:
      The dog is barking.
    - incorrect:
      The dog barking.
- semantics: meaning, interpretation
    - correct:
      The dog barked.
    - incorrect:
      The dog spoke.

Except in fairy tales!

Basic language features

Chapter 1, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Basics

Hello world

- (Almost) minimal C++ program: `hello.cpp`

```
#include <iostream>

int main(int argc, char *argv[]) {
    std::cout << "hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

hello.cpp

- Compile & link

```
$  g++  -std=c++14  -Wall  -g  -o hello.exe  hello.cpp
```

- Run

```
$  ./hello.exe  world
hello world!
```

Anatomy of hello world

- Include declarations of (standard) libraries

```
#include <iostream>
…
```

required for I/O

- `main` function definition

```
…
int main(int argc, char *argv[]) {
    …
}
```

Application has exactly one `main` function

- Statements in function body

```
…
    std::cout << "hello world!" << std::endl;
    return 0;
…
```

program's exit code

Namespaces

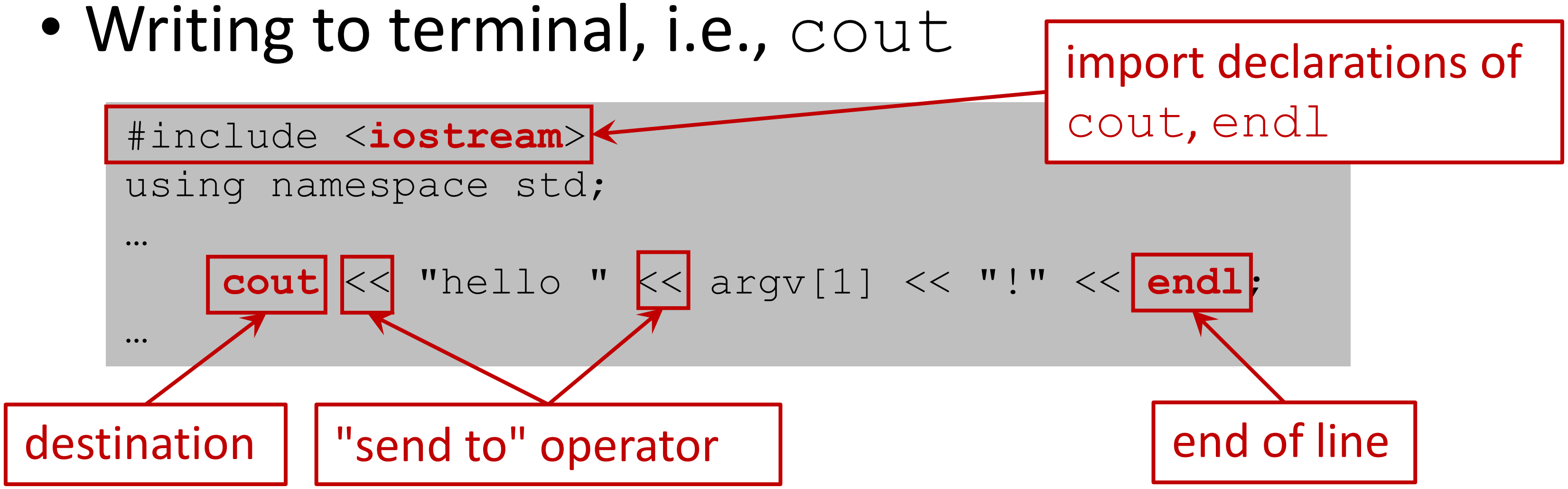
- Avoid name conflicts
  - functions/variables with same name in multiple contexts
- E.g., standard library in namespace `std`
  - `iostream: cout`, `endl`, …
- Either
  - prefix with namespace, e.g., `std::cout`, or
  - use namespace

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
    cout << "hello " << argv[1] << "!" << endl;
    return 0;
}
```

assumed in slides

Getting things out

- Writing to terminal, i.e., `cout`



```
#include <iostream>
using namespace std;
…
    cout << "hello " << argv[1] << "!" << endl;
…
```

import declarations of `cout`, `endl`

destination

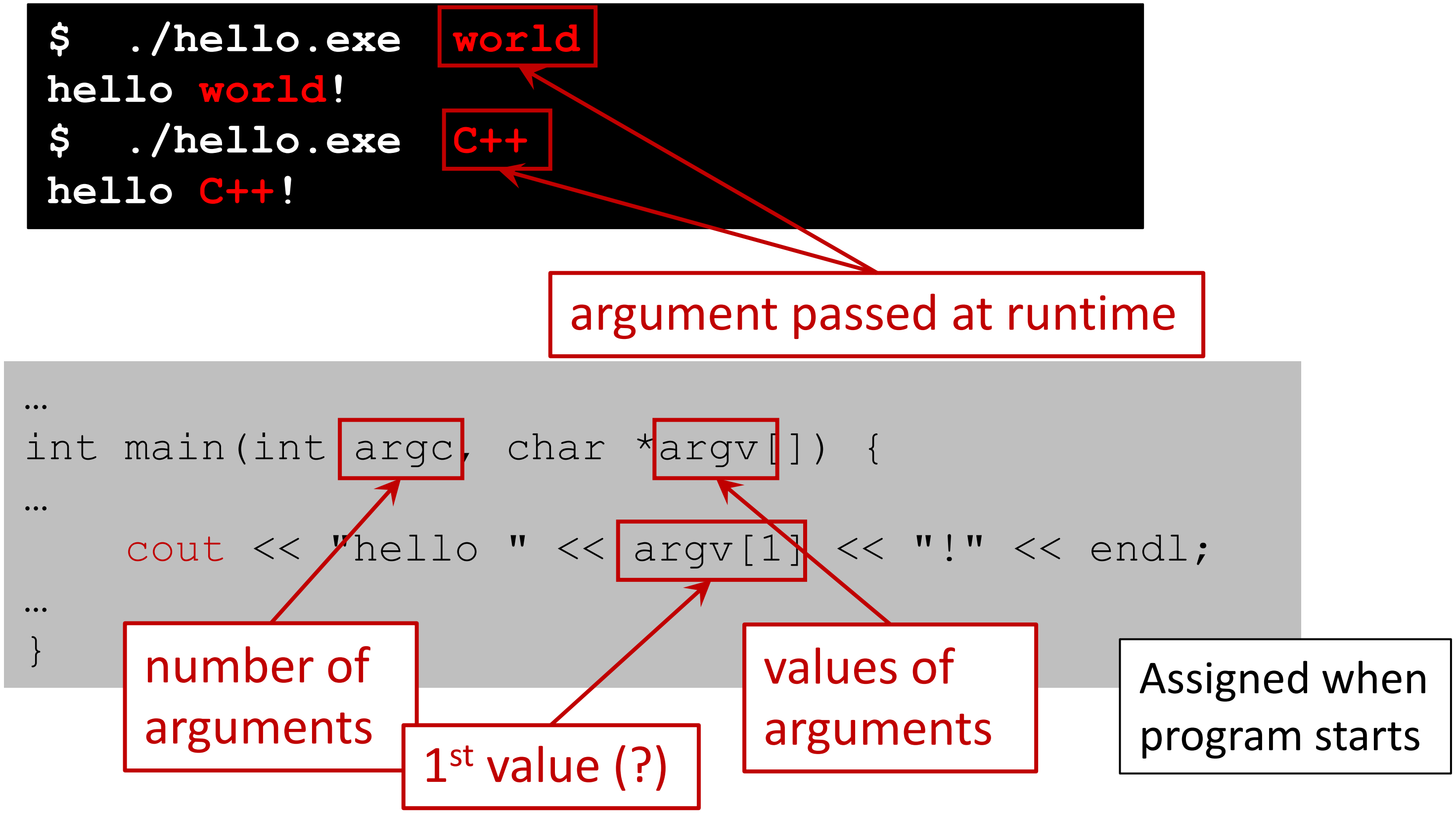
"send to" operator

end of line

`"hello "`: string constant, i.e., text

Getting things in

- Command line arguments



```
$  ./hello.exe  world
hello world!
$  ./hello.exe  C++
hello C++!
```

argument passed at runtime

```
…
int main(int argc, char *argv[]) {
…
    cout << "hello " << argv[1] << "!" << endl;
…
}
```

number of arguments

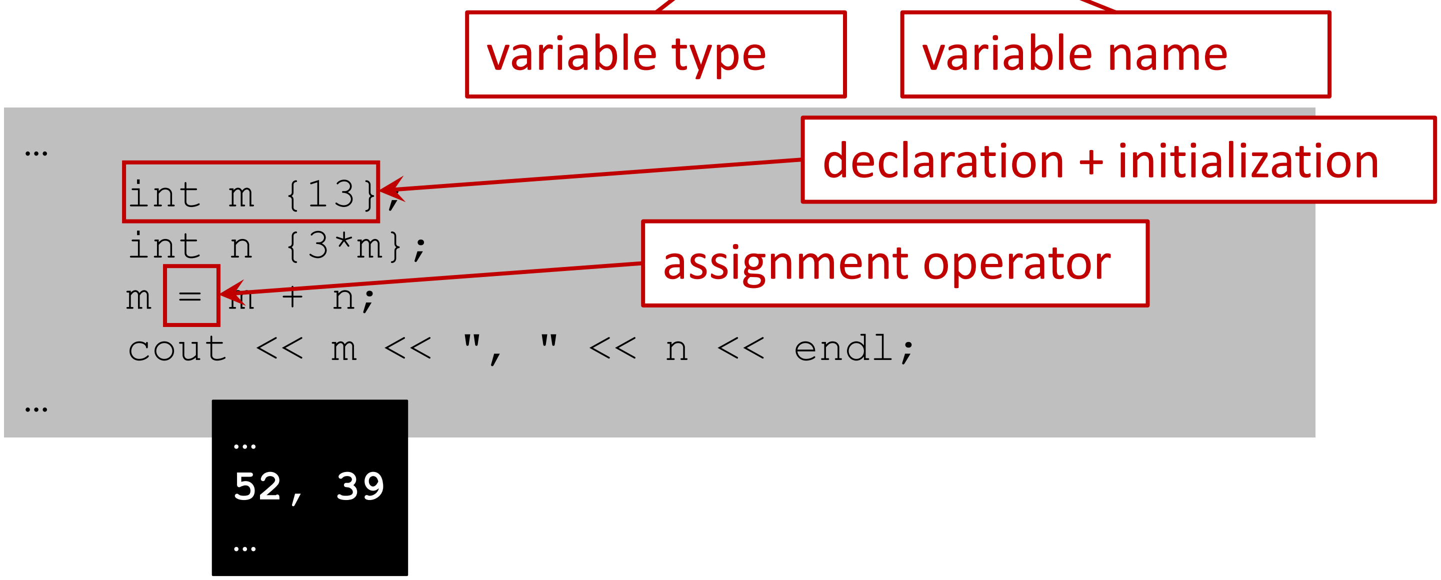
1st value (?)

values of arguments

Assigned when program starts

Variables

- Names for values in memory (RAM)
- Names start with letter or _, can contain digits
- Value can change during run
- Must be declared, i.e., `int argc`



variable type

variable name

declaration + initialization

assignment operator

```
…
     int m {13};
     int n {3*m};
     m = m + n;
     cout << m << ", " << n << endl;
…
```

```
…
52, 39
…
```

Types

- `char`: character, e.g., `'a'`, `'7'`, `'\n'`
- `std::string`: character sequence, e.g., `"hello"`, `""`
- `int`: integer number, e.g., `7`, `-15`, `1034`
- `float`: single precision floating point number, e.g., `7.0f`, `-0.531f`, `1.37e-3f`
  - 4 byte representation
  - 7 significant digits, smallest non-zero ~ 10^{-38}
  - range ~ $[-10^{38}, 10^{38}]$
- `double`: double precision floating point number, e.g., `7.0`, `-0.531`, `1.37e-3`
  - 8 byte representation
  - 15 significant digits , smallest non-zero ~ 10^{-308}
  - range ~ $[-10^{308}, 10^{308}]$
- `bool`: Boolean value, i.e., `true`, `false`

Operators & math functions

Note: `3/5 == 0`

- `int`, `float`, `double`: `+`, **–**, `*`, `/`
- `int`: `%` (modulo)
- `bool`: `&&` (and), `||` (or), `!` (not)
- Comparison
  - `char`, `string`, `int`: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - `float`, `double`: `<`, `<=`, `>`, `>=` and `==`, `!=` (???)

```
1.0/3.0 =?= 0.33333333333333
```

- Mathematical functions
  - `#include <cmath>`
  - e.g., `sin`, `cos`, `tan`, `exp`, `log`, `sqrt`,…

Assignment shortcuts

- Syntactic sugar
  - `x = x + y ≡ x += y`
  - `x = x – y ≡ x -= y`
  - `x = a*x ≡ x *= a`
  - …
  - `n = n + 1 ≡ n++`
  - `n = n – 1 ≡ n--`
- Post-increment/decrement

```
int m {3}, n {5};
m += n++;
cout << m << " " << n…;
```

→

```
8 6
```

- Pre-increment/decrement

```
int m {3}, n {5};
m += ++n;
cout << m << " " << n…;
```

→

```
9 6
```

General remarks

- C++ is case sensitive
  - language keywords
  - variable, function, class names
- Statements end with `;`
- Comments
  - single-line comment

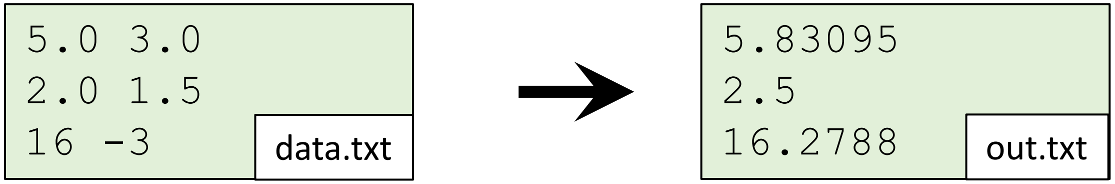
```
int n {10};  // this is a comment
```

  - block comment

```
/*
 This is a
 multi-line
 comment.
*/
```

Task: data transformation

- File `data.txt` contains coordinates in 2D, compute distance from origin, write to `out.txt`



```
5.0 3.0
2.0 1.5
16 -3
```

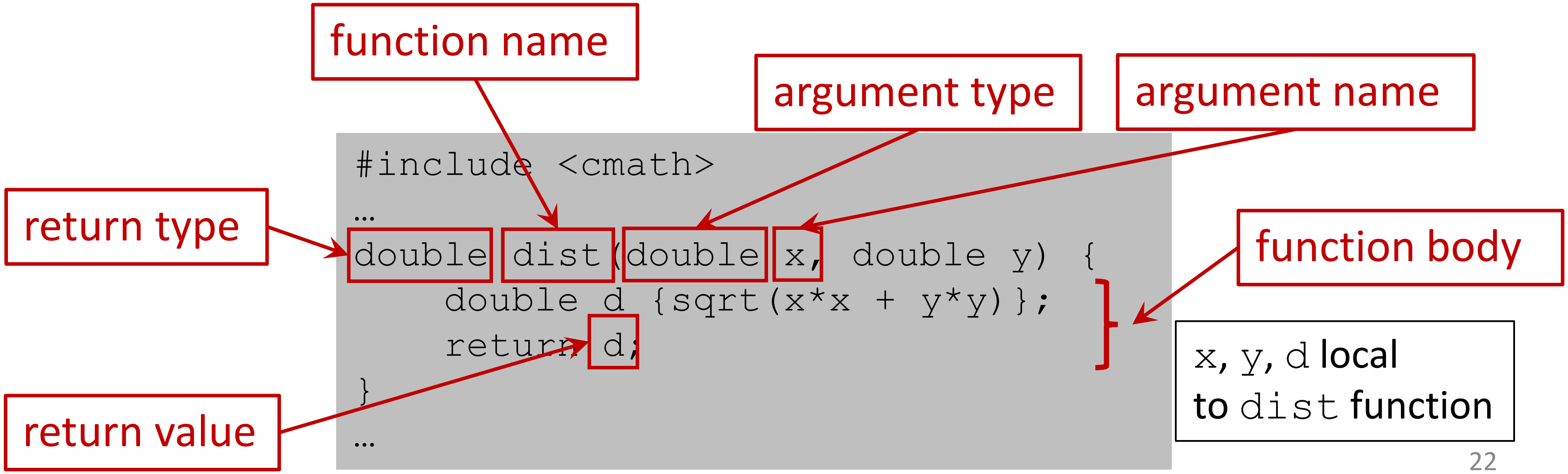
data.txt

```
5.83095
2.5
16.2788
```

out.txt

Functions

- Function signature = declaration
    - name (same rules as for variables)
    - argument types and names (zero or more)
    - return type
- Function implementation: statements in body



```
#include <cmath>
…
double dist(double x, double y) {
    double d {sqrt(x*x + y*y)};
    return d;
}
…
```

function name

argument type

argument name

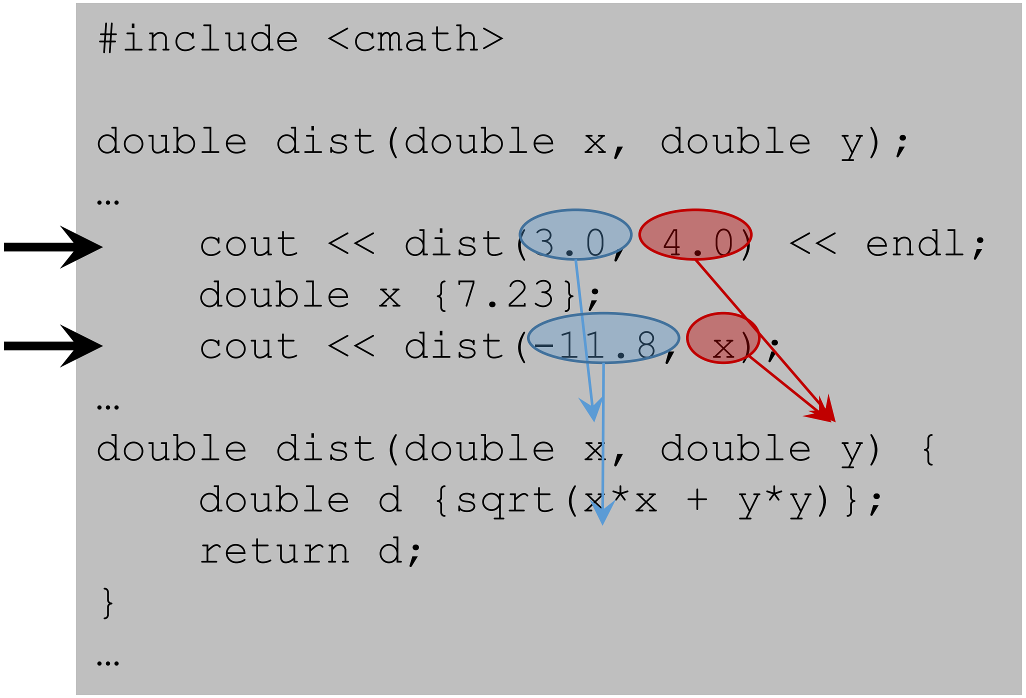
return type

function body

return value

x, y, d local to dist function

Function calls



```
#include <cmath>

double dist(double x, double y);
…
    cout << dist(3.0, 4.0) << endl;
    double x {7.23};
    cout << dist(-11.8, x);
…
double dist(double x, double y) {
    double d {sqrt(x*x + y*y)};
    return d;
}
…
```

- Function arguments assigned at function call
- Cfr. mathematical functions

Call by value versus reference

- Call by value

```
…
    int n {5};
    cout << fac(n) …;
    cout << n …;
…
int fac(int n) {
    int result {1};
    while (n >= 2) {
        result *= n;
        n = n - 1;
    }
    return result;
}
```

5

- Call by reference

```
…
    int a {3};
    int b {5};
    swap(a, b);
    cout << a << ", " << b …;
…
void swap(int& x, int& y) {
    int tmp {x};
    x = y;
    y = tmp;
}
```

5 3

reference to `int`

- Modifications in callee

- Modifications in callee *and* in caller

Overloading

- Functions with same name but at least one distinct argument type

```
…
    int a {3};
    int b {5};
    swap(a, b);
    cout << a << ", " << b …;
…
void swap(int& x, int& y) {
    int tmp {x};
    x = y;
    y = tmp;
}
```

```
…
    double x {3.5};
    double y {5.7};
    swap(x, y);
    cout << x << ", " << y …;
…
void swap(double& x, double& y) {
    double tmp {x};
    x = y;
    y = tmp;
}
```

However: generic programming, see later

Recursion

- Function can call itself

$$n! = \begin{cases} 1 \text{ if } n = 0 \text{ or } n = 1 \\ n \cdot (n-1)! \end{cases}$$

```
int fac(int n) {
    if (n < 2) {
        return 1;
    } else {
        return n*fac(n - 1);
    }
}
```

termination condition

recursive call

Data in, results out

```cpp
#include <iostream>
#include <cmath>

using namespace std;

double dist(double x, double y) {
    return sqrt(x*x + y*y);
}

int main() {
    double a, b;
    while (cin >> a >> b) {
        cout << dist(a, b) << endl;
    }
    return 0;
}
```

```
$  ./dist.exe  < data.txt
5.83095
2.5
16.2788
```

```
$  ./dist.exe  < data.txt  > out.txt
```

I/O streams

- Reading from

  Operator >>

  - standard input: `cin` (via keyboard, I/O redirection)
  - files (see later)
- Writing to

  Operator <<

  - standard output: `cout` (to screen, I/O redirection)
  - standard error: `cerr` (to screen, I/O redirection)
  - files (see later)

```
#include <iostream>
…
    double a, b;
    while (cin >> a >> b) {
        cout << dist(a, b) << endl;
    }
…
```

automatic conversion `string` to `double`

automatic conversion `double` to `string`

I/O operator semantics

- Read string representation of `double` from standard input, assign to variable `a`, read string representation of `double` from standard input, assign to variable `b`, `true` on success, `false` otherwise. Whitespace is separator.

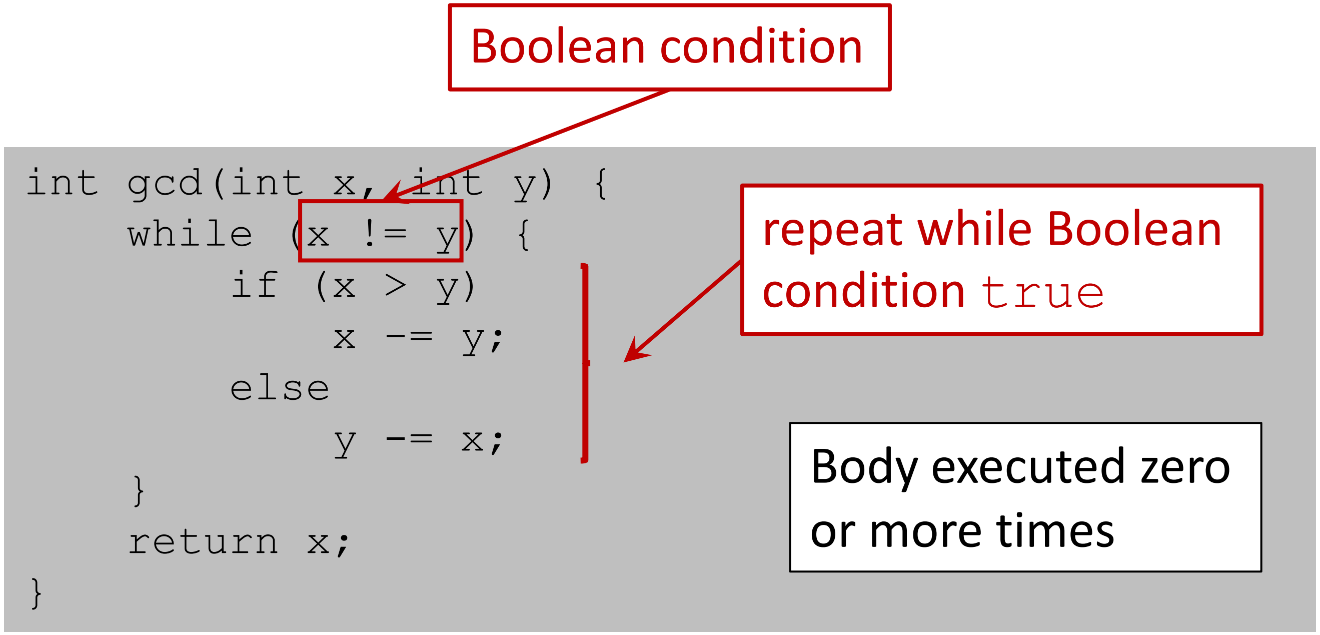
```
double a, b;
… cin >> a >> b …
```

- Convert `double`, i.e., return value of `dist` call to string representation, and write to standard output, write end-of-line to standard output (`'\n'` on Linux/MacOS X, `'\r'` + `'\n'` on Windows).

```
cout << dist(a, b) << endl;
```

While statement

- Greatest common divisor (GCD) of `x` and `y`



Boolean condition

```
int gcd(int x, int y) {
    while (x != y) {
        if (x > y)
            x -= y;
        else
            y -= x;
    }
    return x;
}
```

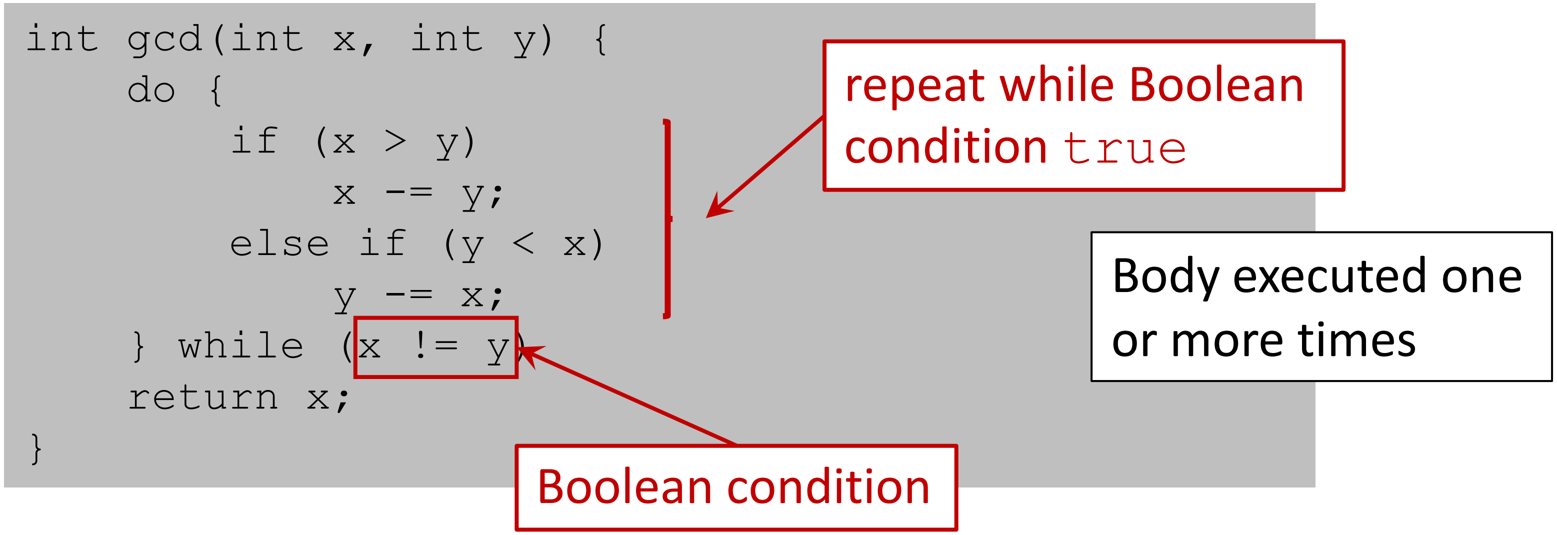
repeat while Boolean condition `true`

Body executed zero or more times

- Repetition statement

Do-while statement

- Alternative to while
- Less frequently used



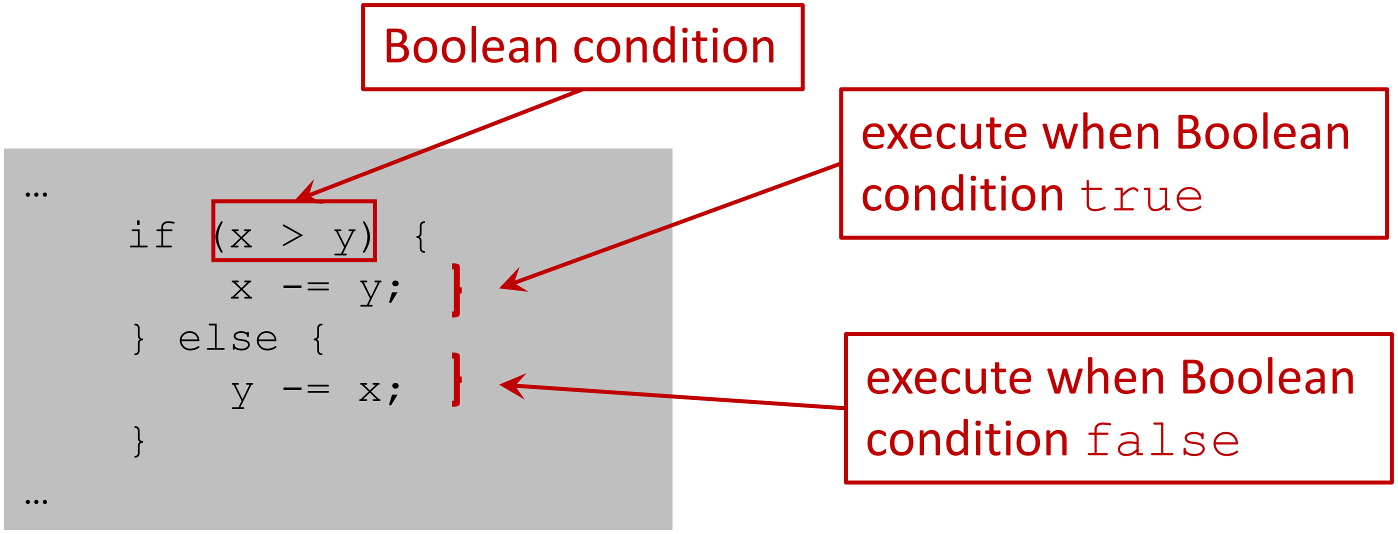
```
int gcd(int x, int y) {
    do {
        if (x > y)
            x -= y;
        else if (y < x)
            y -= x;
    } while (x != y);
    return x;
}
```

repeat while Boolean condition `true`

Body executed one or more times

Boolean condition

If statement



```
…
     if (x > y) {
          x -= y;
     } else {
          y -= x;
     }
…
```

Boolean condition

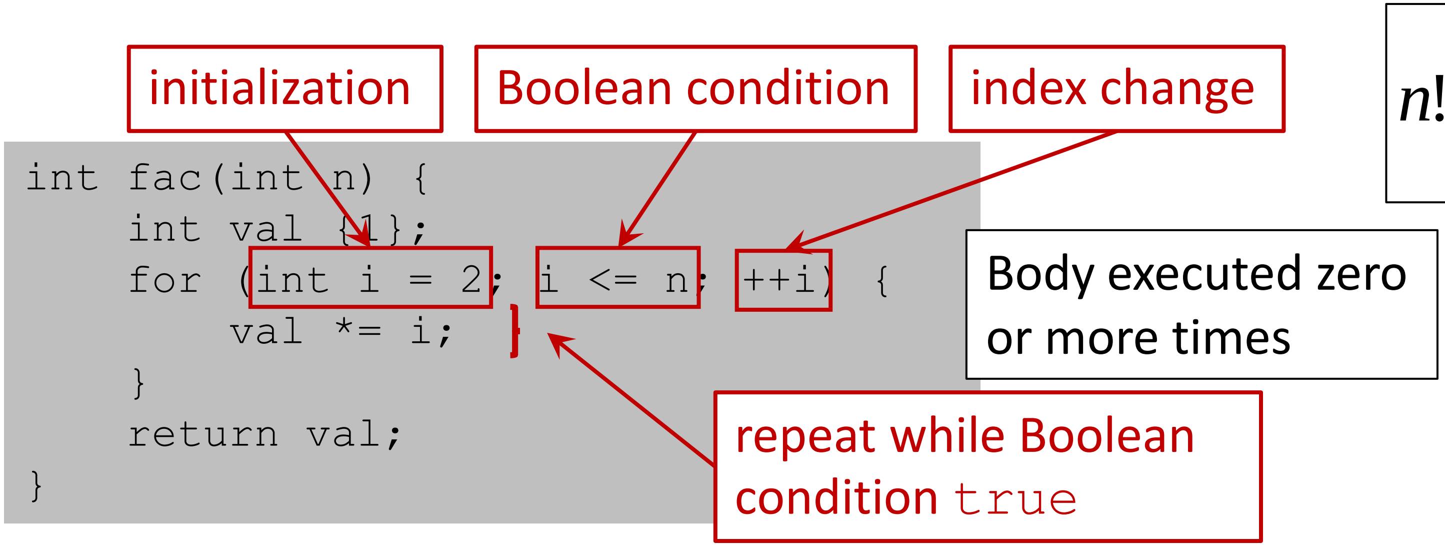
execute when Boolean condition `true`

execute when Boolean condition `false`

- `else`-clause is optional
- Can be chained
- Conditional statement

```
if (…) {
     …
} else if (…) {
     …
} else {
     …
}
```

For statement



```
int fac(int n) {
    int val {1};
    for (int i = 2; i <= n; ++i) {
        val *= i;
    }
    return val;
}
```

initialization

Boolean condition

index change

Body executed zero or more times

repeat while Boolean condition `true`

$$n! = \prod_{i=1}^{n} i$$

- Initialization once, before first iteration: `i = 2`
- Condition check before each iteration: `i <= n`
    - if true, body executed
    - index modified after iteration: `++i`
- Repetition statement

Break & continue statements

- Interrupt repetition statement

```
cout << "Name?" << endl;
while (cin >> name) {
    if (name == "quit")
        break;
    cout << "Hi " << name << "!" << endl;
}
cout << "Bye" << endl;
```

- Interrupt current iteration, start next one

```
std::string line;
double sum {0.0};
while (std::getline(std::cin, line)) {
    if (line[0] == '#') continue;
    sum += std::stof(line);
}
std::cout << "sum = " << sum << std::endl;
```

Blocks

- Blocks: one or more statements
- Enclosed in { … }
- Defines scope

Don't do this: confusing!

```
…
int i {3};
cout << i << endl;
{
    cout << i << endl;
    int i {5};
    cout << i << endl;
}
cout << i << endl;
for (int i = 7; i < 10; ++i)
    cout << i << " ";
cout << i << endl;
…
```

```
3
3
5
3
7 8 9
3
```

Arrays

- Contiguous data storage in memory, fixed size
- Homogeneous types

number of elements

```
…
    double v[5];
    for (int i = 0; i < 5; ++i)
        v[i] = static_cast<double>(i);
    cout << sum_array(v, 5) << endl;
…
double sum_array(double v[], int n) {
    double result {0.0};
    for (int i = 0; i < n; ++i) {
        result += v[i];
    }
    return result;
}
```

v[0]	v[1]	v[2]	v[3]	v[4]
0.0	1.0	2.0	3.0	4.0

0-based indexing!

Alternative(?): STL `std::vector`, see later

Constants

value of n can not change

```
…
     const int n {10};
     double v[n];
     cout << sum_array(v, n) << endl;
     n = 5;
…
double sum_array(const double v[], int n) {
     double result {0.0};
     for (int i = 0; i < n; ++i) {
          result += v[i];
          v[i] = 0.0;
     }
     return result;
}
```

compile error!!!

array values in v can not change

compile error!!!

User defined types

Chapter 2, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/UserDefinedTypes

Data types revisited

- Integers
  - `int`, `long`
    `unsigned int`, `unsigned long`
- More portable integers:
  `int8_t`, `int16_t`, `int32_t`, `int64_t`
  `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
  `size_t`

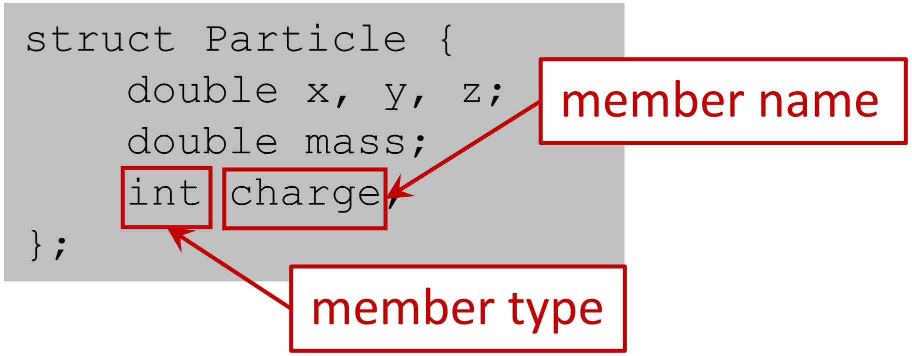
in `cstdint`

- Real numbers
  - `float`
  - `double`
- Vectors, matrices
  - arrays, better `std::array`, `std::valarray`, `std::vector`

Mathematical modelling

Defining structures

- Representing tuples
- Define new type, specify name, members



```
struct Particle {
    double x, y, z;
    double mass;
    int charge;
};
```

member name

member type

- Members can have distinct types

Using structures

- Variable declaration

```
Particle p1;
Particle p2 {
    3.0,  // x
    …
    0.5,  // mass
    1     // charge
};
```

members not initialized!

members initialization

- Using variables

```
…
p1.x = -2.0;
cout << p2.mass;
…
```

Passing structures to functions

- Pass by value copies, *not* what you want

```
…
double dist(const Particle& p1, const Particle& p2) {
    return sqrt(sqr(p1.x - p2.x) + sqr(p1.y - p2.y) +
                sqr(p1.z - p2.z));
}
```

```
…
void move(Particle& p, double dx, double dy, double dz) {
    p.x += dx;
    p.y += dy;
    p.z += dz;
}
```

Note: function doesn't return value

Structures versus classes

- Advantages of structures/classes
  - easy to use
  - good fit for modelling
- Structures
  - Members/methods are public by default
    - members can be modified inadvertently
- Classes
  - Members/methods are private by default
    - inspectors/mutators are defined

Object attributes

- Can be private

```
class Particle {
    private:
        double x_, y_, z_;
        double mass_;
…
};
```

- Can only be accessed (read/write) from within class
- Can also be public
- Determine state of object

Object methods

- Can be public

```
class Particle {
    …
    public:
        …
        double x() const { return x_; }
        …
        double mass() const {return mass_; }
        void move(double dx, double dy, double dz);
        …
};
```

definition for inspector of x_ attribute

declaration of mutator for x_, y_, z_

- Is called on instance
- Can also be private

Constructor

- Can be public

```
class Particle {
    …
    public:
        …
        Particle(double x, double y, double z,
                 double mass, int charge) :
            x_ {x}, y_ {y}, z_ {z},
            mass_ {mass}, charge_ {charge} {};
        …
};
```

empty method body

trivial attribute initialization

- Creates new instance
- Can also be private (factories, …)

Method types

- Constructor(s)
  - creates new object (instance) of class
- Inspectors
  - retrieve state information of object
  - doesn't change state of object
- Mutators
  - changes state of object
- Destructor
  - releases resources acquired by object

Method implementation

- When trivial, in class definition
  - e.g., `x` inspector, …, `Particle` constructor
- Otherwise, outside class definition
  - e.g., `move` mutator

class name

```
void Particle::move(double dx, double dy, double dz) {
    x_ += dx;
    y_ += dy;
    z_ += dz;
}
```

object attributes

Using class and objects

- Constructing a new `Particle` object

```
…
Particle p(0.3, 0.5, 0.7, 1.0, -1);
…
```

- Calling inspectors

```
…
cout << "(" << p.x() << ", " << p.y() << …;
…
```

- Calling mutator

```
…
p.move(0.5, 0.5, 0.5);
…
```

Another method

- Declaration

```
class Particle {
    …
    public:
        …
        double dist(const Particle& other) const;
};
```

other will not change

object will not change

- Implementation

```
inline double sqr(double x) { return x*x; }
double Particle::dist(const Particle& other) const {
    return sqrt(sqr(x_ - other.x()) +
                sqr(y_ - other.y()) +
                sqr(z_ - other.z()));
}
```

Could use other.x_, but other.x() is better

- Use

```
double r {p1.dist(p2)};
```

Interlude: function inlining

- Many functions
  - improve code quality, easier to understand
  - but calls may have performance impact
- Solution: inline
  - explicitly declared: inline keyword (advise to compiler)
  - automatically by compiler

```
inline double sqr(double x) { return x*x; }
double Particle::dist(const Particle& other) const {
    return sqrt((x_ - other.x())*(x_ - other.x()) +
                (y_ - other.y())*(y_ - other.y()) +
                (z_ - other.z())*(z_ - other.z()));
}
```

substitution at compile time

Enum class

- Examples
    - charge: positive, neutral, negative
    - color: magenta, cyan, yellow, black

```
enum class Charge {negative, neutral, positive};

int charge_value(Charge charge) {
    switch (charge)
        case Charge::negative:
            return -1;
        case Charge::neutral:
            return 0;
        case Charge::positive:
            return 1;
    }
}
```

enum class definition

Interlude: switch

- Conditional statement
  - *only* for scalar types (`int`, `char`, `enum classes`)

```
char op;
double result, a, b;
…
switch (op) {
    case '+':
        result = a + b;
        break;
    case '-':
        result = a - b;
        break;
    …
    default:
        // error
}
```

better performance

```
char op;
double result, a, b;
…
if (op == '+') {
    result = a + b;
else if (op == '-') {
    result = a - b;
} … {
    …
} else {
     // error
}
```

more versatile

What was left out?

- `union` data type
  - use `std::variant` (C++17) instead
  - not so relevant for scientific computing

Separate compilation

Chapter 3, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Modularity

Motivation

- Large files
  - difficult to maintain
  - discourage reuse
- Small files
  - files have single concern
  - can be compiled separately
- Header files (`.h`)
  - declarations
  - very short definitions (one liners)
  - (typically) used from various `.cpp` files

Class declaration: header file

particle.h

```
class Particle {
    private:
        double x_, y_, z_;
        double mass_;
    public:
        Particle(double x, double y, double z,
                 double mass) :
            x_ {x}, y_ {y}, z_ {z}, mass_ {mass} {};
        double x() const { return x_; };
        double y() const { return y_; };
        double z() const { return z_; };
        double mass() const {return mass_; }
        void move(double dx, double dy, double dz);
        double dist(const Particle& other) const;
};
```

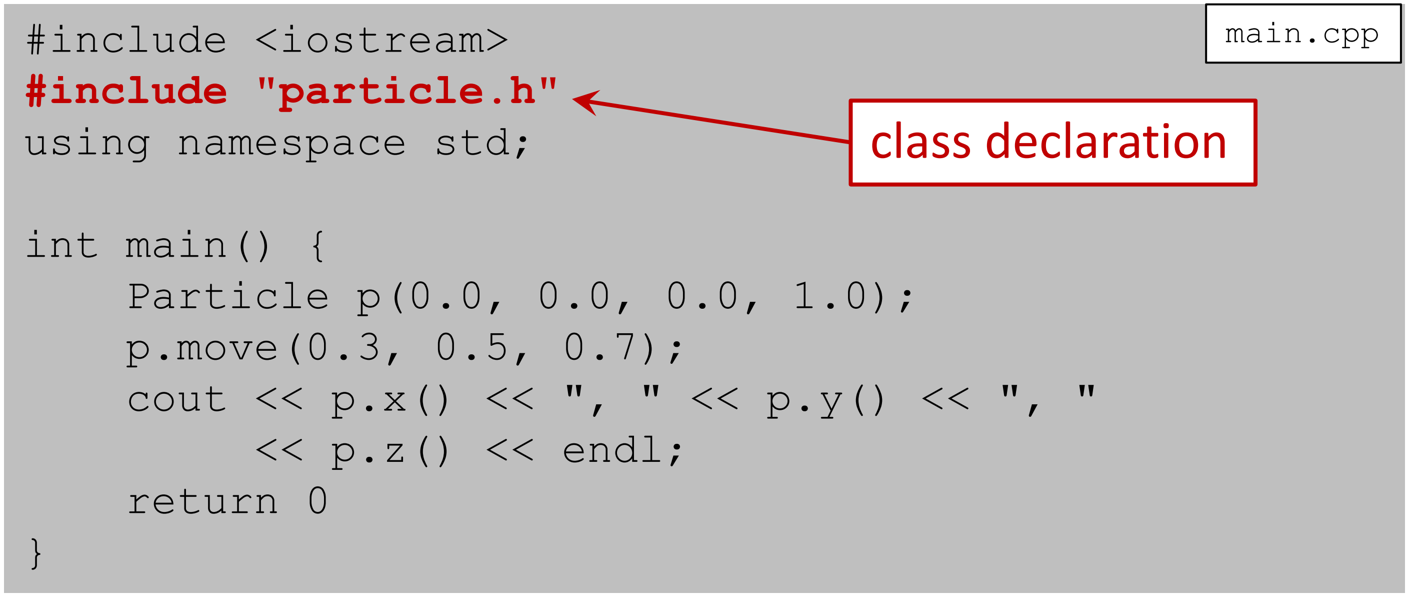
Class methods definition

particle.cpp

```
#include <cmath>
#include "particle.h"
using namespace std;

inline double sqr(double x) { return x*x; }

void Particle::move(double dx, double dy, double dz) {
    x_ += dx;
    y_ += dy;
    z_ += dz;
}

double Particle::dist(const Particle& other) const {
    return sqrt(sqr(x_ - other.x()) +
                sqr(y_ - other.y()) +
                sqr(z_ - other.z()));
}
```

class declaration

Using the class



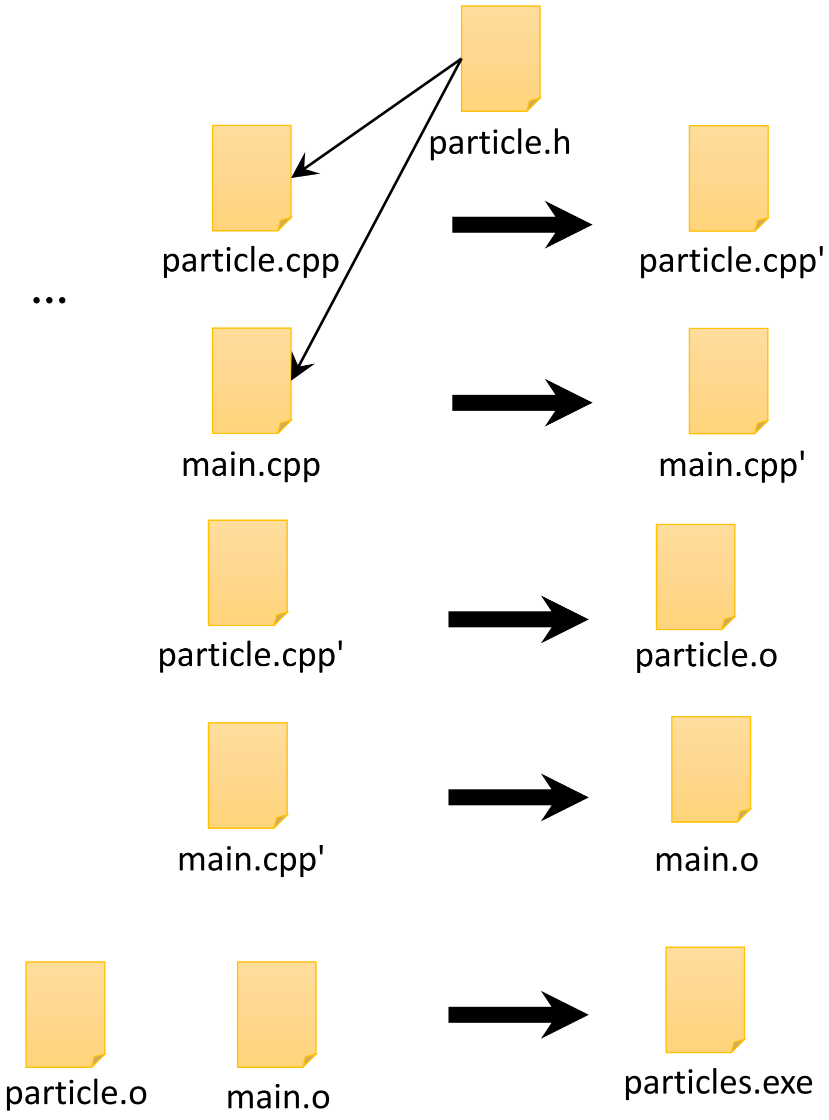
main.cpp

```
#include <iostream>
#include "particle.h"
using namespace std;

int main() {
    Particle p(0.0, 0.0, 0.0, 1.0);
    p.move(0.3, 0.5, 0.7);
    cout << p.x() << ", " << p.y() << ", "
         << p.z() << endl;
    return 0
}
```

class declaration

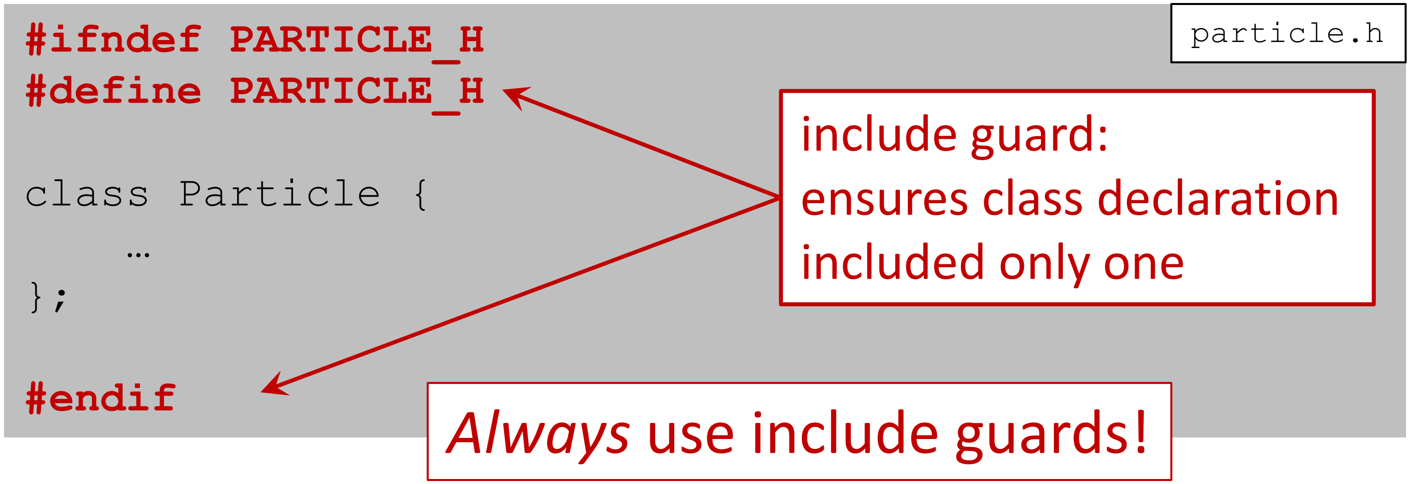
Build process

- Preprocessing
  - processes, e.g., `#include` …
  - called by compiler
- Compilation
  - create object file
- Linking
  - create executable



Preprocessor language

- Defines "programming language"
  - `#include file`: include file
  - `#define cname`: define constant
  - `#define cname val`: assign value to constant
  - `#ifdef cname … #endif`: include if defined
  - `#ifndef cname … #endif`: include unless defined



particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H

class Particle {
    …
};

#endif
```

include guard:
ensures class declaration
included only one

Always use include guards!

Preprocessor macros

- Literal substitution in source code
  - constants

```
#define NR_DIM 3
…
double coords[NR_DIM]
…
```

→ cpp

```
…
double coords[3]
…
```

  - macros

```
…
double vars[2*n];
#define x(i) vars[(i)]
#define y(i) vars[(i) + n]
…
d = sqrt(x(1)*x(1) +
         y(1)*y(1));
…
```

→ cpp

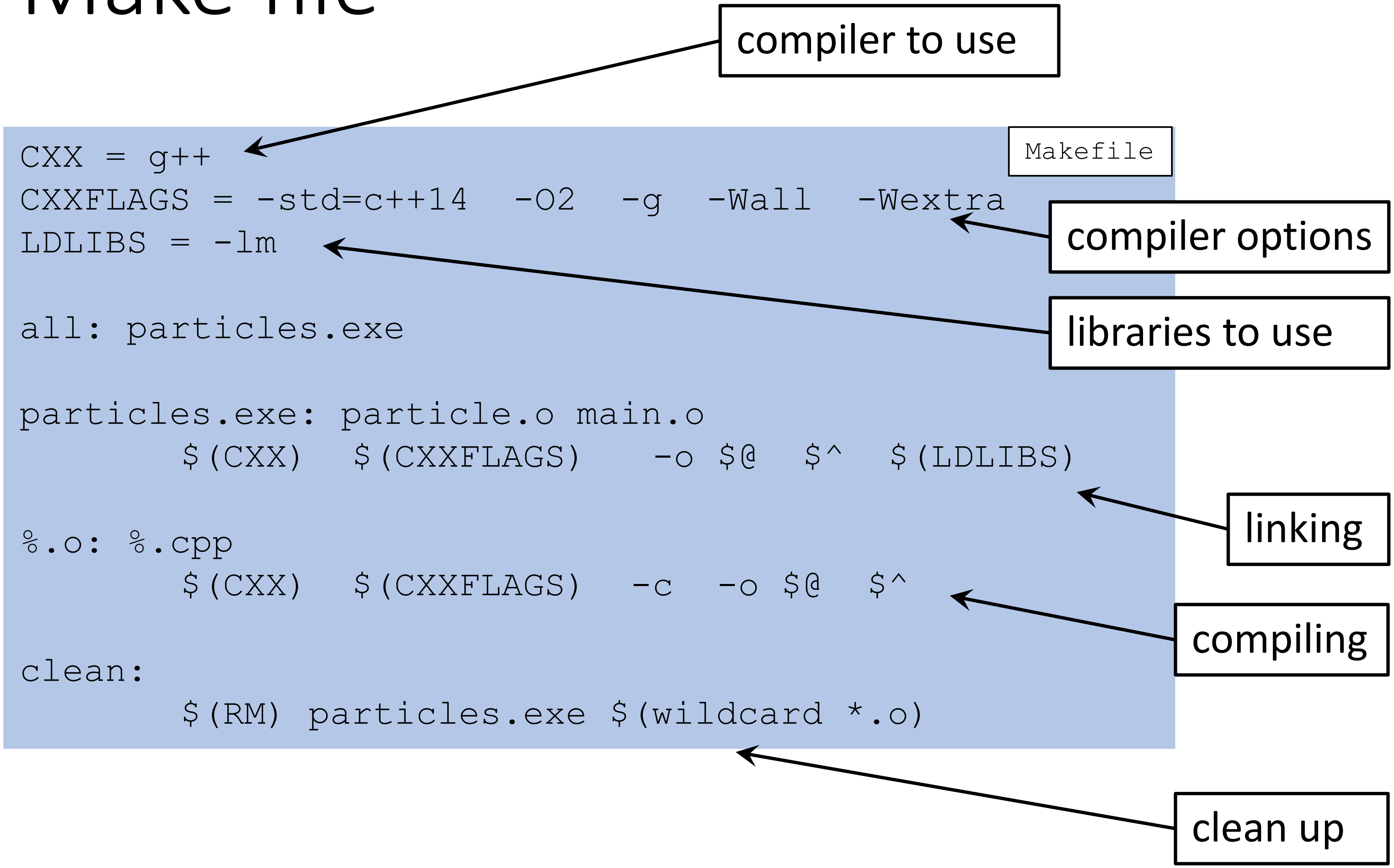
```
…
double vars[2*n];


…
d = sqrt(vars[(1)]*vars[(1)] +
         vars[(1)+n]*vars[(1)+n]);
```

Do *not* overuse!

Make files

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Modularity

Make file



Makefile

```
CXX = g++
CXXFLAGS = -std=c++14  -O2  -g  -Wall  -Wextra
LDLIBS = -lm

all: particles.exe

particles.exe: particle.o main.o
	$(CXX)  $(CXXFLAGS)   -o $@  $^  $(LDLIBS)

%.o: %.cpp
	$(CXX)  $(CXXFLAGS)  -c  -o $@  $^

clean:
	$(RM) particles.exe $(wildcard *.o)
```

compiler to use

compiler options

libraries to use

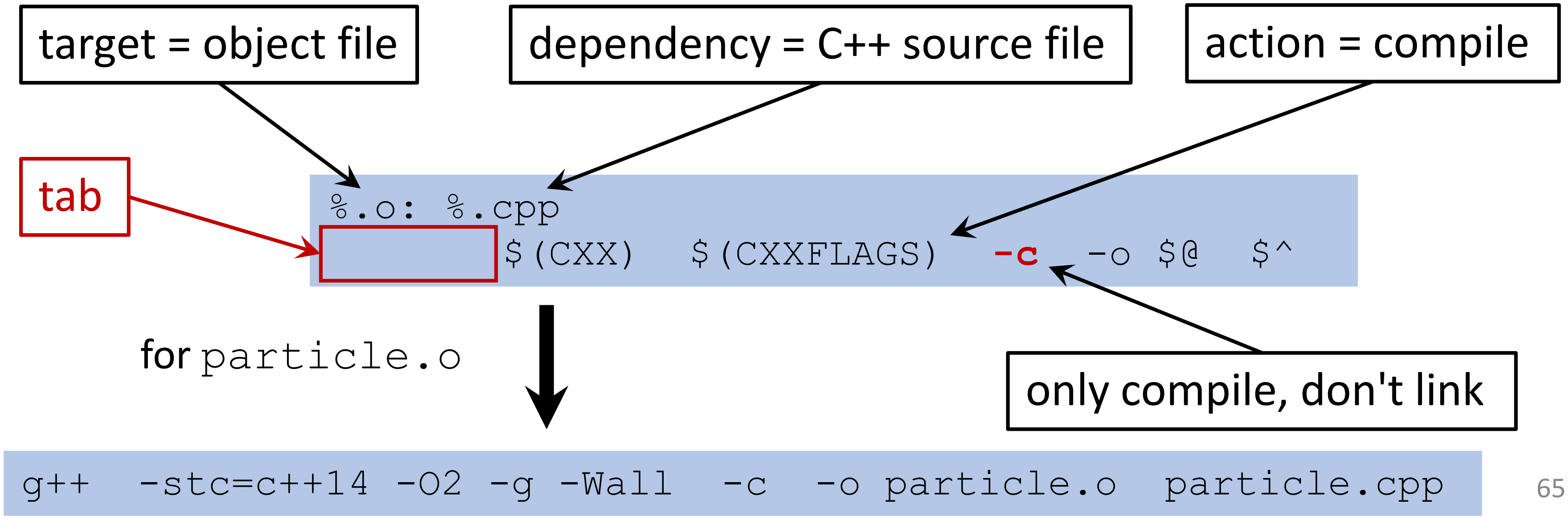
linking

compiling

clean up

Make rule

- Recipe
  - target: what to make
  - dependency: what artifacts are required
  - action: how to do it
- E.g., how to create object files?



target = object file

dependency = C++ source file

action = compile

tab

```
%.o: %.cpp
	$(CXX)  $(CXXFLAGS)  -c  -o $@  $^
```

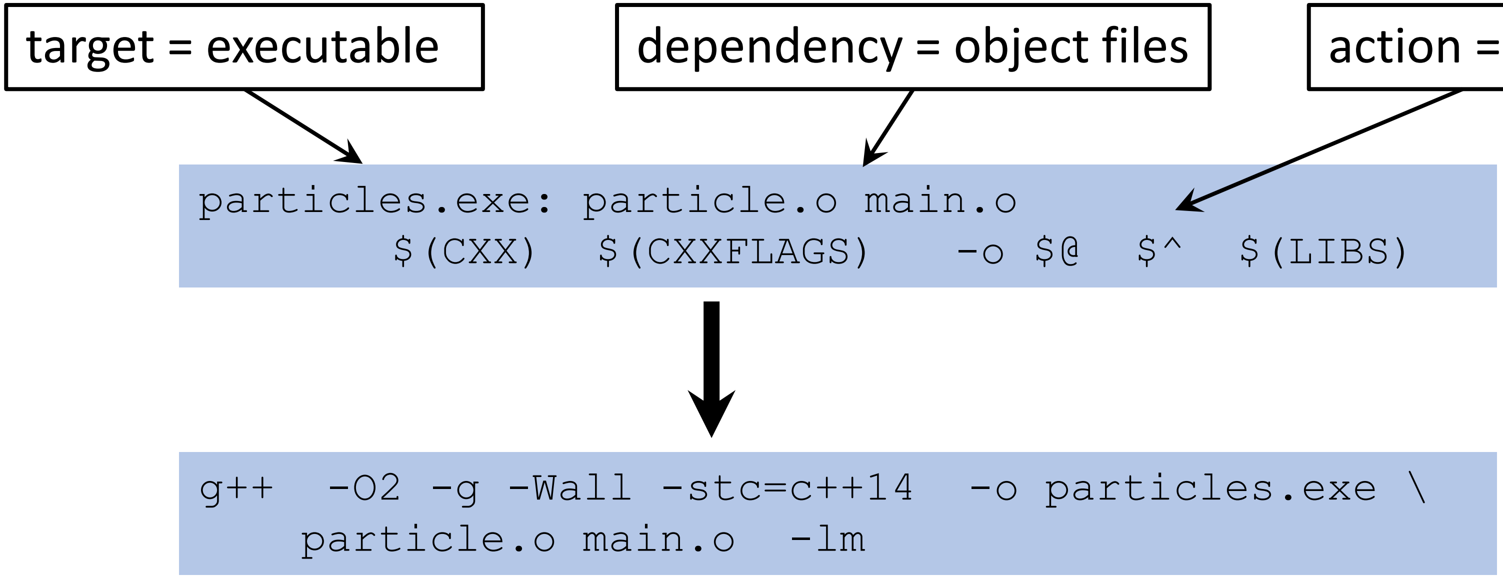
only compile, don't link

for `particle.o`

```
g++  -stc=c++14 -O2 -g -Wall  -c  -o particle.o  particle.cpp
```

More rules

- Linking



target = executable

dependency = object files

action = link

```
particles.exe: particle.o main.o
        $(CXX)  $(CXXFLAGS)   -o $@  $^  $(LIBS)
```

```
g++  -O2 -g -Wall -stc=c++14  -o particles.exe \
    particle.o main.o  -lm
```

- Default target

```
all: particles.exe
```

Using make

- Build executable

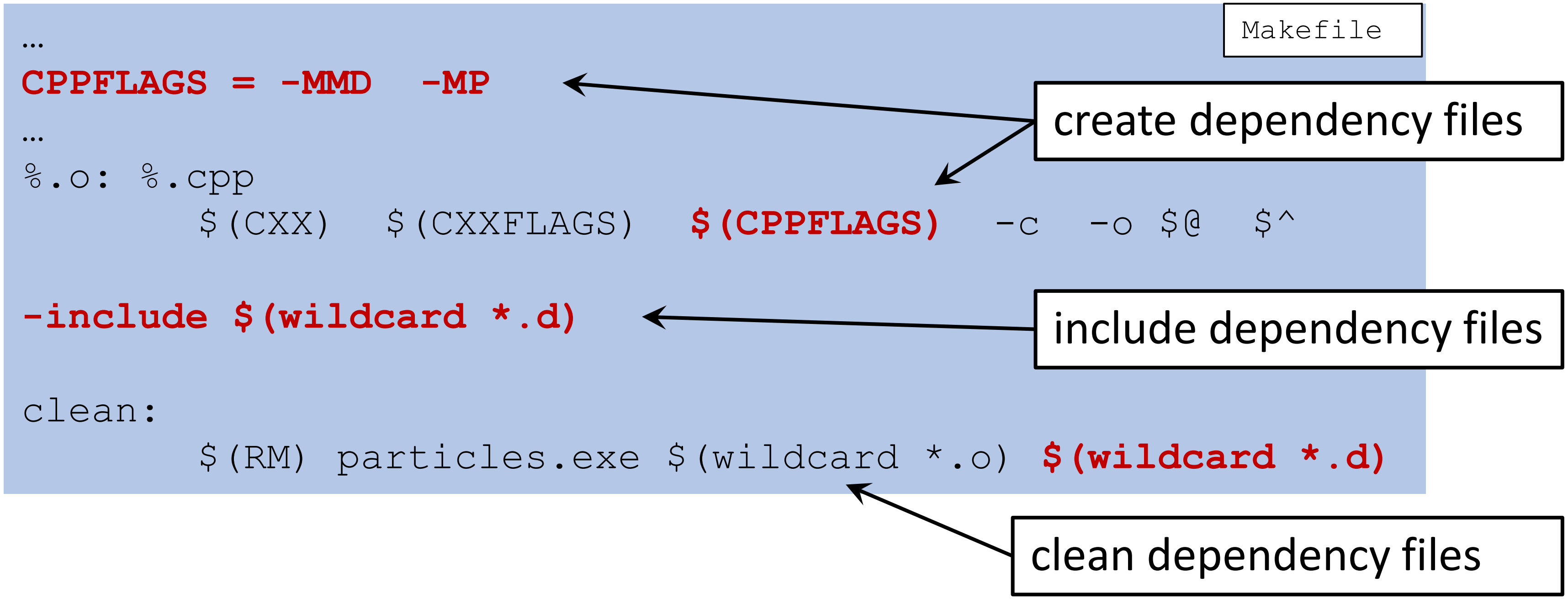
```
$  make
```

- Only execute targets with modified dependencies
  - dependency tracking
  - saves lots of time on large projects
- Clean all build artifacts

```
$  make  clean
```

Dependencies

- C++ dependencies on header files can be non-trivial
  - weird errors
- Can be tracked automatically



Makefile

```
…
CPPFLAGS = -MMD  -MP
…
%.o: %.cpp
        $(CXX)  $(CXXFLAGS)  $(CPPFLAGS)  -c  -o $@  $^

-include $(wildcard *.d)

clean:
        $(RM) particles.exe $(wildcard *.o) $(wildcard *.d)
```

create dependency files

include dependency files

clean dependency files

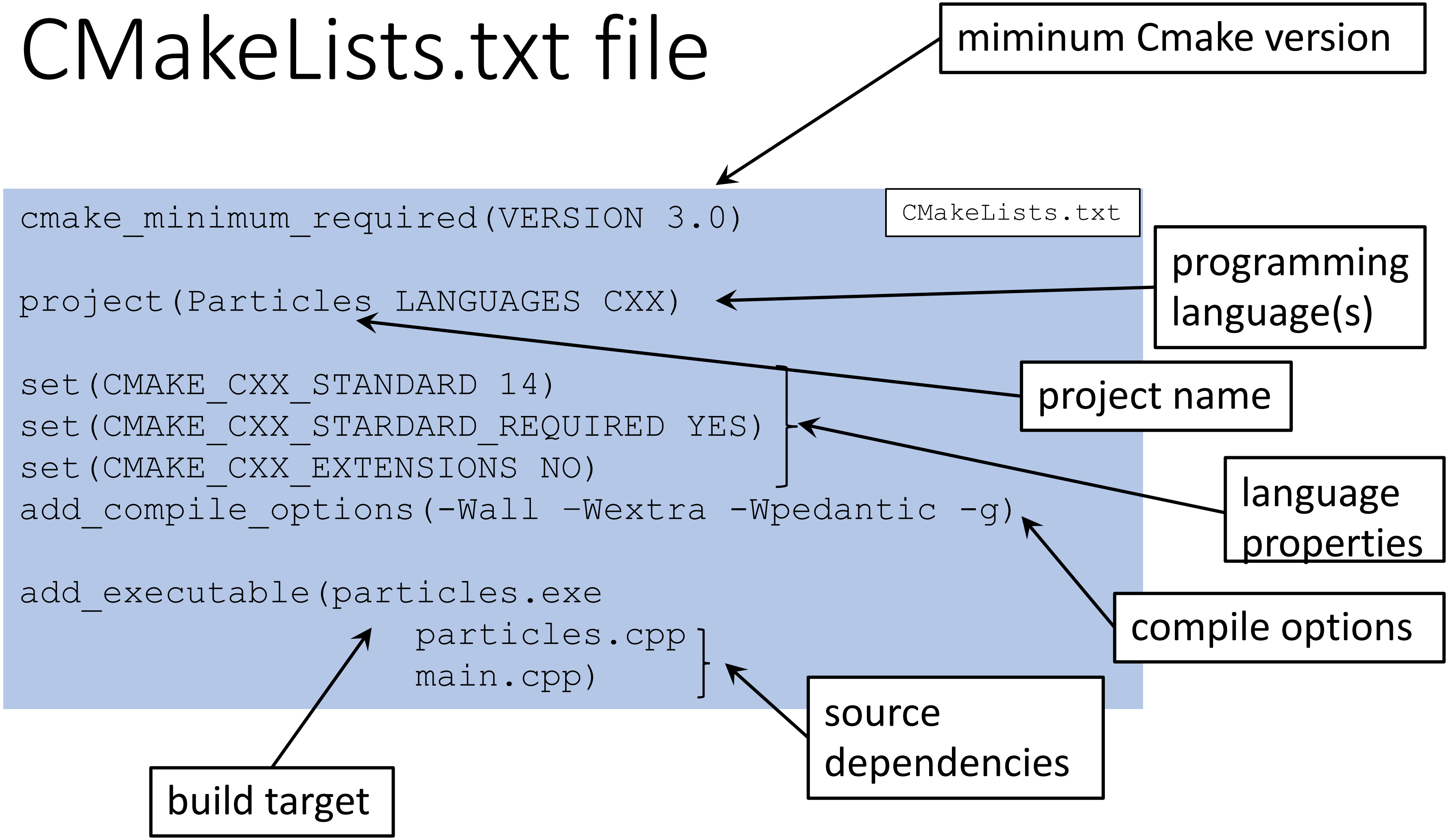
Caveats

- Writing your own make files
  - tedious
  - error prone
  - okay for small projects
- Better: use autotools
  - create `configure.ac` for project
  - create `Makefile.am` per directory
- Better still: consider CMake

CMake

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Modularity

CMakeLists.txt file



CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

project(Particles LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STARDARD_REQUIRED YES)
set(CMAKE_CXX_EXTENSIONS NO)
add_compile_options(-Wall -Wextra -Wpedantic -g)

add_executable(particles.exe
                   particles.cpp
                   main.cpp)
```

miminum Cmake version

programming language(s)

project name

language properties

compile options

source dependencies

build target

Using CMake

- Create, go to build directory

```
$  mkdir build &&  cd build
```

- Generate build files

```
$  cmake ..
```

- Builid software

```
$  cmake  --build .
```

- Only execute targets with modified dependencies
  - dependency tracking
  - saves lots of time on large projects
- Clean all build artifacts

```
$  cmake  --build .  --target clean
```

What was left out/added?

- Added
  - building software using make
  - building software using CMake

Error handling

Chapter 3, B. Stroustrup "A tour of C++"

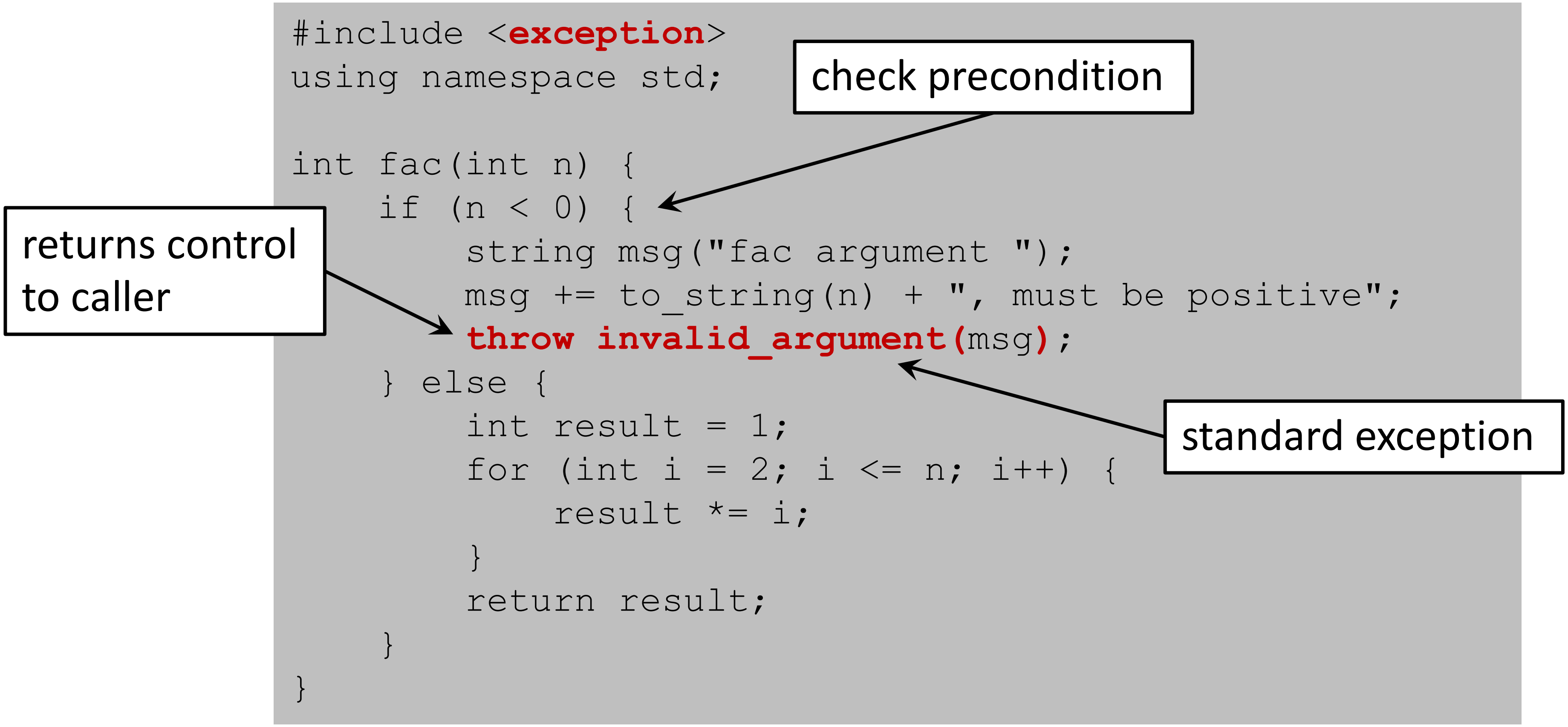
https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Modularity

Error handling

- Check for preconditions
  - valid arguments for functions?
- Invariants
  - valid state of object?
- Check for runtime problems
  - e.g., opening files
- Signal problems
  - don't fail silently

Throw exceptions!

Throw exception



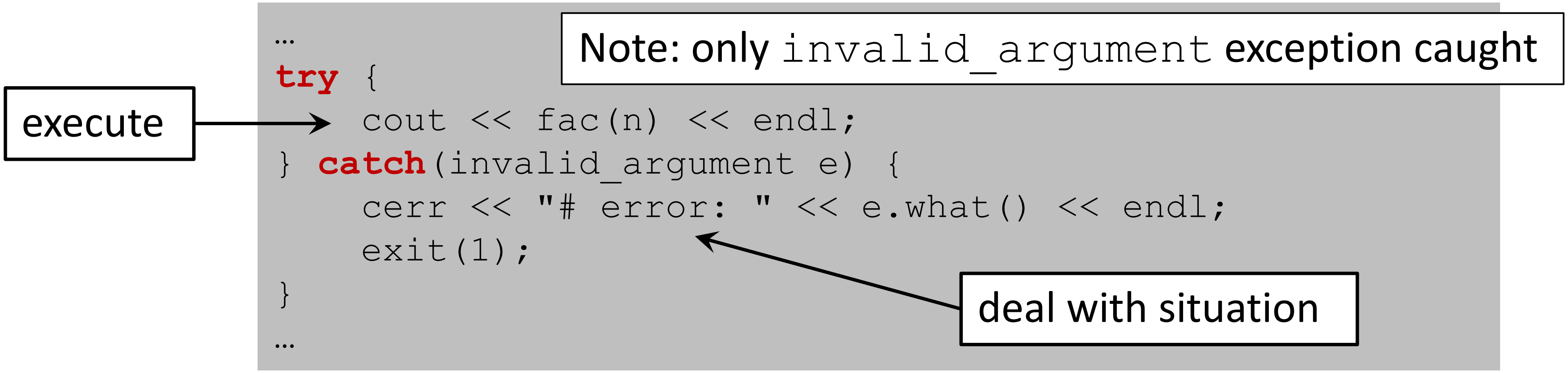
```
#include <exception>
using namespace std;

int fac(int n) {
    if (n < 0) {
        string msg("fac argument ");
        msg += to_string(n) + ", must be positive";
        throw invalid_argument(msg);
    } else {
        int result = 1;
        for (int i = 2; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

check precondition

returns control to caller

standard exception

Catch exception



execute

Note: only `invalid_argument` exception caught

```
…
try {
    cout << fac(n) << endl;
} catch(invalid_argument e) {
    cerr << "# error: " << e.what() << endl;
    exit(1);
}
…
```

deal with situation

- Multiple `catch` phrase are possible
- Exception can be rethrown with `throw;`
- Recover from exception if possible

Caveats

- Good error handling is hard
    - handle error at right level
    - convey maximal information to user
- Increases size of code base considerably
- Think of corner cases
- Requires testing

Do it right, or not at all!

Exit

- Use `std::exit(n)` to convey exit status to shell
  - 0: success
  - 1-127: failure
- Non-zero exit status
  - pick value per error condition, allows shell to do error handling
  - e.g., 1 ~ missing argument, 2 ~ wrong argument type,
    3 ~ wrong argument value

```
$  fac.exe  -1
# error: invalid argument value -1
$  echo  $?
3
```

What was left out/added?

- Left out
  - defining your own namespaces
- Added
  - exit status for using in shell

Classes

Chapter 4, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Classes

Original class

static_particle.h

```
class StaticParticle {
    private:
        double x_, y_, mass_;
    public:
        StaticParticle(double x, double y,
                       double mass) :
            x_ {x}, y_ {y}, mass_ {mass} {};
        double x() const { return x_; };
        double y() const { return y_; };
        double mass() const {return mass_; }
        double dist(const StaticParticle& other) const;
};
```

Extending functionality

- Particles with velocity

particle.h

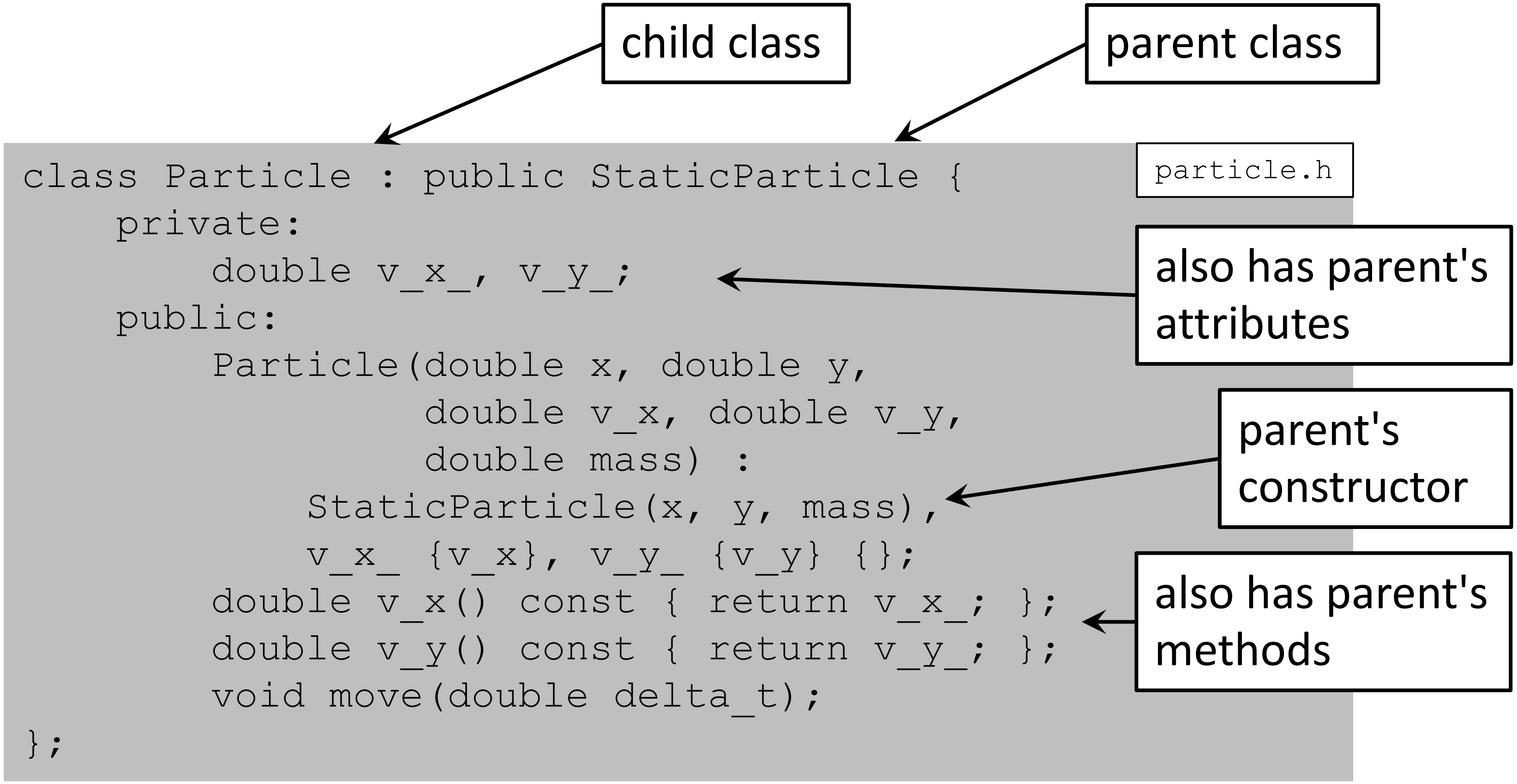
```
class Particle {
    private:
        double x_, y_, v_x_, v_y_, mass_;
    public:
        Particle(double x, double y,
                 double v_x, double v_y,
                 double mass) :
            x_ {x}, y_ {y}, v_x_ {v_x}, v_y_ {v_y},
            mass_ {mass} {};
        double x() const { return x_; };
        double y() const { return y_; };
        double v_x() const { return v_x_; };
        double v_y() const { return v_y_; };
        double mass() const {return mass_; }
        void move(double delta_t);
        double dist(const Particle& other) const;
};
```

red = new

Copy/paste? Bad idea!

- Difficult to maintain
  - bug fixing in many versions
  - new functionality might break older code
- Better: extend through inheritance
  - child can do what parent can
  - child can override parents behavior
  - child can do more than parent can
- Terminology
  - parent class = base class
  - child class = derived class

Inherit from class



child class

parent class

particle.h

```
class Particle : public StaticParticle {
    private:
        double v_x_, v_y_;
    public:
        Particle(double x, double y,
                 double v_x, double v_y,
                 double mass) :
            StaticParticle(x, y, mass),
            v_x_ {v_x}, v_y_ {v_y} {};
        double v_x() const { return v_x_; };
        double v_y() const { return v_y_; };
        void move(double delta_t);
};
```

also has parent's attributes

parent's constructor

also has parent's methods

Implementation: caveat

particle.cpp

```
void Particle::move(double delta_t) {
    x_ += v_x_*delta_t;
    y_ += v_y_*delta_t;
};
```

Problem: `x_` and `y_` are private to `StaticParticle`!

static_particle.h

```
class StaticParticle {
    protected:
        double x_, y_, mass_;
    …
};
```

can be accessed by descendants

Access control

- For
  - attributes: read/modify
  - methods: call
- Levels
  - `private`: only class can access
  - `protected`: only class and descendants can access
  - `public`: everyone can access

Be as paranoid as possible!

Using child classes

main.cpp

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

using namespace std;

int main(void) {
    StaticParticle p_s(0.0, 0.0, 1.0);
    cout << p_s << endl;
    Particle p1(1.0, 0.0, 1.0, 0.5, 1.0);
    cout << p1 << endl;
    Particle p2(0.0, 1.0, 0.0, 0.5, 2.0);
    cout << p2 << endl;
    cout << p1 << endl;
    const double delta_t = 0.1;
    p1.move(delta_t);
    cout << p1 <<  endl;
    cout << p1.dist(p_s) << endl;
    cout << p1.dist(p2) << endl;
    return 0;
}
```

only for `Particle`, not `StaticParticle`

calling inherited method from `StaticParticle`

More overloading

static_particle.cpp

```
#include <ostream>
#include "static_particle.h"
...
std::ostream& operator<<(std::ostream& out,
                         const StaticParticle& p) {
    return out << "(" << p.x() << ", " << p.y() << ")"
           << ", mass = " << p.mass();
}
```

particle.cpp

```
#include <ostream>
#include "particle.h"
...
std::ostream& operator<<(std::ostream& out,
                         const Particle& p) {
    return out << static_cast<StaticParticle>(p) << ", ("
               << p.v_x() << ", " << p.v_y() << ")";
}
```

type cast, `p` is also `StaticParticle`

What was left out?

- Abstract classes
  - virtual functions
- Multiple inheritance/class hierarchy
- Copy versus move

Templates

Chapter 4, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Templates

Function templates

```
void swap_val(int& x, int& y) {
    int tmp {x};
    x = y;
    y = tmp;
}
```

```
void swap_val(double& x, double& y) {
    double tmp {x};
    x = y;
    y = tmp;
}
```

... ???

```
template<typename T>
void swap_val(T& v1, T& v2) {
    T tmp {v1};
    v1 = v2;
    v2 = tmp;
}
```

Using templates

```
template<typename T>
void swap_val(T& v1, T& v2) {
    T tmp {v1};
    v1 = v2;
    v2 = tmp;
}
...
    double x {3.1};
    double y {5.7{;
    swap<double>(x, y);
    int m {3};
    int n {5};
    swap<int>(m, n);
```

Variadic templates

- Implementing function with arbitrary number of arguments

```
double sum() { return 0.0; }

template<typename T, typename... Tail>
double sum(T head, Tail... tail) {
    return head + sum(tail...);
}
…
std::cout << sum(1.2, 2.3, 3.4) << std::endl;
std::cout << sum(1.2, 2.3, 3.4, 4.5) << std::endl;
```

base case:
no arguments

tail recursion:
first element +
function on tail

Function `sum` overloaded

Aliases

- Define new name for type
  - more compact
  - easier to understand/maintain

```
#include <array>
#include <cmath>

using Position = std::array<double, 3>;

inline double sqrt(double x) { return x*x; }

double distance(const Position& p1, const Position& p2) {
    double dist {0.0};
    for (int i = 0; i < p1.size(); i++)
        dist += sqr(p1[i] - p2[i]);
    return std::sqrt(dist);
}
```

Higher order functions

- Consider

```
#include <functional>
#include <iostream>

void integrate(std::function<double(double)> f,
               const double delta_t,
               const double max_t) {
    for (double t = 0.0; t <= t_max; t += delta_t)
        std::cout << t << "," << f(t) << "\n";
}
```

Function as argument of function

What if `f(t, freq)`*, how to use* `integrate`*?*

Function objects

- Class to create "family" of function objects

```
class Pendulum {
    private:
        double freq_;
        constexpr double pi {acos(-1.0)};
    public:
        Pendulum(const double& freq) : freq_ {freq} {};
        double operator()(const double& t) const {
            return cos(2.0*pi*freq_*t);
        };
};

Pendulum pendulum(0.5);
integrate(pendulum, 0.01, 1.0);
```

Interlude: currying with bind

- Bind function arguments to values

```
#include <functional>
const double pi {acos(-1.0)};

double pendulum_func(double t, double freq) {
    return cos(2.0*pi*freq*t);
}
…
using namespace std::placeholders;
auto pendulum = std::bind(pendulum_func, _1, 0.5);
integrate(pendulum, 0.01, 1.0);
```

Interlude: lambda functions

- Anonymous function created at runtime: closures

```
const double pi {acos(-1.0)};

double pendulum_func(double t, double freq) {
    return cos(2.0*pi*freq*t);
}
…
double freq {0.5};
…
integrate([=](double t) { return pendulum_func(t, freq); },
          0.01, 1.0);
```

capture `freq` by value

`[…]`: capture variables in body from context

- `[=]`: by value
- `[&]`: by reference
- `[]`: capture nothing

Templates: discussion

- Useful for
  - generic programming
  - expressing concepts
- Duck typing
- Caveats
  - errors are caught late during compilation
    ⇒ long & cryptic error messages

What was left out/added?

- Left out
  - Container templates, i.e., writing your own generic containers
- Added
  - Currying
  - Lambda functions

Strings & regular expressions

Chapter 7, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Regexes

Strings

- Strings: sequences of characters

```
using namespace std;
…
string str {"hello"};
str += " world!";
cout << str.substr(6, 5) << endl;
auto pos = str.find("w");
str[pos] = toupper(str[pos]);
str.replace(0, 1, "H");
cout << str << endl;
pos = 0;
while ((pos = str.find("o", pos)) != string::npos) {
    cout << "found at " << pos << endl;
    pos++;
}
str.insert(6, "Beautiful ");
cout << str << endl;
```

```
world
```

```
Hello World!
```

```
found at 4
found at 7
```

```
Hello Beautiful World!
```

std::string versus C-style

- C-style string
  - array of `char`
  - last element `'\0'`
  - functions declared in `string.h`
- Useful for calling C functions
- Conversion
  - std::string → C-style: `str.c_str()`
  - C-style → std::string: `std::string` constructor

Regular expressions: definition

- Regular expression
  = description of a language
  ≡ set of strings
- Language can be
  - Finite
  - Infinite
    - Remember, set of all strings is infinite, countable
- Chomsky hierarchy
  - regular languages
    ⊂ context-free languages
    ⊂ context-sensitive languages
    ⊂ recursively enumerable languages

C++ regular expressions can express more than regular languages

Regular expressions: expressive power

- *Never* parse HTML or XML with regular expressions!!!
  - HTML & XML are *context-free* languages
  - Even if you think you can, *don't*, there be dragons
- Can you write a regular expression to match all regular expressions?
  - No: the language of regular expressions is context-free
- Can you parse English using a regular expression
  - No: English is a little bit context-sensitive

Regular expressions: examples I

- DNA: `[ACGT]+`
  - `[ACGT]` = one out of {A, C, G, T}
  - *expr*`+` = one or more repetitions of *expr*
- DNA containing AAT: `[ACGT]*`**`AAT`**`[ACGT]*`
  - $expr_1$ $expr_2$ = $expr_1$ followed by $expr_2$
  - *expr*`*` = zero or more repetitions of *expr*
- DNA containing AAT or TAT:

  `[ACGT]*(`**`AAT`**`|`**`TAT`**`)[ACGT]*`
  - $expr_1$ | $expr_2$ = either $expr_1$, or $expr_2$

Regular expressions: examples II

- Belgian phone number:

  `0[1-9]\d?/[1-9]\d{5,6}`

  - $[c_1–c_2]$ = any character from c_1 to c_2
  - `\d` = `[0-9]`
  - *expr*`?` = zero or one occurrence of *expr*
  - *expr*`{`*m*`,`*n*`}` = *m* to *n* repetitions of *expr*
- All strings, including empty string: `.*`
  - `.` = any character (except newline)
- Email address: `\w+(?:\.\w+)?@\w+(?:\.\w+)+`
  - `\.` = character `'.'`
  - `\w` = `[A-Za-z0-9_]`
  - `(?:` *expr*`)` = grouped *expr*

Don't use this in practice!!!

Similar to brackets in math expressions

Regular expressions: characters

- Characters that must be escaped
  - tab : `\t`
  - new line : `\n`
  - carriage return : `\r`
  - \ : `\\`
  - brackets : `\(`, `\)`, `\[`, `\]`, `\{`, `\}`
  - operators : `\+`, `\-`, `\*`, `\?`
  - `.` (dot) : `\.`
- All other characters literal

Regular expressions: character classes

- `x` = {'x'}
- `[xyz]` = {'x', 'y', 'z'}
- `[x-z]` = {c | 'x' ≤ c ≤ 'z'}
- `[^xyz]` = {any} \ {'x', 'y', 'z'}
- `\w` = {'A',…,'Z', 'a',…,'z', '0',…,'9', '_'}
- `\W` = {any} \ {'A',…,'Z', 'a',…,'z', '0',…,'9', '_'}
- `\d` = {'0',…,'9'}
- `\D` = {any} \ {'0',…,'9'}
- `\s` = {' ', '\t', '\f', '\r', '\n', '\v'} (white space)
- `\S` = {any} \ {' ', '\t', '\f', '\r', '\n', '\v'}
- `.` = {any} \ {'\n'}

Regular expressions: operators

- Concatenation: $expr_1\ expr_2$ (implicit)
- Choice: $expr_1 \mid expr_2$ = either $expr_1$, or $expr_2$
- Repetition:
  - $expr\{n\}$ = exactly n repetitions of $expr$
  - $expr\{m,n\}$ = minimum m, maximum n repetitions of $expr$ where $m \leq n$
  - $expr\{,n\}$ = minimum zero, maximum n repetitions of $expr$
  - $expr\{m,\}$ = minimum m repetitions of $expr$
  - $expr?$ = zero or one occurrence of $expr$
  - $expr*$ = zero or more repetitions of $expr$
  - $expr+$ = one or more repetitions of $expr$

Longest match semantics

Greedy vs. non-greedy operators

- Consider string '`<var name="x">15</var>`'
  - `<.+>` will match substring '`<var name="x">15</var>`'

Longest match semantics!

- Use non-greedy operator
  - `<.+?>` will match substring '`<var name="x">`'
- *expr<op>*? = operator *<op>* with shortest match semantics (i.e., non-greedy) applied to *expr*
- Alternative: `<[^>]+>`

Why not parse XML with REs?

- Task: match start tag in
  - `'<var name="x">15</var>'`
    `<.+?>` will match substring
    `'<var name="x">'` √
  - `'<var name="a->b">15</var>'`
    `<.+?>` will match substring
    `'<var name="a->'`

Oops!

Use a parser for context free language, or, better still, use a third-party library.

Raw strings

- Regular expressions contain many \: pain
  - regular expression:
    `\w+(?:\.\w+)?@\w+(?:\.\w+)+`
  - string representation:
    `"\\w+(?:\\.\\w+)?@\\w+(?:\\.\\w+)+"`
- Raw strings: \ has no special semantics
  - raw string representation:
    `R"(\w+(?:\.\w+)?@\w+(?:\.\w+)+)"`

Searching matches

- Checking occurrence

```
#include <regex>
using namespace std;
…
regex expr {R"(\w+(?:\.\w+)?@\w+(?:\.\w+)+)"};
if (regex_search(str, expr))
    …
```

- Getting matched string

```
regex expr {R"(\w+(?:\.\w+)?@\w+(?:\.\w+)+)"};
smatch matches;
if (regex_search(str, matches, expr))
    cout << "found: " << matches[0] << endl;
```

Extracting matches

- Grouping: `(?:…)`
- Capturing brackets: `(…)`

```
regex expr {R"((\w+(?:\.\w+)?)@(\w+(?:\.\w+)+)"};
smatch matches;
if (regex_search(str, matches, expr)) {
    string user_name = matches[1];
    string domain_name = matches[2];
    …
}
```

Note: capturing brackets also group, but lots of machinery

Replacing matches

- Format string for replacement
    - `$1`: first capture
    - `$2`: second capture
    - …
    - `$&`: complete match
    - literal characters

```
const string str {"1.5, 2.3, alpha"};
regex expr {R"(([^ ,])+)"};
string new_str = regex_replace(str, expr, "'$1'");
cout << new_str << endl;
```

'1.5', '2.3', 'alpha'

Iterating matches

```
string line;
regex expr {R"((\w+))"};
unordered_map<string, int> counter;
while (getline(cin, line)) {
    for (sregex_iterator token(line.begin(), line.end(), expr);
                token != sregex_iterator {}; token++) {
        string word = (*token)[1];
        if (counter.find(word) == counter.end())
            counter[word] = 0;
        counter[word]++;
    }
}
…
```

- `sregex_iterator` is bidirectional, hence stop condition
- `token` is address of matched substring, hence `*token`
- Match was capturing, hence `(*token)[1]`

Miscellaneous remarks

- Regular expressions are
  - powerful
  - somewhat slow

  ⇒ use judiciously
- Two functions
  - regex_search: works on streams ⇒ more versatile
  - regex_match: works on strings only ⇒ better performance
- Modifiers
  - case insensitive: `regex expr(…,` **`regex::icase`**)
  - more to come in C++17

What was left out/added?

- Left out
  - String implementation

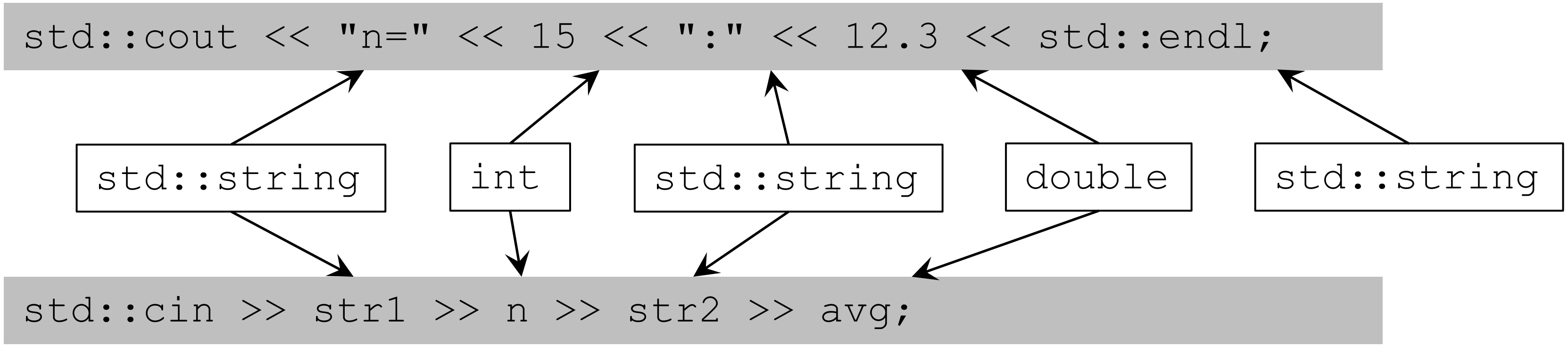
I/O streams

Chapter 7, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/IoStreams

I/O streams

- Output stream (`ostream`)
  - convert typed object(s) to sequence of characters



```
std::cout << "n=" << 15 << ":" << 12.3 << std::endl;
```

`std::string` `int` `std::string` `double` `std::string`

```
std::cin >> str1 >> n >> str2 >> avg;
```

- Input stream (`istream`)
  - convert sequence of characters to typed object(s)

Standard streams

- Output streams
  - `std::cout`: standard output
  - `std::cerr`: standard error
  - "put to" operator: `<<`
  - Cross platform end-of-line: `std::endl`
- Input stream
  - `std::cin`: standard input
  - "get from" operator: `>>`
  - skips initial whitespace : `' '`, `'\t'`, `'\n'`, `'\r'`,…
  - default separator: whitespace
  - read entire line, including end-of-line:
    `std::getline(std::cin, line)`

Stream state

- Result of `>>` is reference to `istream`
- Reference to `istream` evaluates to `true` if ready for reading

```
double data {0.0};
double sum {0.0};
while (std::cin >> data)
    sum += data;
std::out << "sum = " << sum << std::endl;
```

- Explicit check end-of-file: `std::cin.eof()`

Floating point formatting

- Floating point formats: `scientific`, `fixed`, `defaultfloat`
  - Getting/setting precision (number digits), e.g., `cout.precision()`/`cout.precision(4)`

```
#include <iomanip>
…
const double PI {acos(-1.0)};
cout << PI << endl;
cout << scientific << PI << endl;
cout.precision(4);
cout << defaultfloat << PI << endl;
```

```
3.14159
```

```
3.141593e+00
```

```
3.142
```

Formatting: width and fill

- Getting/setting width, e.g., `cout.width()/cout.width(5)`
- Getting/setting fill character, e.g., `cout.fill()/cout.fill('0')`

```
const int data {123};
cout << data << endl;
auto orig_width = cout.width();
cout.width(5);
auto orig_fill = cout.fill();
cout.fill('0');
cout << data << endl;
cout.width(orig_width);
cout.fill(orig_fill);
```

```
123
```

```
00123
```

File streams

- Input file stream `ifstream`

```
#include <fstream>
…
ifstream ifs("data.txt");
if (!ifs) { /* file could not be opened */; }
double data {0.0};
ifs >> data;
…
ifs.close();
```

open and close file

- Output file stream `ofstream`

```
#include <fstream>
…
ofstream ofs("data.txt");
if (!ofs) { /* file could not be opened */; }
double data = …;
ofs << data;
ofs.close();
```

open and *close* file!

String streams

- Reading from/writing to `std::string`

```
#include <sstream>
#include <vector>
…
vector<double> data;
string line;
getline(cin, line);
stringstream str(line);
double item {0.0};
str >> item;
data.push_back(item);
char sep;
while ((sep = str.get()) != -1) {
    str >> item;
    data.push_back(item);
}
```

Pointers

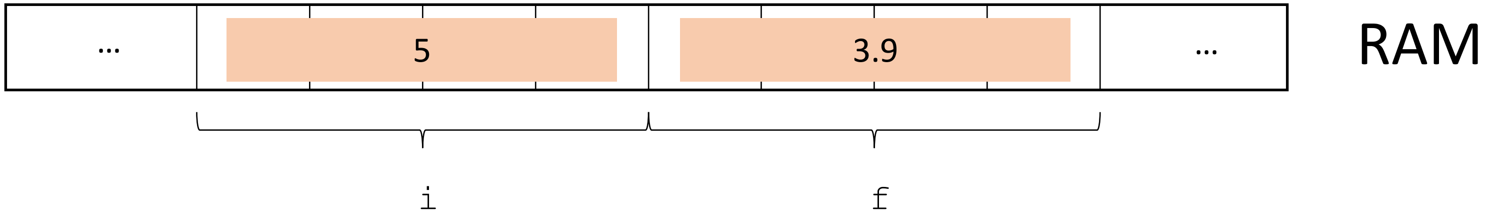
Chapter 4, B. Stroustrup "A tour of C++“

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Pointers

Data management

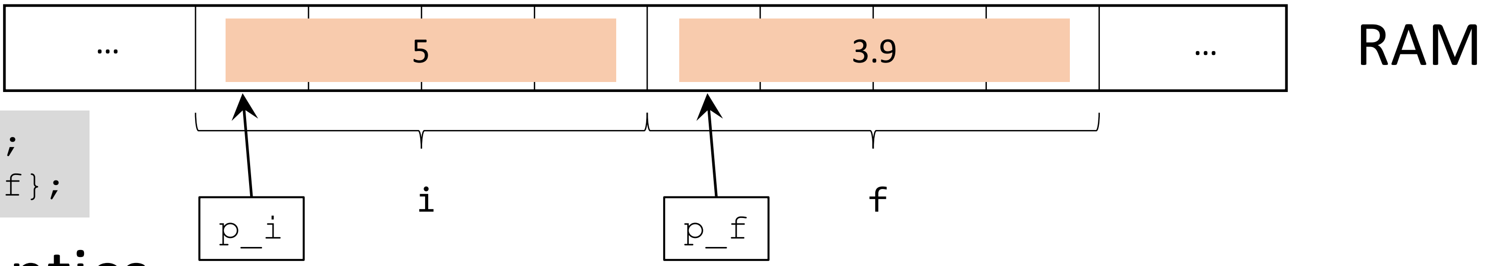
- Working data is stored in volatile RAM (Random Access Memory)
- RAM ≈ sequence of bytes
- Value of variable = sequences of bytes in RAM
- (Value of) variable has address

```
int i {5};
float f {3.9f};
```



Addresses

- Get address: & operator
- Assign to "address" variable = pointer
  - address of int to int pointer = int*
  - address of float to float pointer = float*
  - …



```
int* p_i {&i};
float* p_f {&f};
```

- Semantics
  - p_i: address of int value
  - p_f: address of float value

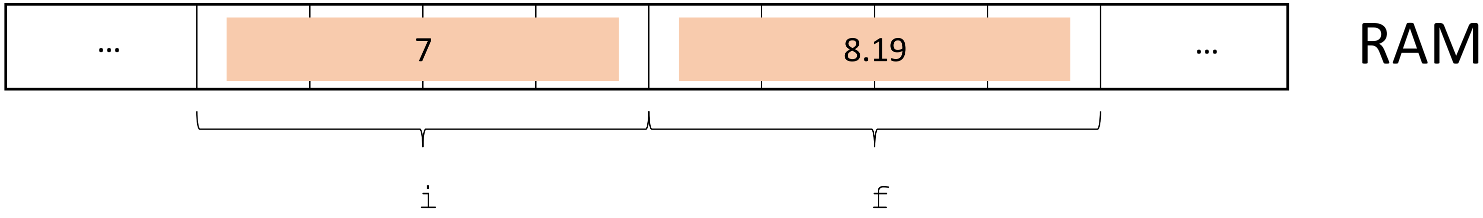
Using addresses

- Value at address: * operator
  - *p_i ≡ 5
  - *p_f ≡ 3.9f
- Use value at address

```
std::cout << "value at " << p_i << " = " << *p_i << std::endl;
std::cout << std::cos(*p_f) << std::endl;
```

- Assign new value to address

```
*p_i = 7;
*p_f *= 2.1f;
```



One step further…

Fundamental theorem of software engineering:

We can solve any problem by introducing an extra level of redirection.

David J. Wheeler

- `p_i` is variable
    - value is at `&p_i`
- Assign address to pointer to pointer to `int`

```
int** pp_i {&p_i};
```

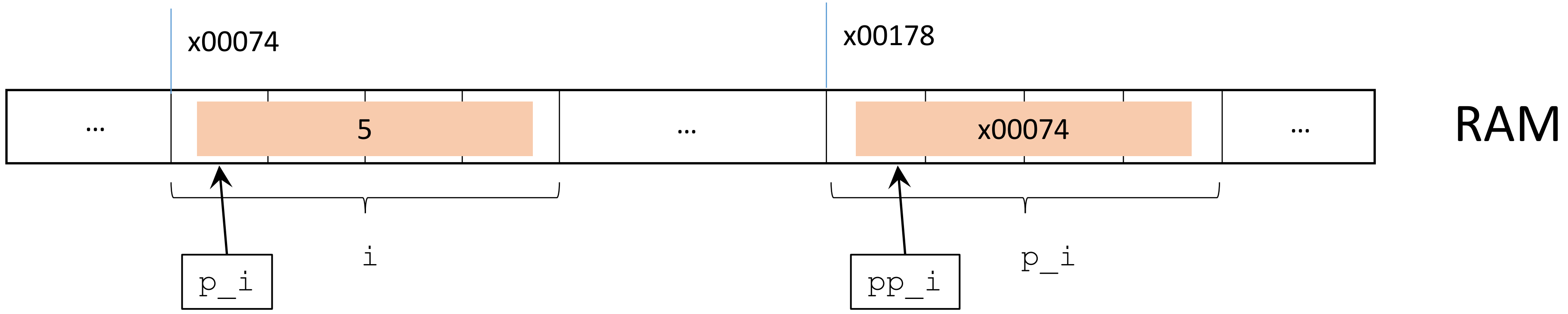
- Use address

```
std::cout << "i is at " << p_i << ", "
          << "p_i is at " << pp_i << ", "
          << "value of pp_i is " << *pp_i << ", "
          << "value of i is " << **pp_i;
```

indirection

double indirection

Double indirection



Using object vs. pointer to object

```
struct Point {
    double x, y;
    Point(double x_, double y_) : x {x_}, y {y_} {}
    void print() const { std::cout << x << "," << y; }
};
```

`Point` object

```
Point p(3.2, 5.1);
p.x = 3.7;
p.print();
```

pointer to `Point` object

```
Point* p = new Point(3.2, 5.1);
p->x = 3.7;
p->print();
```

dot operator ≈ member operator ≈ arrow operator

Do we care?

- Mostly no… but sometimes we do!
- C++ programs use two types of memory
  - stack
    - stores function arguments
    - stores local variables
    - return value
  - heap
    - stores explicitly allocated data
      - by data types, e.g., `std::valarray`, `std::vector`, …
      - by programmer: `new`

stack frame

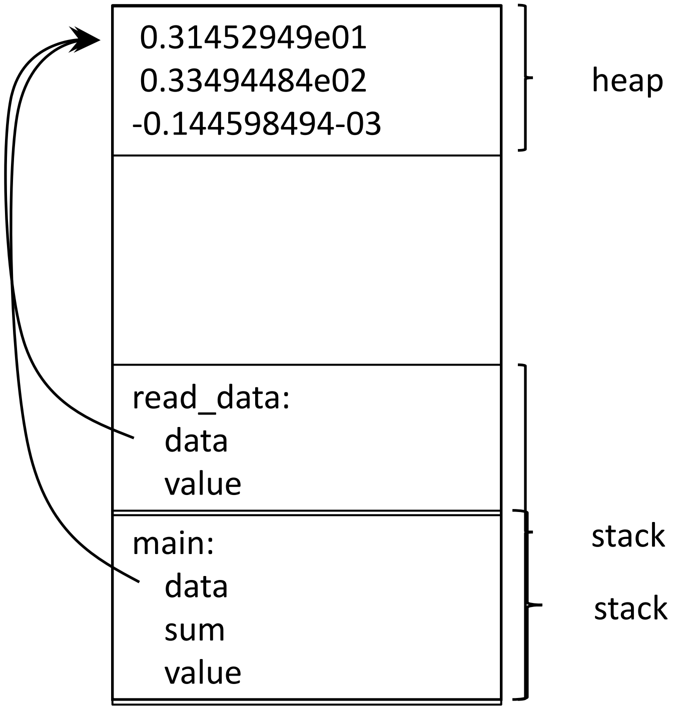
lifetime: function execution

lifetime: managed by programmer

Example: `std::vector`

```
int main() {
    …
    std::vector<double> data = read_data();
    double sum {0.0};
    for (const auto& value: data)
        sum += value;
    …
}
```

```
std::vector<double> read_data() {
    std::vector<double> data;
    double value;
    while (std::cin >> value)
        data.push_back(value);
    return data;
}
```



0.31452949e01
0.33494484e02
-0.144598494-03

heap

read_data:
data
value

main:
data
sum
value

stack

stack

Memory management

- Heap memory
  - explicitly allocated when required
  - explicitly deallocated when no longer required
- STL containers do that for you
  - constructor: memory allocation
  - move constructor/assignment: move resource handles
  - copy constructor/assignment: copy resources
  - destructor: memory deallocation

Manual memory management

Avoid it!
Use STL or smart pointers

- Allocate memory heap: `new`
- Ensure correct copy of data: copy constructor, copy assignment
- Ensure correct move of data: move constructor, move assignment
- Deallocate memory: `delete`
- Problems
  - no `delete`: memory leak
  - double `delete`: segmentation fault
  - no move semantics: performance issues
  - no resource copying: segmentation fault or bugs

Semi-automatic: smart pointers

- `std::unique_ptr<T>`: unique resource ownership
  - auto-deleted when owner goes out of scope
- `std::shared_ptr<T>`: shared resource ownership
  - auto-deleted when last owner goes out of scope
  - requires bookkeeping: number of owners is tracked
- `std::weak_ptr<T>`: temporary resource ownership
  - constructed from `std::shared_ptr<T>`
  - not counted for reference count
  - to use, convert to `std::shared_ptr<T>`
  - use cases
    - models temporary ownership
    - breaks cyclic references (e.g., graphs)

What was left out/added?

- Added
  - Memory management
  - C-style pointers

Containers

Chapter 9, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Containers

Motivation

- Data structures are key to good programming
  - implementation conceptually close to model
  - fewer lines of code = less bugs
  - better performance
- Programming languages
  - C++: STL (Standard Template Library)
  - Python: core language, standard library
  - Java: standard library

For all languages, many 3rd party libraries

- Don't reinvent the wheel!

It's a zoo…

- Many data structures
    - specific properties
    - specific applications
    - relationship to algorithms!
- Important to have an overview

Which data structure to use in models?

Which data structure to choose for algorithm?

- Programming language independent
    - conceptual, mathematical level

Notation

- Type T: set of values, e.g.,
  - boolean = {true, false}
  - int = {-2147483648, - 2147483647, …, -1, 0, 1…, 2147483647}
- Size of type T: $|T|$
- Property: $\forall T_1, T_2 : T_1 \neq T_2 \Rightarrow T_1 \cap T_2 = \varnothing$
- Power set of T: 2^T, e.g.,
  - 2^{boolean} = {∅, {true}, {false}, {true, false}}
  - 2^{int} = {∅, {0}, {1}, {-1}, …, {0, 1}, {0, -1}, …}

$|2^T| = 2^{|T|}$

- Set of all sequences of T: T^*, e.g.,
  - boolean^* = {∅, true, false, true·true, true ·talse, talse ·true,…}
  - int* = {∅ , 0, 1, …, 0 ·0, 0 ·1, …, 0 ·0 ·0, 0 ·0 ·1, …}

$|T^*| = \infty$

Basic data structures

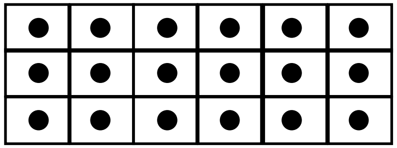
- Data structures provided
  - core language
  - standard libraries
- Other data structures can be implemented on top
- Contents
  - array
  - valarray
  - vector
  - tuple
  - list
  - set
  - map

Array

- Characteristics
  - access: random by ordinal index
  - ordered
  - fixed length
  - update: O(1)
  - retrieval: O(1)
  - search: O(n)
  - element type: homogenous
- Implementation: core language

d-dimensional array a

$$a \in T^{n_0} \times T^{n_1} \times \ldots \times T^{n_{d-1}}$$



Array examples

```
…
int[] a = {3, 5, 7, 9};
for (int i = 0; i < 4; ++i) {
    cout << a[i]*a[i] << endl;
}
```

Avoid if possible!

```
…
a[0] = 12;
a[1] = a[0] + 13;
```

Note: array indexing is zero based!

STL array

- Properties of array
- Size is known at compile time.
- Implementation: STL

STL array examples

```
#include <array>
…
std::array<int, 4> a {3, 5, 7, 9};
for (const auto& element: a) {
    cout << element*element << endl;
}
```

```
…
a[0] = 12;
a[1] = a[0] + 13;
```

Note: array indexing is zero based!

Value array

- Properties of array
- Support for mathematical operations
  - `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`
  - functions: `sqrt`, `sin`, `cos`, `log`, `exp`, …
- Implementation: STL

Value array example

```
#include <valarray>
…
valarray<double> data = {3.5, 7.3, 9.1};
valarray<double> data_tr(data.size());
data_tr = 3.0 + data;
for (const auto& value: data_tr) {
    cout << value << endl;
}
```

`valarray` keeps track of size

overloaded arithmetic operators

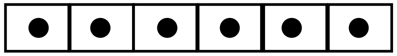
- range for loop
  - iterates over all values in container
  - variable type = data type in container
  - use `const` when value won't change

Vector

- Characteristics
  - access: random by ordinal index
  - ordered
  - length can vary
  - update: O(1)
  - retrieval: O(1)
  - search: O(n)
  - element type: homogenous
- Implementation: STL

1-dimensional array-like a

$$a \in T^n$$



Vector example I

```
#include <vector>
…
vector<double> read_data(istream& in) {
    vector<double> data;
    double item;
    while (in >> item)
        data.push_back(item);
    return data;
}
```

Vector example II

```
#include <vector>
…
Stats compute_stats(vector<double>& data) {
    int n = data.size();
    double sum {0.0};
    for (const double item: data)
        sum += item;
    Stats stats;
    stats.n = n;
    stats.mean = sum/n;
    return stats;
}
```

Note: vector indexing is zero based!

STL `Container` API

- `c.empty()`
  - true if container empty
- `c.size()`
  - number of items in container
- `c.max_size()`
  - maximum capacity of container

STL `SequenceContainer` API

- `c.at(index)`
  - accessing element at `index` (0-based)
  - range checked, safer
- `c[index]`
  - accessing element `index` (0-based)
  - not ranged checked, faster
- `c.front()`/`c.back()`
  - first/last element
- `c.push_back(e)`
  - add element `e` at end
- `c.insert(it, e)`
  - insert an element `e` before position `it` iterator

Tuple

- Characteristics
    - access: random by ordinal index
    - ordered
    - fixed length
    - insert/update: N/A
    - retrieval: O(1)
    - search: N/A
    - element type: any combination
- Implementation: STL

d-tuple t

$$t \in T_0 \times T_1 \times \ldots \times T_{d-1}$$

Tuple example

```
#include <tuple>
…
auto electron_prop = std::make_tuple(9.11e-31, -1);
…
std::cout << "mass: " << std::get<0>(electron_prop)
          << std::endl;
std::cout << "charge: " << std::get<1>(electron_prop)
          << std::endl;
…
double mass;
int charge;
std::tie(mass, charge) = electron_prop;
std::cout << "mass: " << mass << std::endl;
std::cout << "charge: " << charge << std::endl;
```

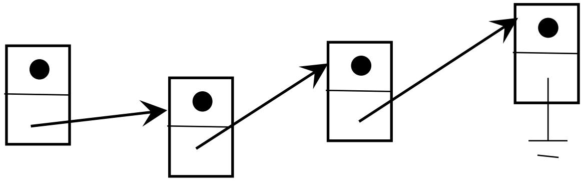
Note: tuple indexing is zero based!

List

- Characteristics
  - access: random by ordinal index
  - ordered
  - length can vary
  - insert/update: O(n)
  - retrieval: O(n)
  - search: O(n)
  - prepend/append/pop/unshift: O(1)
  - element type: homogenous
  - operations: concatenation
- Implementation: STL

list l

$l \in T^*$



List examples

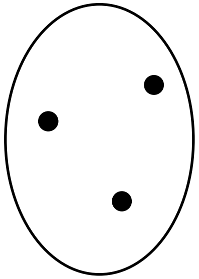
```
#include <list>
…
std::list<int> list;
for (int i = 0; i < 10; i++)
    list.push_back(i);
for (const auto& value: list)
    std::cout << value << std::endl;
```

Set

- Characteristics
  - access: iterator
  - unordered
  - size can vary
  - insert/remove: O(1)
  - search: O(1)
  - element type: homogenous
  - elements are unique in set
  - operations: union, intersection, …
- Implementation: STL

set s

$s \in 2^T$



Set example

id	dim_nr	temp
1	1	-0.5
2	1	0.0
4	2	-0.5
5	2	0.0
6	2	0.5
8	3	0.0
9	3	0.5
10	4	0.0

```
#include <iostream>
#include <unordered_set>

int main(void) {
    std::string col1, col2, col3;
    std::cin >> col1 >> col2 >> col3;
    int id, dim_nr;
    double temp;
    std::unordered_set<int> dim_nrs;
    while (std::cin >> id >> dim_nr >> temp)
        dim_nrs.insert(dim_nr);
    for (const auto& dim_nr: dim_nrs)
        std::cout << dim_nr << std::endl;
    return 0;
}
```

Map

- Characteristics
  - access: random by key
  - unordered
  - size can vary
  - insert/update: O(1)
  - retrieval: O(1)
  - search: O(1)
  - element type:
    - homogenious for key
    - homogenious for value
  - keys are unique in map
  - operations: union
- Implementation: STL

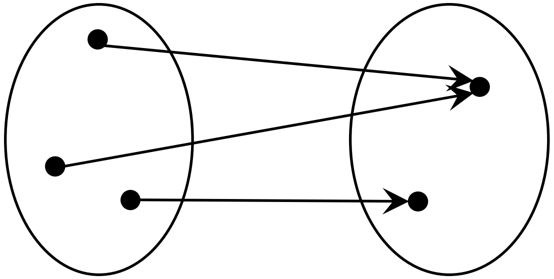
map m

$m \in T_1 \rightarrow T_2$

surjective function

keys $\subseteq T_1$

values $\subseteq T_2$



Map example

id	dim_nr	temp
1	1	-0.5
2	1	0.0
4	2	-0.5
5	2	0.0
6	2	0.5
8	3	0.0
9	3	0.5
10	4	0.0

```
#include <iostream>
#include <unordered_map>

int main(void) {
    std::string col1, col2, col3;
    std::cin >> col1 >> col2 >> col3;
    int id, dim_nr;
    double temp;
    std::unordered_map<int, int> dim_nr_counts;
    while (std::cin >> id >> dim_nr >> temp)
        dim_nr_counts[dim_nr]++;
    for (const auto dim_nr: dim_nr_counts)
        std::cout << dim_nr.first << ": "
                  << dim_nr.second << std::endl;
    return 0;
}
```

pair

Unordered versus default

- `unordered_set`
  - elements not sorted
  - faster insert
- `set`
  - elements sorted (custom comparator supported)
  - slower insert

- `unordered_map`
  - keys not sorted
  - faster insert
- `map`
  - keys sorted (custom comparator supported)
  - slower insert

Contiguous vs. non-contiguous

- Data stored contiguously in memory allows prefetch
  - decreases memory latency

Many codes are memory bound!

- Data types
  - valarray
  - vector

Use these for memory-intensive algorithms, *never* list/queue/…

Specialized data structures

- Data structures provided
  - standard libraries
  - third-party libraries
- Often implemented on top of basic data structures
- Other data structures can be implemented on top
- Contents
  - stack
  - queue, priority queue
  - graph, DAG, tree

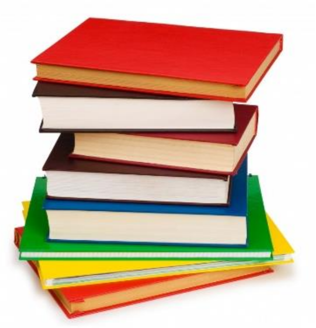
Stack

- Characteristics
  - access: only top
  - ordered
  - length can vary
  - push/peek/pop: O(1)
  - element type: homogenous

stack s

$s \in T^*$

First in, last out



- Implementation: STL

Stack examples

```
#include <stack>
…
std::stack<int> s;
for (int i = 0; i < 10; i++)
    s.push(i);
while (!s.empty()) {
    std::cout << s.top() << std::endl;
    s.pop();
}
```

Queue

- Characteristics
  - access: front to pop and back to push
  - ordered
  - length can vary
  - push/front/pop: O(1)
  - element type: homogenous

queue q

$q \in T^*$

First in, first out

- Implementation: STL

Queue examples

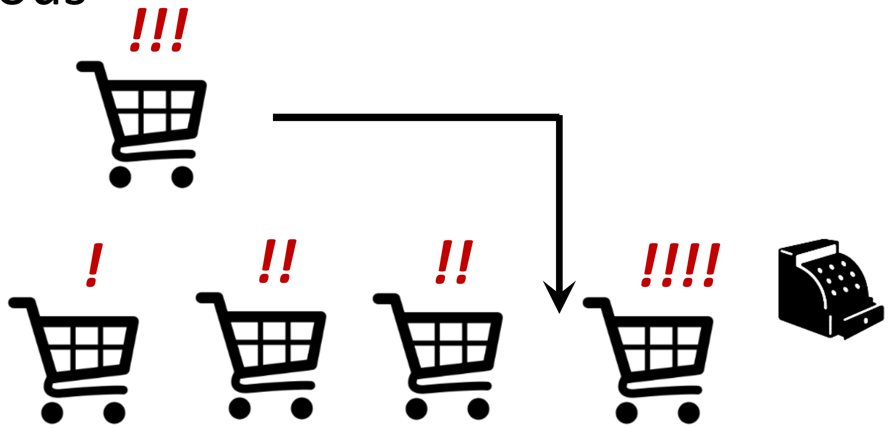
```
#include <queue>
…
std::queue<int> q;
for (int i = 0; i < 10; i++)
    q.push(i);
while (!q.empty()) {
    std::cout << q.front() << std::endl;
    q.pop();
}
```

Priority queue

- Characteristics
  - access: only front to pop, push inserts in order
  - ordered according to priority
  - length can vary
  - front: O(1), pop/push: O(log n)
  - element type: homogenous
- Implementation: STL

priority queue q

$q \in T^*$



Graph

- Characteristics
  - represents relationships (= edges) between objects (= vertices)
  - ordered (directed graph or digraph), unordered (undirected graph)
  - number of vertices can vary
  - number of edges can vary
  - edges can have associate info

graph g

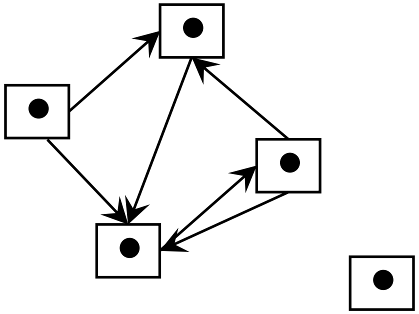
$g \in T_v \times T_v \to T_e$

vertex type

edges

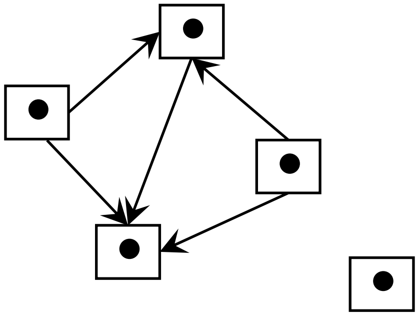
edge info type

- Implementations
  - e.g., as adjacency list
  - Boost library

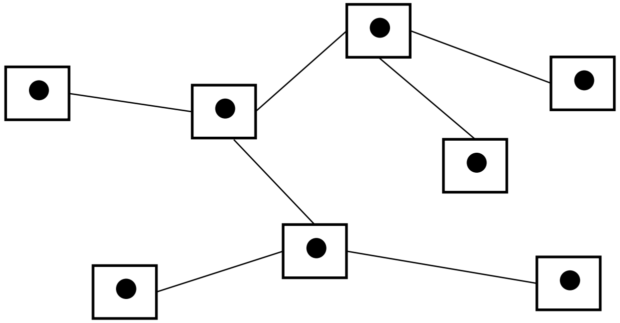


Some special graph types

- Directed Acyclic Graph (DAG)
  - directed graph contains no cycles

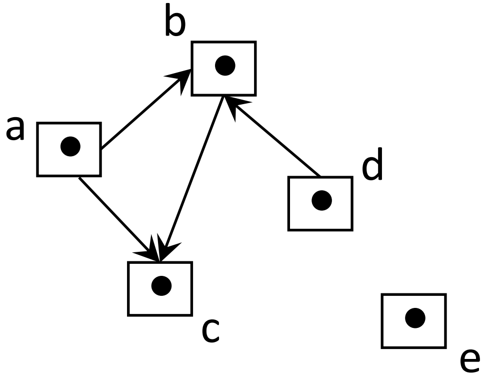


- Tree
  - for every pair of vertices v_i and v_j, there is exactly one path from v_i to v_j



Graph algorithms

- Max-flow: maximum flow rate between source and destination in graph weighted with capacities
- Shortest path: find shortest path between source and destination in graph weighted with distances
- Topological sort: linear order on vertices of digraph such that "precedes" relation is respected



topological sort →

d, a, e, b, c

a, d, b, c, e

…

What was added?

- value arrays (discussed in chapter 12.6)
- tuple (discussed in chapter 11.3)
- set
- stack
- queue/priority queue

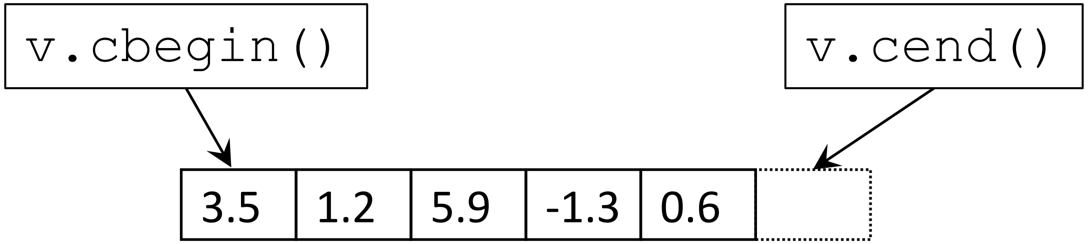
Algorithms

Chapter 10, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Algorithms

Iterators

```
vector<double> v {3.5, 1.2, 5.9, -1.3, 0.6};
for (auto it = v.cbegin(); it != v.cend(); ++it)
    cout << *it << endl;
```



v.cbegin()

v.cend()

3.5	1.2	5.9	-1.3	0.6	

`it` contains address of element (pointer): value `*it`

Sorting

```
#include <algorithm>
…
vector<double> v {3.5, 1.2, 5.9, -1.3, 0.6};
sort(v.begin(), v.end());
for (auto& it = v.cbegin(); it != v.cend(); ++it)
    cout << *it << endl;
```

- `cbegin()/cend()`
  - constant iterator
  - elements will not be modified
- `begin()/end()`
  - elements can be modified

Use `const` iterators whenever possible

Defining order

- Define data structure

```
struct Particle {
    double x, y, mass;
};
vector<Particle> particles = init_particles(n);
```

- Define order relation on mass

```
bool mass_cmp(const Particle& p1, const Particle& p2) {
    return p1.mass < p2.mass;
}
```

- Sort on mass

```
sort(particles.begin(), particles.end(), mass_cmp);
```

Finding things

- Predicate find

```
vector<int> data {…};
if (find_if(data.cbegin(), data.cend(),
            [] (int x) { return x < 0; }) != data.cend())
    cout << "found!" << endl;
```

- Sequence search

Can use Boyer-Moore algorithm

```
const string dna {…};
const string subseq {"ACCGTA"};
auto it = search(dna.cbegin(), dna.cend(),
                 subseq.cbegin(), subseq.cend());
if (it != dna.cend())
    cout << "found!" << endl;
```

Similar: `find`, `count`, `count_if`, …

Transformation

- Single container

```
array<int, 10> v1 {…};
array<int, 10> v2;
transform(v1.cbegin(), v1.cend(), v2.begin(),
          [] (int x) -> int { return x*x; });
```

- Two containers (aka zip)

Similar: `foreach`, `replace`, `replace_if`, …

```
array<double, 10> v1 {…};
array<double, 10> v2 {…};
array<double, 10> v3;
const double w1 {…};
const double w2 {…};
transform(v1.cbegin(), v1.cend(), v2.cbegin(),
    v3.begin(),
    [=] (double x, double y) { return w1*x + w2*y; });
```

Other algorithms

- `all_of`, `any_of`, `none_of`: check predicate on collection
- `mismatch`: find position where sequences differ
- `equal`: check equality of sequences
- `copy`, `move`: copy, move sequence to other sequence
- `remove`, `remove_if`: remove elements
- `shuffle`: random shuffle sequence
- `accumulate`, `inner_product`
- many more, even more in C++17!

Ranges

- Problem C++17

```
Data data(20);
…
Data t1;
std::copy_if(data.begin(), data.end(),
             std::back_inserter(t1), is_even);
Data t2(t1.begin() + skip, t1.end());
for (auto it = t2.rbegin(); it != t2.rend(); ++it) {
    std::cout << *it << " ";
}
```

Temporary variables!

- C++20 introduces ranges

```
std::ranges::reverse_view rv {
    std::ranges::drop_view {
        std::ranges::filter_view {data, is_even}, skip
    }
};
for (const auto& value: rv) {
    std::cout << value << " ";
}
```

Views

Performance boost!

- Ranges: inside out

```
std::ranges::reverse_view rv {
    std::ranges::drop_view {
        std::ranges::filter_view {data, is_even}, skip
    }
};
for (const auto& value: rv) {
    std::cout << value << " ";
}
```

- Views: more clear (some C++23 feature)

```
 for (const auto& value: data
           | std::views::filter(is_even)
           | std::views::drop(skip)
           | std::views::reverse) {
    std::cout << value << " ";
}
```

Much more compact/elegant

Almost Python

```
td::vector<char> data {'a', 'b', 'd', 'z’};
for (const auto [id, value]: std::views::enumerate(data)) {
    std::cout << id << " -> " << value << "\n";
}
```

What was left out/added?

- Left out
  - stream iterators
  - discussion of iterator types
- Added
  - extra examples
  - Ranges, views

References

- Introduction to algorithms
  Thomas H. Cromen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
  MIT Press, 2009 (3rd edition)

Numerics

Chapter 12, B. Stroustrup "A tour of C++"

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Numerics

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Armadillo

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/Boost

https://github.com/gjbex/Scientific-C-plus-plus/tree/master/source-code/UsingCLibraries

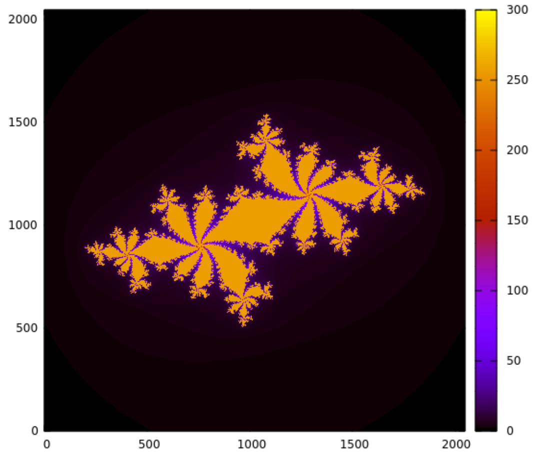
Complex numbers

```
#include <complex>
…
using namespace std;
…
const complex<double> c(-0.62772, -0.42193);
for (double x = -1.8; x < 1.8; x += 0.001)
    for (double y = -1.8; y < 1.8; y += 0.001) {
        complex<double> z(x, y);
        while (abs(z) < 2.0 && n++ < max_n)
            z = z*z + c;
        cout << x << " " << y << " " << n << endl;
    }
```

(Overloaded) math functions

More efficient:

```
real(z)*real(z) + imag(z)*imag(z) < 4.0
```



Numerical limits

```
#include <limits>
```

- **Integer:** `int`, `long`, `int8_t`, `int16_t`, `int32_t`, `int64_t`
  - minimum: `std::numeric_limits<int>::min()`
  - maximum: `std::numeric_limits<int>::max()`
- **Floating point:** `float`, `double`, `long double`
  - smallest number > 0: `std::numeric_limits<double>::min()`
  - maximum: `std::numeric_limits<double>::max()`
  - $1 < 1 + \varepsilon$: `std::numeric_limits<double>::epsilon()`
  - significant digits, base 10: `std::numeric_limits<double>::digits10`
  - `isfinite(…)` : true if not ±infinity, or NaN

Limit values

usually `short`

usually `int`

usually `long`

	int8_t	int16_t	int32_t	int64_t
min()	-128	-32768	-2147483648	-9223372036854775808
max()	127	32767	2147483647	9223372036854775807

$\sim 10^9$

$\sim 10^{19}$

	float	double	long double
digits10	6	15	18
min()	1.176e-38	2.225e-308	3.362e-4932
epsilon()	1.192e-07	2.221e-16	1.084e-19
max()	3.403e+38	1.798e+308	1.190e+4932

performance penalty!

32-bit

64-bit

96-bit

More precision?

- Possible, but at high cost
  - performance
  - development
- Consider other algorithms first
- Libraries for arbitrary precision arithmetic
  - GMP: for integers
  - MPFR: for floating point numbers
  - MPC: for complex floating point numbers

Random number generation

- Engine: generates random number sequence
  - `std::random_device`: non-deterministic
  - `std::ranlux48`
  - `std::mt19937_64`: Mersenne twister
  - …
- Distributions
  - `uniform_int_distribution<type>(a, b)`
  - `uniform_real_distribution<type>(a, b)`
  - `normal_distribution<type>(mu, sigma)`
  - …

Typical workflow

1. Create random device
2. Create seed distribution
3. Draw seed from seed distribution using random device
4. Create engine, seed
5. Create actual distribution
6. Draw random number from actual distribution using engine

Example: normal distribution

```
  #include <functional>
  #include <random>
  …
  using seed_dist_t = uniform_int_distribution<size_t>;
  …
① random_device dev;
② seed_dist_t seed_distr(0, numeric_limits<size_t>::max());
③ auto seed = seed_distr(dev);
  cout << seed << endl;
④ mt19937_64 engine(seed);
  auto distr = bind(normal_distribution<double>(0.0, 1.0),
⑤                   engine);
  for (int i = 0; i < 5; i++)
⑥     cout << distr() << endl;
```

Multiple distributions

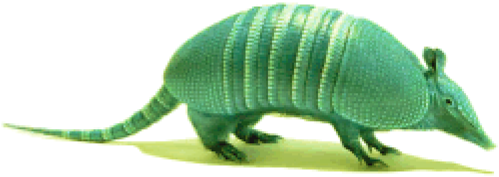
- `bind` binds by value, i.e., copies, unless wrapped

```
#include <functional>
#include <random>
…
using seed_dist_t = uniform_int_distribution<size_t>;
…
mt19937_64 engine(seed);
auto x_distr = bind(normal_distribution<double>(0.0, 1.0),
                    ref(engine));
auto y_distr = bind(normal_distribution<double>(0.0, 2.0),
                    ref(engine));
```

Without `ref(…)`, both `x_distr` and `y_distr` produce same numbers!

Linear algebra

- Several libraries, don't do your own!
  - Eigen (http://eigen.tuxfamily.org/)
    - purely header files
    - trivial to install
  - Armadillo (http://arma.sourceforge.net/)
    - uses BLAS/Lapack
    - quite convenient
    - good performance
    - no distributed algorithms
  - …



Here: a flavor of Armadillo

Data types

- Vectors
  - `Col<type>`, `colvec`, `vec`
  - `Row<type>`, `rowvec`
- Matrices
  - dense: `Mat<type>`, `mat`
  - sparse: `SpMat<type>`, `sp_mat`
- Cubes (3D arrays)
  - `Cube<type>`, `cube`
- Fields (2D or 3D arrays, arbitrary objects)
  - `Field<obj_type>`

shortcuts: `type` is `double`

```
#include <armadillo>
…
using namespace arma;
…
```

`type` is scalar

`obj_type` is arbitrary

Initialization

- Literal initialization

```
vec v {7.3, 9.1};
mat A {{-1.0, 3.1, 4.3}, {2.1, -2.4, 0.9}};
```

- Generated vectors

```
vec x = linspace<vec>(-1.0, 1.0, 501);
vec y = regspace<vec>(0.0, 0.1, 1.0);
```

- Generated matrices

```
mat A = eye<mat>(5, 5);
```

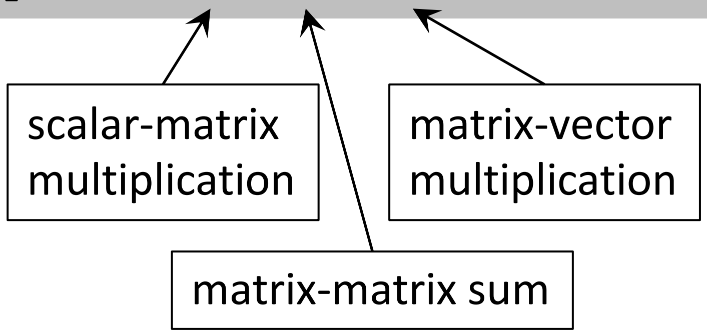
- Generated vector/matrices/cubes

```
mat A = randn<mat>(2, 3);
vec x = randu<vec>(5);
vec y = zeros<vec>(10);
mat C = ones<mat>(3, 2);
```

Note resemblance
to MATLAB, numpy

Matrix arithmetic/functions

```
mat A {{-1.0, 3.1, 4.3}, {2.1, -2.4, 0.9}};
mat B {{2.1, -2.0, 0.2}, {0.1, 3.1, -1.7}};
vec x {7.3, 9.1, -3.3};
vec y = (2.0*A + B)*x;
```



scalar-matrix multiplication

matrix-vector multiplication

matrix-matrix sum

Operator overloading for convenient mathematical expressions

```
vec x = randn(10);
vec y = randn(10);
double distance = norm_dot(x, y);
```

Many other math functions: `abs`, `det`, `norm`, `dot`, `min`, `max`, `sum`,…

Matrix access

```
…
for (size_t j = 0; j < A.n_cols; j++)
    for (size_t i = 0; i < A.n_rows; i++)
        A(i, j) = f(i, j);
```

Note: elements stored column wise

```
…
mat B = A.submat(span(min_row, max_row),
                 span(min_col, max_col));
rowvec x = A.row(row_nr);
vec y = A.col(col_nr);
```

```
…
double a, b, c;
…
A.transform([=] (double x) { return a*x*x + b*x + c; });
```

Linear algebra

- Many decomposition methods, e.g., SVD

```
…
mat A(nr_rows, nr_cols);
…
mat U, V;
vec s;
svd(U, s, V, A);
mat S = diagmat(s);
mat A_p = (U*S)*V.t();
```

- Matrix transpose: `A.t()`
- Matrix inverse: `A.i()`

ODEs with Boost::odeint

- Declarations

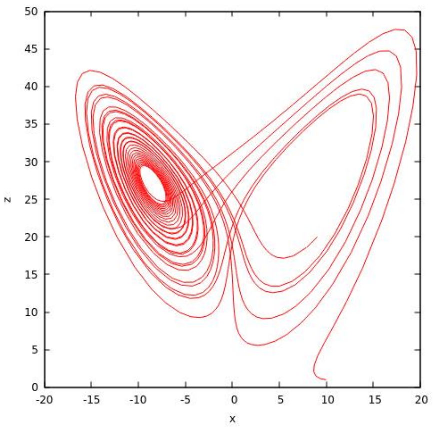
```
#include <array>
#include <functional>
#include <boost/numeric/odeint.hpp>
using namespace boost::numeric::odeint;
using state_type = array<double, 3>;
```

- Define equations

```
void lorenz_param(const state_type& x, state_type& dxdt, double t,
                  double sigma, double R, double b) {
    dxdt[0] = sigma*(x[1] - x[0]);
    dxdt[1] = R*x[0] - x[1] - x[0]*x[2];
    dxdt[2] = -b*x[2] + x[0]*x[1];
}
```

RHS of ODEs

Solving ODEs



- Writing steps

```
void write_lorenz(const state_type& x, const double t) {
    cout << t << '\t' << x[0] << '\t' << x[1] << '\t' << x[2]
         << endl;
}
```

- Integration

```
const double sigma = 10.0;
const double R = 28.0;
const double b = 8.0/3.0;
auto lorenz = [=] (const state_type& x, state_type& dxdt, double t) {
    return Lorenz_param(x, dxdt, t, sigma, R, b);
};
state_type x = { 10.0, 1.0, 1.0 };
integrate(lorenz, x, 0.0, max_t, delta_t, write_lorenz);
```

GNU Scientific Library

- Large collection of algorithms for scientific computing
  - numerical integration
  - minimizing functions
  - interpolation
  - statistics
  - linear algebra
  - solvers for ordinary differential equations
  - Fourier transforms
  - …
- However, C library, not C++
  - some tinkering required

Finding minimum with GSL

- Declarations

```
…
#include <gsl/gsl_errno.h>
#include <gsl/gsl_min.h>
…
double func(const double x, void *params);
…
```

function signature expected by minimizer

- Function definition

```
double func(const double x, void *params) {
    auto params_arr = static_cast<double*>(params);
    double a {params_arr[0]};
    double b {params_arr[1]};
    double c {params_arr[2]};
    return (a*x + b)*x + c;
}
```

Setting up minimizer

- Function to minimize

```
double params[] {1.0, -1.0, 1.0};
gsl_function F {
    .function = &func,
    .params = params
};
```

should be
`double (*) (const double, void*)`

- Minimizer

```
auto minimizer {gsl_min_fminimizer_alloc(gsl_min_fminimizer_brent)};
int status {gsl_min_fminimizer_set(minimizer, &F, x, x_min, x_max)};
if (status == GSL_EINVAL) {
    std::cerr << "### error: interval [" << x_min << ", " << x_max
              << "] doesn't contain a minimum" << std::endl;
    std::exit(GSL_EINVAL);
}
```

Finding minimum

- Iterating

```
int status;
int iter_nr {0};
do {
    iter_nr++;
    gsl_min_fminimizer_iterate(minimizer);
    x_min = gsl_min_fminimizer_x_lower(minimizer);
    x_max = gsl_min_fminimizer_x_upper(minimizer);
    status = gsl_min_test_interval(x_min, x_max, 1e-6, 0.0);
} while (status == GSL_CONTINUE && iter_nr < nr_iters);
```

absolute error

relative error

- minimum location

```
if (status == GSL_SUCCESS) {
    x = gsl_min_fminimizer_x_minimum(minimizer);
}
```

What was left out/added?

- Left out
  - Value arrays, see section on containers
- Added
  - Linear algebra with Armadillo
  - ODEs with Boost
  - Mixing C and C++ code, using GSL

Conclusions

Conclusions

- C++: nice for scientific computing
  - modern programming language
  - good standard library
  - data processing relatively easy
- However, much more to learn
  - this is but a starting point!
  - performance issues can be non-trivial

Additional topics

- Concurrency: for scientific code use
  - OpenMP
  - TBB (Threading Building Blocks
- Create your own containers/data structures
- Good object oriented design
  - for large software systems

Further reading

- *A tour of C++, 3rd edition*
  Bjarne Stroustrup
  Addison-Wesley, 2022
- *Effective modern C++*
  Scott Meyers
  O'Reilly Media, 2015
- C++ reference
- *C++ core guidelines*
  Bjarne Stroustrup, Herb Sutter
- Google C++ Style Guide
- https://isocpp.org/wiki/faq

More reading

- *C++ templates: the complete guide, 2nd edition*
  David Vandevoorde, Nicolas M. Jossutis, Douglas Gregor
  Addison-Wesley, 2018
- *The C++ programming language, 4th edition*
  Bjarne Stroustrup
  Pearson Education, 2013
- *Introduction to algorithms, 4th edition*
  Thomas H. Cromen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
  MIT Press, 2022

Online learning resources

- http://www.cplusplus.com/
- https://www.tutorialspoint.com/cplusplus/cpp_overview.htm

Tools

- Compilers
  - GCC g++ (https://gcc.gnu.org/)
  - Intel OneAPI compilers (https://software.intel.com/en-us/c-compilers)
  - clang++ (https://clang.llvm.org/)
  - Compiler Explorer (https://godbolt.org/)
- Interpreter
  - Cling (https://github.com/vgvassilev/cling)
- Online compilers
  - Wandbox (http://wandbox.org/)
  - Tutorialspoint (https://www.tutorialspoint.com/cplusplus/cpp_overview.htm)
  - CodeChef (https://www.codechef.com/ide)
  - Replit (https://replit.com/)
- Static code checkers
  - Cppcheck (http://cppcheck.sourceforge.net/)
- IDEs
  - Jetbrains CLion (https://www.jetbrains.com/clion/)
  - Microsoft Visual Code (https://code.visualstudio.com/)
  - Eclipse (https://www.eclipse.org/ide/)