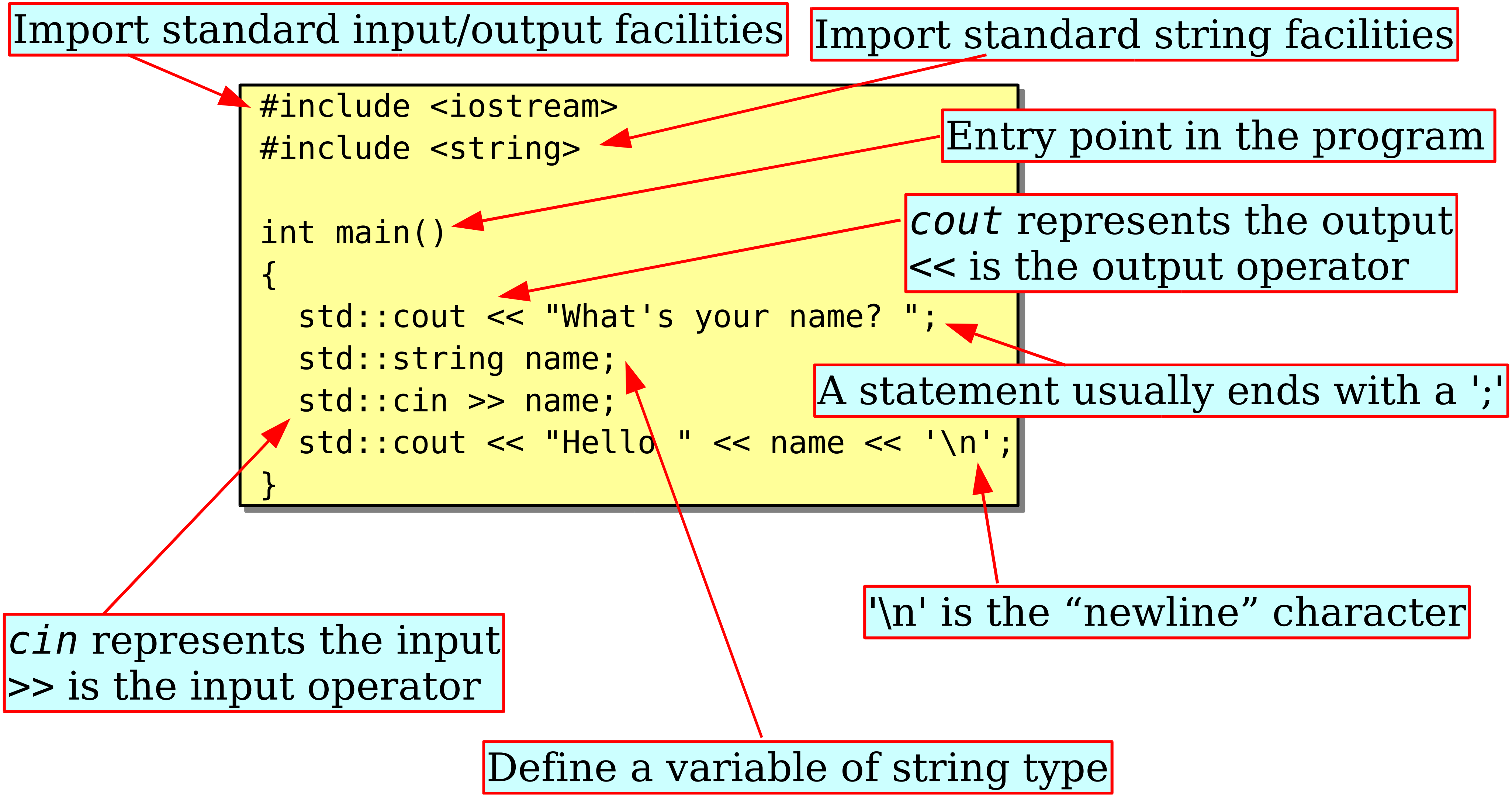
A starting example

- A program that:
    - Writes a prompt to enter your name
    - Reads the name
    - Writes out “Hello <name>”
- code/first_example/
    - Build it with *`c++ main.cpp`*
- What happens if:
    - You don't give any name?
    - You give more than one name?



Import standard input/output facilities

Import standard string facilities

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

int main()
{
  std::cout << "What's your name? ";
  std::string name;
  std::cin >> name;
  std::cout << "Hello " << name << '\n';
}
```

Entry point in the program

cout represents the output
<< is the output operator

A statement usually ends with a ';'

'\n' is the “newline” character

cin represents the input
>> is the input operator

Define a variable of string type

- Standard facilities are imported with an appropriate *`#include`* directive
  - Typically they correspond to a file
- The entry point in a program is a function called *`main`*
  - One possible form of *`main`* does not take any argument and returns an integer
- A variable must be declared before being used
- *`cin`* and *`cout`* represent (standard) input and output respectively

- Operators **>>** and **<<** are used to read from input and to write to output
- Multiple reads and multiple writes can be chained one after the other
- A statement usually ends with a semicolon (;)

Identifiers

- An identifier is a user-defined name that denotes program entities (variable, functions, classes, namespaces...)
- An identifier is composed of one or more characters
  - The first character must be a letter (letters include the underscore '_')
    - C++ is case sensitive
  - The other characters can be letters or numbers
  - The length is unlimited (system-specific)

Identifiers (2)

- C++ keywords and alternative tokens cannot be used as identifiers
- Names that:
  - start with an underscore and a capital letter or
  - contain two consecutive underscores

  are reserved

C++ keywords

asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while

C++ alternative tokens

and
not_eq
and_eq
or
bitand
or_eq
bitor
xor
compl
xor_eq
not

Another example

- code/second_example/
- What does this program do?
- Try it
  - Build it with *`c++ main.cpp`*
- What happens if there is no strings.txt?
  - What happens if there is no test?
- Read from standard input rather than from file
- Read strings rather than whole lines
- Read integers rather than strings

Import standard vector facilities

Import standard file I/O facilities

Import some standard algorithms

C++-style comment

Open strings.txt for reading

Check if the file was correctly opened

cerr represents the output for errors

```
#include <vector>
#include <string>
#include <fstream>
#include <algorithm>       // for sort
#include <iostream>

int main()
{
  std::ifstream input_file("strings.txt");
  if (!input_file) {
    std::cerr << "Cannot open file strings.txt\n";
    return 1;
  }
  std::vector<std::string> v;
  std::string line;
  while (getline(input_file, line)) {
    v.push_back(line);
  }
  sort(v.begin(), v.end());
  for (int i = 0; i < v.size(); ++i) {
    std::cout << v[i] << '\n';
  }
}
```

Exit from main (and from the program) with status 1

We keep the strings in a vector

Read a line at a time from the input file

Append it to the vector

Sort the vector (in place)

Print all the strings

Implicitly return 0 (valid only for main)

Some abstractions...

- Procedural abstraction, e.g. getline, sort
- Data abstraction, e.g. vector, string, ifstream
- Type abstraction, e.g. getline, sort, vector, iterators, string, ifstream
  - Abstractions can be combined

- The C++ library offers a rich set of predefined algorithms
- A C++-style comment starts at the token // and goes until the end of the line
- C-style comments are also supported
  - The comment is enclosed between the tokens /* and */, also on multiple lines
- A variable can be declared just when it's needed
  - This is recommended

- An expression can be automatically converted to a boolean value (true/false), e.g. when testing for *`if`*, *`while`* and *`for`* conditions
  - The conversion can be built-in or user-defined
- The operator *`!`* (not) negates a boolean value
- ++ is the increment operator
  - The expression ++*`i`* increments *`i`* by 1

- *`vector`* is a standard container
  - Others exist: *`list, map, set`*…
- *`[]`* is the subscript operator
  - *`v[i]`* accesses the *`i`*th element of *`v`*
- *`begin()`* and *`end()`* denote the first and one beyond the last element of the vector
  - The denoted range is half-open

Objects, variables, types

- An object is a region of storage
- An object is created by a definition
  - Or by other means we'll see later
- The properties of an object (i.e. its type) are determined when the object is created
  - A type defines the proper use of an object
- A variable is introduced by the declaration of an object. The variable's name denotes the object

Built-in types

- A boolean type (*`bool`*)
  - *`true`*, *`false`*
- Character types (*`char`* and *`wchar_t`*; *`signed`*, *`unsigned`*)
  - 'a', '5', '\n', '\t'
- Integer types (*`short`*, *`int`*, *`long`*; `signed`, `unsigned`)
  - 234, -7483, **0**456 (octal), **0x**deaf (exa), 878**U** (force unsigned), 8475**L** (force long)
- The above are integral types

Built-in types

- Floating-point types (*`float`*, *`double`*, *`long double`*)
  - 1., 1.0, -2.4637, .7e-2, .7**F** (force float), .7**L** (force long double)
- Integral types and floating-point types are arithmetic types

```
int i;             // definition; its value may be undefined
int i = 0;         // definition; initialized to zero
int i(0);          // definition; initialized to zero
int i = int();     // definition; initialized to zero
extern int i;      // declaration only
double d;          // definition; its value may be undefined
double d = 3.;     // definition; initialized to 3.
char c = 'f';      // definition; initialized to 'f'
bool b = true;     // definition; initialized to true
```

Built-in types

- void
  - Special type to mean that no type information is available
    - e.g. No return value from a function

```
void v;    // error
void f();  // ok, function declaration with no return value
```

User-defined types

- It's possible to combine built-in types to construct other, user-defined types
  - Pointers
  - Arrays
  - References
  - Data structures and classes
- User-defined types can themselves be the basis for further aggregations

Expressions

- An expression is a sequence of operators and operands that specifies a computation
- An expression can result in a value and cause side-effects
- Expression operands are variables and literals
- Appropriate conversions (build-in and user-defined) are executed to adjust the type of the operands

- The order of evaluation of subexpressions within an expression is undefined
- The precedence of operators is the “usual one”
  - Arithmetic > logical > assignment
  - If in doubt, use parenthesis

(Some) Operators

- Arithmetic

  `++ -- + - * / %`
- Binary

  `~ | & ^ >> <<`
- Logical

  `! && ||`
- Comparison

  `== != < > >= <=`
- Assignement

  `= += -= *= /= %= <<= >>= &= |= ^=`
- Subscript

  `[]`
- Conditional expr

  `?:`
- Function call

  `()`
- Scope resolution

  `::`

Statements

- Label
- Expression
- Compound (block)
- Selection
  - if, switch
- Iteration
  - while, do-while, for
- Jump
  - break, continue, goto
- Declaration
- Try block

Expression statement

```
<expression> ;
```

- Typically <*expression*> is an assignment or a function call

```
y = x + 1;
f(x);
```

- The result of the expression is thrown away
- <*expression*> can be missing (empty statement

Compound statement

```
{ <statement list> }
```

- So that several statements can be used where one is expected
  - Function body, *`if`* block, *`while`* block...
- A compound statement defines a local scope

if (selection statement)

```
if (<condition>) <statement>
if (<condition>) <statement> else <statement>
```

- If the condition is true the first statement is executed
- If the condition is false the second statement, if present, is executed
- Condition must be convertible to bool

```
if (error) {
  std::cerr << “An error occurred\n”;
}
```

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

switch (selection statement)

```
switch ( <condition> ) <statement>
```

- But typically *<statement>* is a sequence of *case* statements and an optional *default* statement

```
switch (n) {
case 1:
  std::cout << “one\n”;
  break; // without this it would continue through case 2
case 2:
  std::cout << “two\n”;
  break;
default:
  std::cout << “unknown\n”
}
```

Switch and *enum*

- The *switch* statement and an *enum* type work well together

```
enum rgb { red, green, blue };
rgb color;
switch (color) {
case red:
  std::cout << “red\n”;
  break;
case green:
  std::cout << “green\n”;
  break;
case blue:
  std::cout << “blue\n”
  break;
}
```

The compiler warns if some cases are left out

Iteration statements

```
while (<condition>) <statement>
```

- *<statement>* is executed as far as *<condition>* is true
  - *<statement>* is executed zero or more times

```
do <statement> while (<expression>)
```

- *<statement>* is executed as far as *<expression>* is true
  - *<statement>* is executed one or more times
  - *<expression>* must be convertible to *bool*

for (iteration statement)

```
for (<initialization>; <condition>; <expression>) <statement>
```

- Roughly equivalent to

```
{
  <initialization>;
  while (<condition>) {
    <statement>
    <expression>;
  }
}
```

- A *for* loop is preferred to a while loop when there is a variable controlling the loop

Jump statements

`break;`

- Exits from an iteration or from a switch

`continue;`

- Terminates the current iteration of a loop

`return <expression>;`

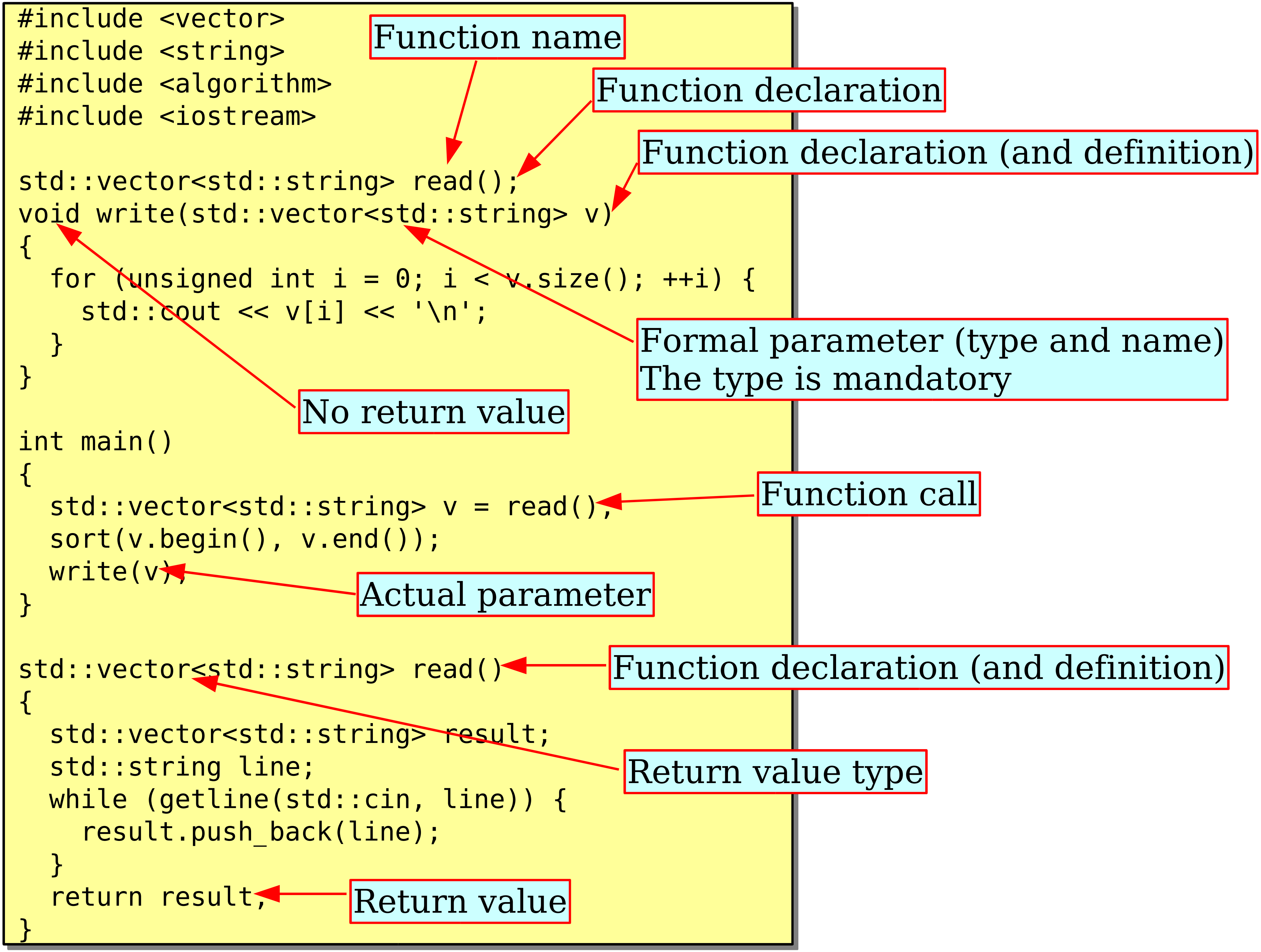
- Returns from a function with a return value equal to the result of `<`*`expression`*`>`

`goto <label>;`

- Jump the execution to `<`*`label`*`>`
- ARE YOU SURE YOU WANT TO USE IT?

Functions

- A function is the C++ mechanism to support code abstraction
  - Inherited from C
- The function internal workings are hidden to its clients
- A function may take one or more input parameters
- A function may return a value
- A function may have side-effects
  - Changes to entities not explicitly mentioned in its parameter list



```
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

std::vector<std::string> read();
void write(std::vector<std::string> v)
{
  for (unsigned int i = 0; i < v.size(); ++i) {
    std::cout << v[i] << '\n';
  }
}

int main()
{
  std::vector<std::string> v = read();
  sort(v.begin(), v.end());
  write(v);
}

std::vector<std::string> read()
{
  std::vector<std::string> result;
  std::string line;
  while (getline(std::cin, line)) {
    result.push_back(line);
  }
  return result;
}
```

Function name

Function declaration

Function declaration (and definition)

Formal parameter (type and name)
The type is mandatory

No return value

Function call

Actual parameter

Function declaration (and definition)

Return value type

Return value

- As for variables, a function must be declared before its use
- A function declaration is a definition if it specifies also the function body
- The client calls a function passing appropriate parameters in the right order
  - Formal parameters are initialized with the actual parameters
- The client can ignore the return value
- Reading from `cin` and writing to `cout` are examples of side effects

- Functions can be overloaded
  - Same name, different number and/or type of parameters
  - The compiler will choose the best match for a call
  - The return type is not involved

```
std::vector<int> read();
std::vector<int> read(int max_numbers);
std::list<int> read(); // error
```

- Function parameters can have default values

```
std::vector<int> read(int max_numbers = -1);
void write(std::vector<int> v = std::vector<int>());
```

Pass by value vs Pass by reference

- The body of the function *`write()`* works on a copy of its parameter, i.e. a copy of the vector of strings is made before calling *`write()`*
  - No changes visible to the client
- If changes need to be visible to the client or the parameter is not copyable or the parameter is “big”, another mechanism is available (passing a “reference” to the original)

References

- A reference declaration introduces an alternative name for an already declared object
- Must be initialized
- Cannot be changed after initialization
  - always bound to the initial object

```
int i;
int& ri;     // error
int& ri = i; // ok
int j;
ri = j;      // assigns j to i!
int& rc = 1; // error, otherwise we could modify a constant
int const& rc = 1; // ok, a temporary int is created
```

How to pass parameters to functions

- Rule of thumb (there are exceptions)
- If changes need not be visible to the caller:
  - Pass built-in types by value
  - Pass non-built in types by const reference
- If changes need to be visible to the caller
  - Pass always by reference

How to structure source code

- Usually it is convenient to split source code in more than one file, possibly in more than one directory
  - C++ supports separate compilation of semi-independent modules
- Usually it is convenient to factor out some functionality into a reusable “library”
- Usually it is convenient to “export” the interface of the abstraction without showing the actual implementation

Header and source files

- Given a certain functionality:
  - Its interface goes into a “header file”
  - Its implementation goes into a “source file”
- Clients of that functionality will include the header file
- To ensure consistency, the source file includes the header file as well

Data abstraction

- Hide internal representation of an object and allow its manipulation only via its public interface
- Let's implement an abstraction for a complex number
- Start from how we want to use that abstraction
  - Creation of numbers
  - Manipulation
  - Assignments
  - Operations

Class

```
class Complex
{
public:
  // public member declarations
  // other public stuff
private:
  // private member declarations
  // other private stuff
};
```

- By default class members are private
- The `class` keyword can be replaced by `struct`
  - By default class members are public

```
Complex c1;                 // (0.0,0.0)
Complex c2(1.0);            // (1.0,0.0)
Complex c3(1.0, 2.0);       // (1.0,2.0)
Complex c4(c3);             // (1.0,2.0)
Complex c5 = c3;            // (1.0,2.0)
```

Creation

Assignment

```
c1 = c2;
c1 = 1.0;
```

```
c1 + c2;
c1 - c2;
c1 * c2;
c1 / c2;
sqrt(c1);
```

Operations

And also...

```
c1 += c2;
c1 = c2 / c3 + c4;
write(c1);
c1 = read();
c1 = sqrt(c1) * (c2 + c3);
```

Class constructor

- A constructor is a special member function that has the same name as the class
  - No return value
- It is invoked at object creation time to initialize the storage allocated to the object
- It can be overloaded
  - And often it is

Special forms of ctors

- Default
  - No parameters
  - Automatically generated if no other ctor is available

```
Complex c1;
Complex c1 = Complex();
```

- Copy ctor
  - Takes a const reference to another instance of the same class
  - Used to initialize an object as a copy of an existing one
  - Automatically generated if not defined

```
Complex c1;
Complex c2(c1);
Complex c2 = c1;
Complex c2 = Complex();
```

Destructor

- A destructor is is a special member function that has the same name as the class prefixed with the character ~
  - No return value
  - No parameters
- It is invoked when the object is about to disappear to release possible resources owned by the object
  - Dynamic memory, files, locks, sockets...
- Automatically generated if not defined

Assignment operator

- Another special member function is the assignment operator (*`operator=()`*)
  - Takes a const reference to another instance of the same class
  - It should return a reference to itself to allow chaining of assignments
- Used to assign an object to another one (already existing)

```
Complex c1; Complex c2;
c1 = c2;
```

- Automatically generated if not defined

User-defined conversion

- One-parameter ctors can be used for user-defined conversions

```
void f(Complex c);
class Complex {
  Complex(double);
};
Complex c(1.);  // calls Complex(double);
Complex c = 1.; // equivalent to Complex c = Complex(1.);
f(1.);          // equivalent to f(Complex(1.));
```

- An *`explicit`* ctor disables this possibility

```
void f(Complex c);
class Complex {
  explicit Complex(double);
};
Complex c(1.);  // calls Complex(double);
Complex c = 1.; // error
f(1.);          // error
```

Binary operators

- Binary operators such as `+` `-` `*` `/` should be implemented as free functions to allow conversion for both operands

```
class Complex {
  Complex operator+(Complex const& rhs);
};
Complex c1, c2;
c1 + c2; // ok, calls c1.operator+(c2);
c1 + 1.; // ok, calls c1.operator+(Complex(1.));
1. + c2; // error, no conversion on the first operand
```

```
class Complex { /* ... */ };
Complex operator+(Complex const& lhs, Complex const& rhs);
Complex c1, c2;
c1 + c2; // ok, calls operator+(c1, c2);
c1 + 1.; // ok, calls operator+(c1, Complex(1.));
1. + c2; // ok, calls operator+(Complex(1.), c2);
```

- Binary operators such as `+ - * /` produce new objects
  - To be returned by value
- Binary operators such as `+ - * /` should be implemented in terms of `+= -= *= /=`
  - Code reuse
- operator++

Include guard

- If the header file is not adequately protected multiple definitions can happen (violation of ODR)

```
#include “complex.hpp”
#include “complex.hpp” // causes compilation error
                       // class Complex is defined twice
```

- All header files should be written as follows:

```
#ifndef COMPLEX_HPP
#define COMPLEX_HPP
// previous stuff here
#endif
```

Namespaces

- Namespaces are a mechanism to partition the space of names in a C++ program
  - The same name can be chosen in different places (possibly by different people, parties, vendors)

```
// complex.hpp
namespace math {
class Complex { /* usual stuff here */ };
Complex operator+(Complex const&, Complex const&);
//...
}
```

- Argument Dependent Lookup
  - aka Koenig lookup
  - Unqualified functions/operators are looked for also in the argument's namespace(s)

```
math::Complex c1;
math::Complex c2;
c1 + c2; // calls math::operator+(c1, c2)
```

- Using directive
  - “imports” specified symbol

```
using math::Complex;
Complex c;
```

- Using declaration
  - “imports” all the symbols in the specified namespace

```
using namespace math;
Complex c;
```

- Namespaces can be nested

```
namespace math {
  namespace advanced {
    class Complex { /* usual stuff here */ };
    Complex operator+(Complex const&, Complex const&);
    //...
  }
}
namespace ma = math::advanced; // namespace alias
ma::Complex
```

- Anonymous namespace
  - Guarantees the uniqueness of the namespace name in the TU where that code ends up

```
namespace {
  void f();
}
f();
```

```
namespace <some unique name> {
  void f();
}
using namespace <some unique name>
f();
```

Dynamic memory

- Sometimes it is useful to create objects that could survive the current scope
- Such objects are created with the *new* operator
- Such objects need to be explicitly destroyed with the *delete* operator

```
class X { /* ... */ };
X* make_X() { return new X; }
void f()
{
  X* pointer_to_x = make_X();
  // use pointer_to_x here
  delete pointer_to_x;
}
```

- The return value of the *new* expression applied to type *T* is a pointer to *T* and is denoted by *T**
- The use of pointers is very error-prone

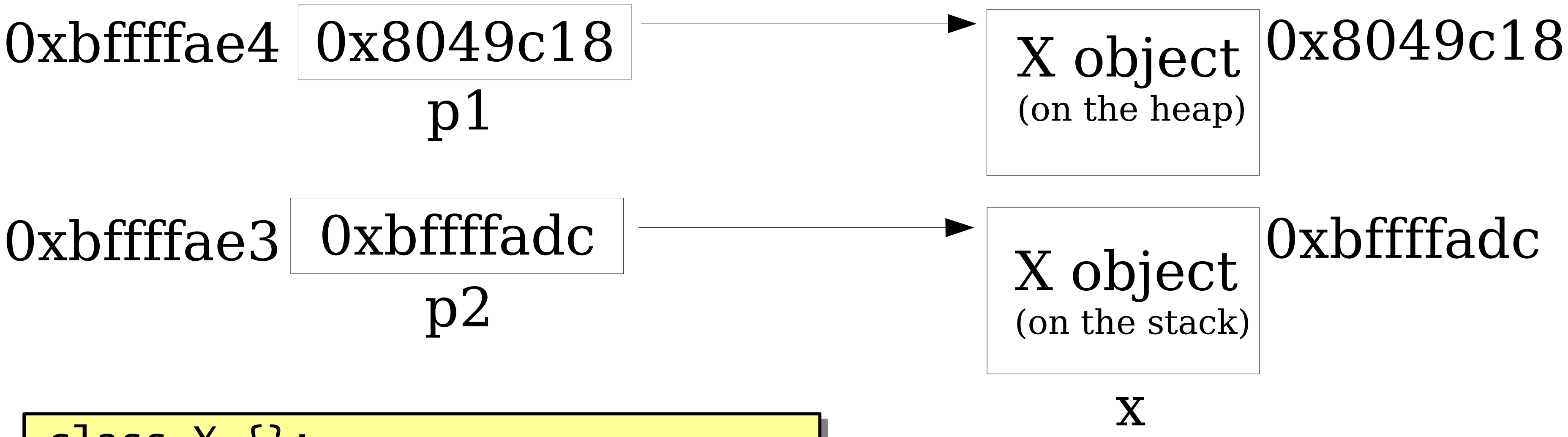
```
X* p = new X;
p = new X; // ops, how can I delete now the first X?
```

```
X* p = new X;
// forgot to delete; memory leak
```

```
X* p = new X;
delete p;
delete p; // ops; double delete
```

```
X* p = 0;
delete p; // ok, this is valid and does nothing
```

How to use pointers



0xbffffae4 0x8049c18

p1

X object
(on the heap)
0x8049c18

0xbffffae3 0xbffffadc

p2

X object
(on the stack)
0xbffffadc

x

```
class X {};
int main()
{
  X* p1 = new X;
  X x;
  X* p2 = &x;
  delete p1;
}
```

- *operator&()* (address-of) takes the address of an object

```
class X { int i; };
X* p = new X;
X x = *p;
x.i;
p->i;
(*p).i;
X& x2 = *p;
X const& x3 = *p;
```

- `.` (dot) is the member access operator
  - Cannot be overloaded for user-defined types
- `*` and `->` are the pointer dereference operators
  - *`(*p).i`* is equivalent to *`p->i`*
  - Can be overloaded (and they are!) for user-defined types