C++ for C Programmers

Victor Eijkhout

TACC Training 2019

Introduction

Stop Coding C!

1. C++ is a more structured and safer variant of C:
   There are very few reasons not to switch to C++.
2. C++ (almost) contains C as a subset.
   So you can use any old mechanism you know from C
   However: where new and better mechanisms exist, stop using the old style C-style idioms.

In this course

1. Object-oriented programming.
2. New mechanisms that replace old ones:
   I/O, strings, arrays, pointers.
3. Other new mechanisms:
   exceptions, namespaces, closures, templating

I'm assuming that you know how to code C loops and functions and you understand what structures and pointers are!

About this course

Slides and codes are from my open source text book:

`https://bitbucket.org/VictorEijkhout/`

`textbook-introduction-to-scientific-programming`

General note about syntax

Many of the examples in this lecture need the compiler option `-std=c++17`. This works for both compilers, so:

```
// for Intel:
icpc -std=c++17 yourprogram.cxx
// for gcc:
g++ -std=c++17 yourprogram.cxx
```

There is no reason not to use that all the time.

Minor enhancements

Just to have this out of the way

- There is a `bool` type with values `true`, `false`
- Single line comments:

```
int x = 1; // set to one
```

- Loop variable can be local:

```
for (int i=0; i<N; i++) // do whatever
```

Simple I/O

Headers:

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

Ouput:

```
int main() {
  int OC=4;
  cout << "Hello world (ABEND CODE OC" << OC << ")" << endl;
```

Input:

```
int i;
cin >> i;
```

C standard header files

```
#include <cmath>
#include <cstdlib>
```

But a number of headers are not needed anymore.

Functions

Big and small changes

- Minor changes: default values on parameters, and polymorphism.
- Big change: use references instead of addresses for argument passing.

Parameter passing

Mathematical type function

Pretty good design:

- pass data into a function,
- return result through `return` statement.
- Parameters are copied into the function. (Cost of copying?)
- pass by value
- 'functional programming'

Results other than through return

Also good design:

- Return no function result,
- or return return status (0 is success, nonzero various informative statuses), and
- return other information by changing the parameters.
- *pass by reference*
- Parameters are also called 'input', 'output', 'throughput'.

C++ references different from C

- C does not have an actual pass-by-reference:
  C mechanism passes address by value.
- C++ has 'references', which are different from C addresses.
- The `&` ampersand is used, but differently.
- Asterisks are out:
  rule of thumb for now,
  if you find yourself writing asterisks, you're not writing C++.
  (however, see later)

Reference

A reference is indicated with an ampersand in its definition, and it acts as an alias of the thing it references.

Code:

```
int i;
int &ri = i;
i = 5;
cout << i << "," << ri << endl;
i *= 2;
cout << i << "," << ri << endl;
ri -= 3;
cout << i << "," << ri << endl;
```

Output [basic] ref:

```
5,5
10,10
7,7
```

(You will not use references often this way.)

Parameter passing by reference

The function parameter `n` becomes a reference to the variable `i` in the main program:

```
void f(int &n) {
  n = /* some expression */ ;
};
int main() {
  int i;
  f(i);
  // i now has the value that was set in the function
}
```

Pass by reference example 1

Code:

```
void f( int &i ) {
  i = 5;
}
int main() {

  int var = 0;
  f(var);
  cout << var << endl;
```

Output
[basic] setbyref:

```
5
```

Compare the difference with leaving out the reference.

Pass by reference example 2

```
bool can_read_value( int &value ) {
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status!=0;
}

int main() {
  int n;
  if (!can_read_value(n))
    // if you can't read the value, set a default
    n = 10;
```

Exercise 1

Write a `void` function `swapij` of two parameters that exchanges the input values:

```
int i=2,j=3;
swapij(i,j);
// now i==3 and j==2
```

Optional exercise 2

Write a divisibility function that takes a number and a divisor, and gives:

- a `bool` return result indicating that the number is divisible, and
- a remainder as output parameter.

```
int number,divisor,remainder;
// read in the number and divisor
if ( is_divisible(number,divisor,remainder) )
  cout << number << " is divisible by " << divisor << endl;
else
  cout << number << "/" << divisor <<
      " has remainder " << remainder << endl;
```

More about functions

Default arguments

Functions can have default argument(s):

```
double distance( double x, double y=0. ) {
  return sqrt( (x-y)*(x-y) );
}
  ...
  d = distance(x); // distance to origin
  d = distance(x,y); // distance between two points
```

Any default argument(s) should come last in the parameter list.

Polymorphic functions

You can have multiple functions with the same name:

```
double sum(double a,double b) {
  return (a+b)2; }
double sum(double a,double b,double c) {
  return (a+b+c)/3; }
```

Distinguished by type or number of input arguments: can not differ only in return type.

Const parameters

You can prevent local changes to the function parameter:

```
/* This does not compile:
   void change_const_scalar(const int i) { i += 1; }
*/
```

This is mostly to protect you against yourself.
Const ref has no overhead, no danger of changes:

```
void pass_array_constref( const vector<double>& v ) { /* */ }
```

Parameter passing summary

- Standard mechanism: call by value, copying.
- Using `Type &var`:
  - call by reference,
  - no copy,
  - data in calling environment can be altered.
- Using `const Type &var`:
  - const-reference,
  - by reference so no copy,
  - but data in calling environment can not be changed.

Object-Oriented Programming

Definition of object

An object is an entity that you can request to do certain things. These actions are the *methods* and to make these possible the object probably stores data, the *members*.

When designing an object, first ask yourself: ‘what functionality should this support’.

Object functionality

Small illustration: vector objects.

Code:

```
Vector v(1.,2.); // make vector (1,2)
cout << "vector has length "
     << v.length() << endl;
v.scaleby(2.);
cout << "vector has length "
     << v.length() << endl
     << "and angle " << v.angle()
     << endl;
```

Output
[object] functionality:

```
vector has length 2.23607
vector has length 4.47214
and angle 1.10715
```

Note the 'dot' notation; in a `struct` we use it for the data members; in an object we (also) use it for methods.

Exercise 3

Thought exercise:
What data does the object need to store to do this?
Is there more than one possibility?

Constructor

Use a constructor: function with same name as the class.
Typically used to initialize data members.

```
class Vector {
private: // members
  double x,y;
public: // methods
  Vector( double x,double y )
    : x(x),y(y) {};
```

The synxtax `x(x)` copies the argument to the data member.

Member default values

Class members can have default values, just like ordinary variables:

```
class Point {
private:
  float x=3., y=.14;
private:
  // et cetera
}
```

Each object will have its members initialized to these values.

Member initialization in the constructor

```
class Vector {
private: // members
  double r,theta;
public: // methods
  Vector( double x,double y ) {
    r = sqrt(x*x+y*y);
    theta = atan(y/x);
  }
```

Methods

Functions on objects

Code:

```
class Vector {
private:
  double x,y;
public:
  Vector( double x,double y )
    : x(x),y(y) {};
  double length() {
    return sqrt(x*x + y*y); };
  double angle() {
    return 0.; /* something trig */; };
};

int main() {
  Vector p1(1.,2.);
  cout << "p1 has length "
       << p1.length() << endl;
```

Output
[geom] pointfunc:

```
p1 has length 2.23607
```

We call such internal functions 'methods'.

Data members, even `private`, are global to the methods.

Methods that alter the object

Code:

```
class Vector {
  /* ... */
  void scaleby( double a ) {
    vx *= a; vy *= a; };
  /* ... */
};
  /* ... */
  Vector p1(1.,2.);
  cout << "p1 has length "
       << p1.length() << endl;
  p1.scaleby(2.);
  cout << "p1 has length "
       << p1.length() << endl;
```

Output
[geom] pointscaleby:

```
p1 has length 2.23607
p1 has length 4.47214
```

Methods that create a new object

Code:

```
class Vector {
  /* ... */
  Vector scale( double a ) {
    return Vector( vx*a, vy*a ); };
  /* ... */
  cout << "p1 has length "
       << p1.length() << endl;
  Vector p2 = p1.scale(2.);
  cout << "p2 has length "
       << p2.length() << endl;
```

Output
[geom] pointscale:

```
p1 has length 2.23607
p2 has length 4.47214
```

Exercise 4

Make class `Point` with a constructor

```
Point( float xcoordinate, float ycoordinate );
```

Write the following methods:

- `distance_to_origin` returns a `float`.
- `printout` uses cout to display the point.
- `angle` computes the angle of vector (x, y) with the x-axis.

Exercise 5

Extend the `Point` class of the previous exercise with a method: `distance` that computes the distance between this point and another: if `p,q` are `Point` objects,

```
p.distance(q)
```

computes the distance between them.

Hint: remember the 'dot' notation for members.

Exercise 6

Write a method `halfway_point` that, given two `Point` objects `p,q`, construct the `Point` halfway, that is, $(p+q)/2$.

You can write this function directly, or you could write functions `Add` and `Scale` and combine these.
(Later you will learn about operator overloading.)

Default constructor

```
Vector v1(1.,2.), v2;
cout << "v1 has length " << v1.length() << endl;
v2 = v1.scale(2.);
cout << "v2 has length " << v2.length() << endl;
```

gives (g++; different for intel):

```
pointdefault.cxx: In function 'int main()':
pointdefault.cxx:32:21: error: no matching function for call to
                'Vector::Vector()'
```

Default constructor

The problem is with *v2*:

```
Vector v1(1.,2.), v2;
```

- *v1* is created with the constructor;
- *v2* uses the default constructor;
- as soon as you define a constructor, the default constructor goes away;
- you need to redefine the default constructor:

```
Vector() {};
Vector( double x,double y )
  : x(x),y(y) {};
```

Classes for abstact objects

Objects can model fairly abstract things:

Code:

```
class stream {
private:
  int last_result{0};
public:
  int next() {
    return last_result++; };
};

int main() {
  stream ints;
  cout << "Next: "
       << ints.next() << endl;
  cout << "Next: "
       << ints.next() << endl;
  cout << "Next: "
       << ints.next() << endl;
```

Output
[object] stream:

```
Next: 0
Next: 1
Next: 2
```

Preliminary to the following exercise

A prime number generator has:
an API of just one function: `nextprime`

To support this it needs to store:
an integer `last_prime_found`

Exercise 7

Write a class primegenerator that contains

- members how_many_primes_found and last_number_tested,
- a method nextprime;
- Also write a function isprime that does not need to be in the class.

Your main program should look as follows:

```
cin >> nprimes;
primegenerator sequence;
while (sequence.number_of_primes_found()<nprimes) {
  int number = sequence.nextprime();
  cout << "Number " << number << " is prime" << endl;
}
```

Direct alteration of internals

Return a reference to a private member:

```
class Vector {
private:
  double x,y;
public:
  double &x_component() { return x; };
};
int main() {
  Vector v;
  v.x_component() = 3.1;
}
```

Only define this is really needed.

Reference to internals

Returning a reference saves you on copying.
Prevent unwanted changes by using a 'const reference'.

```
class Grid {
private:
  vector<Point> thepoints;
public:
  const vector<Point> &points() {
    return thepoints; };
};
int main() {
  Grid grid;
  cout << grid.points()[0];
  // grid.points()[0] = whatever ILLEGAL
}
```

‘this’ pointer to the current object

Inside an object, a pointer to the object is available as `this`:

```
class Myclass {
private:
  int myint;
public:
  Myclass(int myint) {
    this->myint = myint;
  };
};
```

‘this’ use

You don’t often need the `this` pointer. Example: you need to call a function inside a method that needs the object as argument)

```
class someclass;
void somefunction(const someclass &c) {
  /* ... */ }
class someclass {
// method:
void somemethod() {
  somefunction(*this);
};
```

(Rare use of dereference star)

More constructors

Copy constructor

- Several default copy constructors are defined
- They copy an object:
  - simple data, including pointers
  - included objects recursively.
- You can redefine them as needed, for instance for deep copy.

```
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v <<
    endl;
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v <<
    endl;
    mine = v; };
  void printme() { cout
      << "I have: " << mine <<
    endl; };
};
```

Copy constructor in action

Code:

```
has_int an_int(5);
has_int other_int(an_int);
an_int.printme();
other_int.printme();
```

Output
[object] copyscalar:

```
set: 5
copy: 5
I have: 5
I have: 5
```

Destructor

- Every class `myclass` has a *destructor* `~myclass` defined by default.
- The default destructor does nothing:

```
~myclass() {};
```

- A destructor is called when the object goes out of scope. Great way to prevent memory leaks: dynamic data can be released in the destructor. Also: closing files.

Destructor example

Just for tracing, constructor and destructor do `cout`:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << endl;
  };
  ~SomeObject() {
    cout << "calling the destructor"
         << endl;
  };
};
```

Destructor example

Destructor called implicitly:

Code:

```
cout << "Before the nested scope"
     << endl;
{
  SomeObject obj;
  cout << "Inside the nested scope"
       << endl;
}
cout << "After the nested scope"
     << endl;
```

Output
[object] destructor:

```
Before the nested scope
calling the constructor
Inside the nested scope
calling the destructor
After the nested scope
```

Headers

C headers plusplus

You know how to use `.h` files in C.

Classes in C++ need some extra syntax.

Class prototypes

Header file:

```
class something {
private:
  int i;
public:
  double dosomething( std::vector<double> v );
};
```

Implementation file:

```
double something::dosomething( std::vector<double> v ) {
  // do something with v
};
```

Data members in proto

Data members, even private ones, need to be in the header file:

```
class something {
private:
  int localvar;
public:
  double somedo(vector);
};
```

Implementation file:

```
double something::somedo(vector v) {
   .... something with v ....
   .... something with localvar ....
};
```

Static class members

A static member acts as if it's shared between all objects.
(Note: C++17 syntax)

Code:

```
class myclass {
private:
  static inline int count=0;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
  myclass obj1,obj2;
  cout << "I have defined "
       << obj1.create_count()
       << " objects" << endl;
```

Output
[link] static17:

```
I have defined 2 objects
```

Static class members, C++11 syntax

```
class myclass {
private:
  static int count;
public:
  myclass() { count++; };
  int create_count() { return count; };
};
  /* ... */
// in main program
int myclass::count=0;
```

Class relations: has-a

Has-a relationship

A class usually contains data members. These can be simple types or other classes. This allows you to make structured code.

```
class Course {
private:
  Person the_instructor;
  int year;
}
class Person {
  string name;
  ....
}
```

This is called the has-a relation.

Literal and figurative has-a

A line segment has a starting point and an end point.

A `Segment` class can store those points:

```
class Segment {
private:
  Point starting_point,
    ending_point;
public:
  Point get_the_end_point() {
    return ending_point; };
}
  ...
  Segment somesegment;
  Point somepoint =
    somesegment.
    get_the_end_point();
```

or store one and derive the other:

```
class Segment {
private:
  Point starting_point;
  float length,angle;
public:
  Point get_the_end_point() {
    /* some computation
       from the
       starting point */ };
}
```

Implementation vs API: implementation can be very different from user interface.

Polymorphism in constructors

You have to decide what to store and what to derive, but you can construct two ways:

```
class Segment {
private:
 // up to you how to implement!
public:
  Segment( Point start,float length,float angle )
    { .... }
  Segment( Point start,Point end ) { ... }
```

Advantage: with a good API you can change your mind about the implementation without changing the calling code.

Exercise 8

- Make a class `Rectangle` (sides parallel to axes) with a constructor:

  ```
  Rectangle(Point bl,float w,float h);
  ```

  The logical implementation is to store these quantities. Implement methods

  ```
  float area(); float rightedge(); float topedge();
  ```

- Add a second constructor

  ```
  Rectangle(Point bl,Point tr);
  ```

  Can you figure out how to use member initializer lists for the constructors?
- Write another version of your class so that it stores two `Point` objects.

Class inheritance: is-a

Examples for base and derived cases

- Base case: employee. Has: salary, employee number.
  Special case: manager. Has in addition: underlings.
- Base case: shape in drawing program. Has: extent, area, drawing routine.
  Special case: square et cetera; has specific drawing routine.

General case, special case

You can have classes where an object of one class is a special case of the other class. You declare that as

```
class General {
protected: // note!
 int g;
public:
 void general_method() {};
};

class Special : public General {
public:
  void special_method() { g = ... };
};

int main() {
  Special special_object;
  special_object.general_method();
  special_object.special_method();
}
```

Inheritance: derived classes

Derived class `Special` *inherits* methods and data from base class `General`:

```
int main() {
  Special special_object;
  special_object.general_method();
```

Members and methods need to be `protected`, not `private`, to be inheritable.

Constructors

When you run the special case constructor, usually the general constructor needs to run too. By default the 'default constructor', but usually explicitly invoked:

```
class General {
public:
  General( double x,double y ) {};
};
class Special : public General {
public:
  Special( double x ) : General(x,x+1) {};
};
```

Access levels

Methods and data can be

- private, because they are only used internally;
- public, because they should be usable from outside a class object, for instance in the main program;
- protected, because they should be usable in derived classes.

Exercise 9

Take your code where a `Rectangle` was defined from one point, width, and height.

Make a class `Square` that inherits from `Rectangle`. It should have the function `area` defined, inherited from `Rectangle`.

First ask yourself: what should the constructor of a `Square` look like?

Overriding methods

- A derived class can inherit a method from the base class.
- A derived class can define a method that the base class does not have.
- A derived class can *override* a base class method:

```
class Base {
public:
  virtual f() { ... };
};
class Deriv : public Base {
public:
  virtual f() override { ... };
};
```

Override and base method

Code:

```
class Base {
protected:
  int i;
public:
  Base(int i) : i(i) {};
  virtual int value() { return i; };
};

class Deriv : public Base {
public:
  Deriv(int i) : Base(i) {};
  virtual int value() override {
    int ivalue = Base::value();
    return ivalue*ivalue;
  };
};
```

Output
[object] virtual:

```
25
```

Operator overloading

```
<returntype> operator<op>( <argument> ) { <definition> }
```

For instance:

Code:

```
Vector operator*(double factor) {
  return Vector(factor*vx,factor*vy);
};
/* ... */
cout << "p1 has length "
     << p1.length() << endl;
Vector scale2r = p1*2.;
cout << "scaled right: "
     << scale2r.length() << endl;
// ILLEGAL Vector scale2l = 2.*p1;
```

Output
[geom] pointmult:

```
p1 has length 2.23607
scaled right: 4.47214
```

Can even redefine equals and parentheses.

Friend classes

A `friend` class can access private data and methods even if there is no inheritance relationship.

```
class A;
class B {
  friend class A;
private:
  int i;
};
class  A {
public:
  void f(B b) { b.i; };
};
```

More

- Multiple inheritance: an X is-a A, but also is-a B.
  This mechanism is somewhat dangerous.
- Virtual base class: you don't actually define a function in the base class, you only say 'any derived class has to define this function'.

Vectors

Vectors are better than arrays

Vectors are fancy arrays. They are easier and safer to use:

- They know what their size is.
- Bound checking.
- Freed when going out of scope: no memory leaks.
- Dynamically resizable.

In C++ you never have to `malloc` again.
(Not even `new`.)

Initialization

Array creation

Multiple ways of creating:

```
{
  vector<int> numbers{5,6,7,8,9,10};
  cout << numbers.at(3) << endl;
}
{
  vector<int> numbers = {5,6,7,8,9,10};
  numbers.at(3) = 21;
  cout << numbers.at(3) << endl;
}
```

(Initializer-lists have more uses than this)

Range over elements

You can write a range-based for loop, which considers the elements as a collection.

```
for ( float e : array )
  // statement about element with value e
for ( auto e : array )
  // same, with type deduced by compiler
```

Code:

```
vector<int> numbers = {1,4,2,6,5};
int tmp_max = numbers[0];
for (auto v : numbers)
  if (v>tmp_max)
    tmp_max = v;
cout << "Max: " << tmp_max
     << " (should be 6)" << endl;
```

Output
[array] dynamicmax:

```
Max: 6 (should be 6)
```

Range over elements by reference

Range-based loop indexing makes a copy of the array element. If you want to alter the array, use a reference:

```
for ( auto &e : my_vector)
  e = ....
```

Code:

```
vector<float> myvector
  = {1.1, 2.2, 3.3};
for ( auto &e : myvector )
  e *= 2;
cout << myvector.at(2) << endl;
```

Output [array] vectorrangeref:

```
6.6
```

(Can also use `const auto& e` to prevent copying, but also prevent altering data.)

Vector definition

Definition, mostly without initialization.

```
#include <vector>
using std::vector;

vector<type> name;
vector<type> name(size);
vector<type> name(size,init_value);
```

where

- `vector` is a keyword,
- `type` (in angle brackets) is any elementary type or class name,
- `name` is up to you, and
- `size` is the (initial size of the array). This is an integer, or more precisely, a `size_t` parameter.
- `init_value` will be used for all elements.

Accessing vector elements

Square bracket notation:

```
vector<double> x(5, 0.1 );
x[1] = 3.14;
cout << x[2];
```

With bound checking:

```
x.at(1) = 3.14;
cout << x.at(2);
```

Safer, slower.

Vectors, the new and improved arrays

- C array/pointer equivalence is silly
- C++ vectors are just as efficient
- ... and way easier to use.

Don't use use explicitly allocated arrays anymore

```
double *array = new double[n]; // please don't
```

Exercise 10

Create a `vector` x of `float` elements, and set them to random values.

Now normalize the vector in L_2 norm and check the correctness of your calculation, that is,

1. Compute the L_2 norm of the vector:

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. Divide each element by that norm;
3. The norm of the scaled vector should now by 1. Check this.

What type of loop are you using?

Vector copy

Vectors can be copied just like other datatypes:

Code:

```
vector<float> v(5,0), vcopy;
v.at(2) = 3.5;
vcopy = v;
vcopy.at(2) *= 2;
cout << v.at(2) << ","
     << vcopy.at(2) << endl;
```

Output
[array] vectorcopy:

```
3.5,7
```

Vector methods

- Get elements with `ar[3]` (zero-based indexing).
- Get elements, including bound checking, with `ar.at(3)`.
- Size: `ar.size()`.
- Other functions: `front`, `back`, `empty`.
- `vector` is a 'templated class'

Dynamic behaviour

Dynamic extension

Extend with `push_back`:

Code:

```
vector<int> array(5,2);
array.push_back(35);
cout << array.size() << endl;
cout << array[array.size()-1] << endl;
```

Output
[array] vectorend:

```
6
35
```

also *pop_back*, *insert*, *erase*.
Flexibility comes with a price.

Dynamic size extending

```
vector<int> iarray;
```

creates a vector of size zero. You can then

```
iarray.push_back(5);
iarray.push_back(32);
iarray.push_back(4);
```

Vector extension

You can push elements into a vector:

```
  vector<int> flex;
/* ... */
  for (int i=0; i<LENGTH; i++)
    flex.push_back(i);
```

If you allocate the vector statically, you can assign with `at`:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat.at(i) = i;
```

Vector extension

With subscript:

```
  vector<int> stat(LENGTH);
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

You can also use `new` to allocate

```
  int *stat = new int[LENGTH];
/* ... */
  for (int i=0; i<LENGTH; i++)
    stat[i] = i;
```

Timing

```
Flexible time: 2.445
Static at time: 1.177
Static assign time: 0.334
Static assign time to new: 0.467
```

Exercise 11

Write code to take a vector of integers, and construct two vectors, one containing all the odd inputs, and one containing all the even inputs. So:

input:
```
5,6,2,4,5
```
output:
```
5,5
6,2,4
```

Can you write a function that accepts a vector and produces two vectors as described?

Vectors and functions

Vector as function return

You can have a vector as return type of a function.

Example: this function creates a vector, with the first element set to the size:

Code:

```
vector<int> make_vector(int n) {
  vector<int> x(n);
  x.at(0) = n;
  return x;
}
  /* ... */
  vector<int> x1 = make_vector(10);
  // "auto" also possible!
  cout << "x1 size: " << x1.size() <<
    endl;
  cout << "zero element check: " << x1.
    at(0) << endl;
```

Output
[array] vectorreturn:

```
x1 size: 10
zero element check: 10
```

Vector as function argument

You can pass a vector to a function:

```
void print0( vector<double> v ) {
  cout << v.at(0) << endl;
};
```

Vectors, like any argument, are passed by value, so the vector is actually copied into the function.

Vector pass by value example

Code:

```
void set0
  ( vector<float> v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << endl;
```

Output
[array] vectorpassnot:

```
3.5
```

Vector pass by reference

If you want to alter the vector, you have to pass by reference:

Code:

```
void set0
  ( vector<float> &v,float x )
{
  v.at(0) = x;
}
  /* ... */
  vector<float> v(1);
  v.at(0) = 3.5;
  set0(v,4.6);
  cout << v.at(0) << endl;
```

Output [array] vectorpassref:

```
4.6
```

Vectors in classes

Can you make a class around a vector?

Vector needs to be created with the object, so you can not have the size in the class definition

```
class witharray {
private:
  vector<int> the_array( ???? );
public:
  witharray( int n ) {
    thearray( ???? n ???? );
  }
}
```

Create and assign

The following mechanism works:

```
class witharray {
private:
  vector<int> the_array;
public:
  witharray( int n )
    : the_array(vector<int>(n)) {
  };
};
```

Better than

```
  witharray( int n ) {
    the_array = vector<int>(n);
  };
```

Multi-dimensional vectors

Multi-dimensional is harder with vectors:

```
vector<float> row(20);
vector<vector<float>> rows(10,row);
```

Create a row vector, then store 10 copies of that:
vector of vectors.

Matrix class

```
class matrix {
private:
  vector<vector<double>> elements;
public:
  matrix(int m,int n) {
    elements =
      vector<vector<double>>(m,vector<double>(n));
  }
  void set(int i,int j,double v) {
    elements.at(i).at(j) = v;
  };
  double get(int i,int j) {
    return elements.at(i).at(j);
  };
};
```

Matrix class'

Better idea:

```
elements = vector<double>(rows*cols);
...
void get(int i,int j) {
  return elements.at(i*cols+j);
}
```

(Old-style solution: use cpp macro)

Exercise 12

Add methods such as `transpose`, `scale` to your matrix class.

Implement matrix-matrix multiplication.

Vectors from C arrays

Use a range constructor to make a vector from a C array:

```
vector<double> x( pointer_to_first, pointer_after_last );
```

Note subtleties:

Code:

```
float *x;
x = (float*)malloc(length*sizeof(
float));
/* ... */
vector<float> xvector(x,x+length);
cout << "xvector has size: " <<
xvector.size() << endl;
xvector.push_back(5);
cout << "Push back was successful"
<< endl;
cout << "pushed element: "
     << xvector.at(length) << endl;
cout << "original array: "
     << x[length] << endl;
```

Output
[array] cvector:

```
xvector has size: 53
Push back was successful
pushed element: 5
original array: 3.40149e+21
```

Span

To be written.

Strings

String declaration

```
#include <string>
using std::string;

// .. and now you can use 'string'
```

(Do not use the C legacy mechanisms.)

String creation

A string variable contains a string of characters.

```
string txt;
```

You can initialize the string variable or assign it dynamically:

```
string txt{"this is text"};
string moretxt("this is also text");
txt = "and now it is another text";
```

Concatenation

Strings can be *concatenated*:

```
txt = txt1+txt2;
txt += txt3;
```

String indexing

You can query the *size*:

```
int txtlen = txt.size();
```

or use subscripts:

```
cout << "The second character is <<"
     << txt[1] << ">>" << endl;
```

More vector methods

Other methods for the vector class apply: `insert`, `empty`, `erase`, `push_back`, et cetera.

Methods only for `string`: `find` and such.

`http://en.cppreference.com/w/cpp/string/basic_string`

I/O

Default unformatted output

Code:

```
for (int i=1; i<200000000; i*=10)
  cout << "Number: " << i << endl;
cout << endl;
```

Output
[io] cunformat:

```
Number: 1
Number: 10
Number: 100
Number: 1000
Number: 10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Reserve space

You can specify the number of positions, and the output is right aligned in that space by default:

Code:

```
cout << "Width is 6:" << endl;
for (int i=1; i<200000000; i*=10)
  cout << "Number: "
       << setw(6) << i << endl;
cout << endl;

cout << "Width is 6:" << endl;
cout << "."
     << setw(6) << 1 << 2 << 3 << endl;
cout << endl;
```

Output [io] width:

```
Width is 6:
Number:      1
Number:     10
Number:    100
Number:   1000
Number:  10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000

Width is 6:
.     123
```

Padding character

Normally, padding is done with spaces, but you can specify other characters:

Code:

```
#include <iomanip>
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.')
         << setw(6) << i
         << endl;
```

Output
[io] formatpad:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Note: single quotes denote characters, double quotes denote strings.

Left alignment

Instead of right alignment you can do left:

Code:

```
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
  /* ... */
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.')
         << setw(6) << i << endl;
```

Output
[io] formatleft:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

Number base

Finally, you can print in different number bases than 10:

Code:

```
#include <iomanip>
using std::setbase;
using std::setfill;
  /* ... */
  cout << setbase(16) << setfill(' ');
  for (int i=0; i<16; i++) {
    for (int j=0; j<16; j++)
      cout << i*16+j << " " ;
    cout << endl;
  }
```

Output
[io] format16:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a
20 21 22 23 24 25 26 27 28 29 2a
30 31 32 33 34 35 36 37 38 39 3a
40 41 42 43 44 45 46 47 48 49 4a
50 51 52 53 54 55 56 57 58 59 5a
60 61 62 63 64 65 66 67 68 69 6a
70 71 72 73 74 75 76 77 78 79 7a
80 81 82 83 84 85 86 87 88 89 8a
90 91 92 93 94 95 96 97 98 99 9a
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa
```

Exercise 13

Make the first line in the above output align better with the other lines:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

Fixed point precision

Fixed precision applies to fractional part:

Code:

```
x = 1.234567;
cout << fixed;
for (int i=0; i<10; i++) {
  cout << setprecision(4) << x << endl;
  x *= 10;
}
```

Output
[io] fix:

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

Exercise 14

Use integer output to print real numbers aligned on the decimal:

```
  1.345
 23.789
456.1234
```

Use four spaces for both the integer and fractional part; test only with numbers that fit this format.

Scientific notation

```
cout << "Combine width and precision:" << endl;
x = 1.234567;
cout << scientific;
for (int i=0; i<10; i++) {
  cout << setw(10) << setprecision(4) << x << endl;
  x *= 10;
}
```

Output

```
Combine width and precision:
1.2346e+00
1.2346e+01
1.2346e+02
1.2346e+03
1.2346e+04
1.2346e+05
1.2346e+06
1.2346e+07
1.2346e+08
1.2346e+09
```

Text output to file

Streams are general: work the same for console out and file out.

```
#include <fstream>
```

Use:

```
#include <fstream>
using std::ofstream;
  /* ... */
  ofstream file_out;
  file_out.open("fio_example.out");
  /* ... */
  file_out << number << endl;
  file_out.close();
```

Redefine less-less

If you want to output a class that you wrote yourself, you have to define how the << operator deals with your class.

```
class container {
  /* ... */
  int value() const {
  /* ... */
  };
  /* ... */
ostream &operator<<(ostream &os,const container &i) {
  os << "Container: " << i.value();
  return os;
};
  /* ... */
  container eye(5);
  cout << eye << endl;
```

C++20 formatting

C++20 will have the `fmtlib` library as part of the standard.

This is closer to `printf` in design, and very much like python's `format` in python3.

Smart pointers

C pointers are barely needed.

- Use `std::string` instead of `char` array; use `std::vector` for other arrays.
- Parameter passing by reference: use actual references.
- Ownership of dynamically created objects: smart pointers.
- Pointer arithmetic: iterators.

Pointers and references

C and F pointers

C++ and Fortran have a clean reference/pointer concept: a reference or pointer is an 'alias' of the original object

C/C++ also has a very basic pointer concept:
a pointer is the address of some object
(including pointers)

If you're writing C++ you should (generally) not use this address mechanism.
if you write C, you'd better understand it.

Reference: change argument

A reference makes the function parameter a synonym of the argument.

```
void f( int &i ) { i += 1; };
int main() {
  int i = 2;
  f(i); // makes it 3
```

Reference: save on copying

```
class BigDude {
public:
  vector<double> array
    (5000000);
}

void f(BigDude d) {
 cout << d.array[0];
};

int main() {
  BigDude big;
  f(big); // whole thing is
    copied
```

Instead write:

```
void f( BigDude &thing ) { ....
     };
```

Prevent changes:

```
void f( const BigDude &thing )
    { .... };
```

Smart pointers

Creating a shared pointer

Allocation and pointer in one:

```
shared_ptr<Obj> X =
    make_shared<Obj>( /* constructor args */ );
  // or:
auto X = make_shared<Obj>( /* args */ );

X->method_or_member;
```

Much better than

```
Obj *X;
*X = Obj( /* args */ );
```

Simple example

Code:

```
class HasX {
private:
  double x;
public:
  HasX( double x) : x(x) {};
  auto &val() { return x; };
};

int main() {
  auto X = make_shared<HasX>(5);
  cout << X->val() << endl;
  X->val() = 6;
  cout << X->val() << endl;
```

Output
[pointer] pointx:

```
5
6
```

Pointers to arrays

The constructor syntax is a little involved for vectors:

```
auto x = make_shared<vector<double>>(vector<double>{1.1,2.2});
```

Getting the underlying pointer

```
X->y;
// is the same as
X.get()->y;
// is the same as
( *X.get() ).y;
```

Code:

```
auto Y = make_shared<HasY>(5);
cout << Y->y << endl;
Y.get()->y = 6;
cout << ( *Y.get() ).y << endl;
```

Output
[pointer] pointy:

```
5
6
```

Pointers don't go with addresses

The oldstyle `&y` address pointer can not be made smart:

```
auto
  p = shared_ptr<HasY>( &y );
p->y = 3;
cout << "Pointer's y: "
     << p->y << endl;
```

gives:

```
address(56325,0x7fff977cc380) malloc: *** error for object
    0x7ffeeb9caf08: pointer being freed was not allocated
```

Automatic memory management

Memory leaks

- Vectors obey scope: deallocated automatically.
- Destructor called when object goes out of scope, including exceptions.
- 'RAII'
- Dynamic allocation doesn't obey scope: objects with smart pointers get de-allocated when no one points at them anymore.
  (Reference counting)

Reference counting illustrated

We need a class with constructor and destructor tracing:

```
class thing {
public:
  thing()  { cout << ".. calling constructor\n"; };
  ~thing() { cout << ".. calling destructor\n"; };
};
```

Pointer overwrite

Let's create a pointer and overwrite it:

Code:

```
cout << "set pointer1"
     << endl;
auto thing_ptr1 =
  make_shared<thing>();
cout << "overwrite pointer"
     << endl;
thing_ptr1 = nullptr;
```

Output
[pointer] ptr1:

```
set pointer1
.. calling constructor
overwrite pointer
.. calling destructor
```

Pointer copy

Code:

```
cout << "set pointer2" << endl;
auto thing_ptr2 =
  make_shared<thing>();
cout << "set pointer3 by copy"
     << endl;
auto thing_ptr3 = thing_ptr2;
cout << "overwrite pointer2"
     << endl;
thing_ptr2 = nullptr;
cout << "overwrite pointer3"
     << endl;
thing_ptr3 = nullptr;
```

Output
[pointer] ptr2:

```
set pointer2
.. calling constructor
set pointer3 by copy
overwrite pointer2
overwrite pointer3
.. calling destructor
```

Linked list code, old style

```
node *node::prepend_or_append(node *other) {
  if (other->value>this->value) {
    this->tail = other;
    return this;
  } else {
    other->tail = this;
    return other;
  }
};
```

Can we do this with shared pointers?

A problem with shared pointers

```
shared_pointer<node> node::prepend_or_append
    ( shared_ptr<node> other ) {
  if (other->value>this->value) {
    this->tail = other;
```

So far so good. However, `this` is a `node*`, not a `shared_ptr<node>`, so

```
    return this;
```

returns the wrong type.

Solution: shared from this

It is possible to have a 'shared pointer to this' if you define your node class with (warning, major magic alert):

```
class node : public enable_shared_from_this<node> {
```

This allows you to write:

```
    return this->shared_from_this();
```

Smart pointer example: linked lists

Linked list structures

Linked list: data structure with easy insertion and deletion of information.

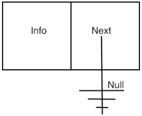
Two basic elements:

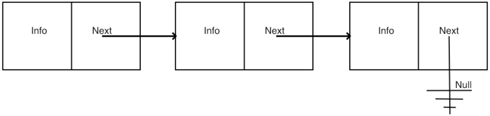
- List, has pointer to first element, or null pointer
- Node, has information, plus pointer to next element (or null)

We are going to look at info routines about a list ('length'), or routines that alter the list ('insert').

(in pictures)

Node data structure and linked list of nodes





Definition of List class

A linked list has as its only member a pointer to a node:

```
class List {
private:
  unique_ptr<Node> head{nullptr};
public:
  List() {};
```

Initially null for empty list.

Definition of Node class

A node has information fields, and a link to another node:

```
class Node {
  friend class List;
private:
  int datavalue{0},datacount{0};
  unique_ptr<Node> next{nullptr};
public:
  friend class List;
  Node() {}
  Node(int value,unique_ptr<Node> tail=nullptr)
    : datavalue(value),datacount(1),next(move(tail)) {};
  ~Node() { cout << "deleting node " << datavalue << endl; };
```

A Null pointer indicates the tail of the list.

Recursive computation of the list length

```
int recursive_length() {
  if (head==nullptr)
    return 0;
  else
    return head->listlength();
};

int listlength_recursive() {
  if (!has_next()) return 1;
  else return 1+next->listlength();
};
```

Iterative computation of the list length

Use a bare pointer, which is appropriate here because it doesn't own the node.

```
int listlength_iterative() {
  int count = 0;
  Node *current_node = head.get();
  while (current_node!=nullptr) {
    current_node = current_node->next.get(); count += 1;
  }
  return count;
};
```

(You will get a compiler error if you try to make `current_node` a smart pointer: you can not copy a unique pointer.)

Exercise 15

Write a function

```
bool List::contains_value(int v);
```

to test whether a value is present in the list.

Try both recursive and iterative.

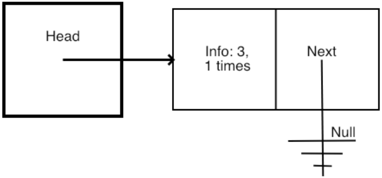
Insert routine design

We will write functions

```
void List::insert(int value);
void Node::insert(int value);
```

that add the value to the list. The `List::insert` value can put a new node in front of the first one; the `Node::insert` assumes the the value is on the current node, or gets inserted after it.

Insert in empty list

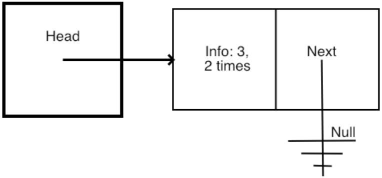


Exercise 16

Next write the case of `Node::insert` that handles the empty list. You also need a method `List::contains` that tests if an item if in the list.

```
mylist.insert(3);
cout << "After one insertion the length is: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << endl;
else
  cout << "Hm. Should contain 3" << endl;
if (mylist.contains_value(4))
  cout << "Hm. Should not contain 4" << endl;
else
  cout << "Indeed: does not contain 4" << endl;
cout << endl;
```

Element is already present

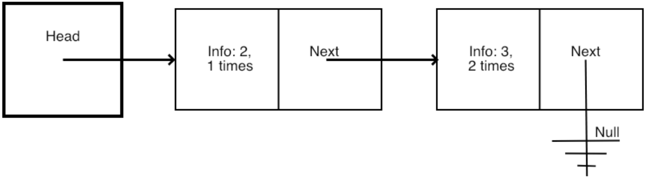


Exercise 17

Inserting a value that is already in the list means that the count value of a node needs to be increased. Update your `insert` method to make this code work:

```
mylist.insert(3);
cout << "Inserting the same item gives length: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(3)) {
  cout << "Indeed: contains 3" << endl;
  auto headnode = mylist.headnode();
  cout << "head node has value " << headnode->value()
       << " and count " << headnode->count() << endl;
} else
  cout << "Hm. Should contain 3" << endl;
cout << endl;
```

Insert element before

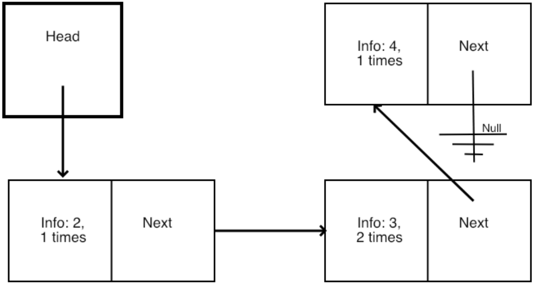


Exercise 18

One of the remaining cases is inserting an element that goes at the head. Update your `insert` method to get this to work:

```
mylist.insert(2);
cout << "Inserting 2 goes at the head; now the length is: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(2))
  cout << "Indeed: contains 2" << endl;
else
  cout << "Hm. Should contain 2" << endl;
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << endl;
else
  cout << "Hm. Should contain 3" << endl;
cout << endl;
```

Insert an element at the tail



Exercise 19

Finally, if an item goes at the end of the list:

```
mylist.insert(4);
cout << "Inserting 4 goes at the tail; now the length is: "
     << mylist.listlength_iterative() << endl;
if (mylist.contains_value(4))
  cout << "Indeed: contains 4" << endl;
else
  cout << "Hm. Should contain 4" << endl;
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << endl;
else
  cout << "Hm. Should contain 3" << endl;
cout << endl;
```

Advanced pointer topics

Opaque pointer

Use `std::any` instead of void pointers.

Code:

```
std::any a = 1;
cout << a.type().name() << ": " << std
    ::any_cast<int>(a) << endl;
a = 3.14;
cout << a.type().name() << ": " << std
    ::any_cast<double>(a) << endl;
a = true;
cout << a.type().name() << ": " << std
    ::any_cast<bool>(a) << endl;

try {
  a = 1;
  cout << std::any_cast<float>(a) <<
    endl;
} catch (const std::bad_any_cast& e) {
  cout << e.what() << endl;
}
```

Output
[pointer] any:

```
i: 1
d: 3.14
b: true
bad any cast
```

Null pointer

C++ has the `nullptr`, which is an object of type `std::nullptr_t`.

```
void f(int);
void f(int*);
  f(NULL);    // calls the int version
  f(nullptr); // calls the ptr version
```

Note: dereferencing is undefined behaviour; does not throw an exception.

Namespaces

You have already seen namespaces

Safest:

```
#include <vector>
int main() {
  std::vector<stuff> foo;
}
```

Drastic:

```
#include <vector>
using namespace std;
int main() {
  vector<stuff> foo;
}
```

Prudent:

```
#include <vector>
using std::vector;
int main() {
  vector<stuff> foo;
}
```

Why not ‘using namespace std’?

This compiles, but should not:

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

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

This gives an error:

```
#include <iostream>
using std::cout;
using std::endl;

def swop(int i,int j) {};

int main() {
  int i=1,j=2;
  swap(i,j);
  cout << i << endl;
  return 0;
}
```

Big namespace no-no

Do not put `using` in a header file that a user may include.

Defining a namespace

You can make your own namespace by writing

```
namespace a_namespace {
  // definitions
  class an_object {
  };
|
```

Namespace usage

```
a_namespace::an_object myobject();
```

or

```
using namespace a_namespace;
an_object myobject();
```

or

```
using a_namespace::an_object;
an_object myobject();
```

Templates

Templated type name

If you have multiple routines that do 'the same' for multiple types, you want the type name to be a variable. Syntax:

```
template <typename yourtypevariable>
// ... stuff with yourtypevariable ...
```

Example: function

Definition:

```
template<typename T>
void function(T var) { cout << var << end; }
```

Usage:

```
int i; function(i);
double x; function(x);
```

and the code will behave as if you had defined `function` twice, once for `int` and once for `double`.

Exercise 20

Machine precision, or 'machine epsilon', is sometimes defined as the smallest number ϵ so that $1+\epsilon>1$ in computer arithmetic.

Write a templated function `epsilon` so that the following code prints out the values of the machine precision for the `float` and `double` type respectively:

Code:

```
float float_eps;
epsilon(float_eps);
cout << "Epsilon float: "
     << setw(10) << setprecision(4)
     << float_eps << endl;

double double_eps;
epsilon(double_eps);
cout << "Epsilon double: "
     << setw(10) << setprecision(4)
     << double_eps << endl;
```

Output
[template] eps:

```
Epsilon float: 1.0000e-07
Epsilon double: 1.0000e-15
```

Templated vector

the Standard Template Library (STL) contains in effect

```
template<typename T>
class vector {
private:
  // data definitions omitted
public:
  T at(int i) { /* return element i */ };
  int size() { /* return size of data */ };
  // much more
}
```

Exceptions

Exception throwing

Throwing an *exception* is one way of signalling an error or unexpected behaviour:

```
void do_something() {
  if ( oops )
    throw(5);
}
```

Catching an exception

It now becomes possible to detect this unexpected behaviour by *catching* the exception:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

Exception classes

```
class MyError {
public :
  int error_no; string error_msg;
  MyError( int i,string msg )
  : error_no(i),error_msg(msg) {};
}

throw( MyError(27,"oops");

try {
  // something
} catch ( MyError &m ) {
  cout << "My error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

You can use exception inheritance!

Multiple catches

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch ( int i ) {
  // handle int exception
} catch ( std::string c ) {
  // handle string exception
}
```

Catch any exception

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

Exceptions in constructors

A function try block will catch exceptions, including in initializer lists of constructors.

```
f::f( int i )
  try : fbase(i) {
    // constructor body
  }
  catch (...) { // handle exception
  }
```

More about exceptions

- Functions can define what exceptions they throw:

```
void func() throw( MyError, std::string );
void funk() throw();
```

- Predefined exceptions: `bad_alloc`, `bad_exception`, etc.
- An exception handler can throw an exception; to rethrow the same exception use '`throw;`' without arguments.
- Exceptions delete all stack data, but not `new` data. Also, destructors are called; section **??**.
- There is an implicit `try/except` block around your main. You can replace the handler for that. See the `exception` header file.
- Keyword `noexcept`:

```
void f() noexcept { ... };
```

- There is no exception thrown when dereferencing a `nullptr`.

Destructors and exceptions

The destructor is called when you throw an exception:

Code:

```
class SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << endl; };
  ~SomeObject() {
    cout <<  "calling the destructor"
         << endl; };
};
  /* ... */
  try {
    SomeObject obj;
    cout << "Inside the nested scope"
    << endl;
    throw(1);
  } catch (...) {
    cout << "Exception caught" << endl;
  }
```

Output
[object]
exceptdestruct:

```
calling the constructor
Inside the nested scope
calling the destructor
Exception caught
```

Use assertions during development

```
#include <cassert>
...
assert( bool expression )
```

Assertions are disabled by

```
#define NDEBUG
```

before the include.

You can pass this as compiler flag:
`icpc -DNDEBUG yourprog.cxx`

Iterators

Auto iterators

```
vector<int> myvector(20);
for ( auto copy_of_int :
    myvector )
  s += copy_of_int;
for ( auto &ref_to_int :
    myvector )
  ref_to_int = s;
```

is actually short for:

```
for ( std::vector<int>
     iterator it=myvector.begin
    () ;
     it!=myvector.end() ; ++it
    )
  s += *it ; // note the deref
```

Range iterators can be used with anything that is iteratable (vector, map, your own classes!)

Other iterator uses

Reverse iteration can not be done with range-based syntax.

Use general syntax with reverse iterator: `rbegin`, `rend`.

Also:

```
auto first = myarray.begin();
first += 2;
auto last  = myarray.end();
last  -= 2-;
myarray.erase(first,last);
```

Simple illustration

Let's make a class, called a bag, that models a set of integers, and we want to enumerate them. For simplicity sake we will make a set of contiguous integers:

```
class bag {
  // basic data
private:
  int first,last;
public:
  bag(int first,int last) : first(first),last(last) {};
```

Use case

We can iterate over our own class:

Code:

```
bag digits(0,9);

bool find3{false};
for ( auto seek : digits )
  find3 = find3 || (seek==3);
cout << "found 3: " << boolalpha
     << find3 << endl;

bool find15{false};
for ( auto seek : digits )
  find15 = find15 || (seek==15);
cout << "found 15: " << boolalpha
     << find15 << endl;
```

Output
[loop] bagfind:

```
found 3: true
found 15: false
```

(for this particular case, use `std::any_of`)

Requirements

- a method `iteratable iteratable::begin()`: initial state
- a method `iteratable iteratable::end()`: final state
- an increment operator `void iteratable::operator++`: advance
- a test `bool iteratable::operator!=(const iteratable&)`
- a dereference operator `iteratable::operator*`: return state

Internal state

When you create an iterator object it will be copy of the object you are iterating over, except that it remembers how far it has searched:

```
private:
  int seek{0};
```

Initial/final state

The begin method gives a bag with the seek parameter initialized:

```
public:
  bag &begin()  {
    seek = first; return *this;
  };
  bag end() {
    seek = last; return *this;
  };
```

These routines are public because they are (implicitly) called by the client code.

Termination test

The termination test method is called on the iterator, comparing it to the `end` object:

```
bool operator!=( const bag &test ) const {
  return seek<=test.last;
};
```

Dereference

Finally, we need the increment method and the dereference. Both access the `seek` member:

```
void operator++() { seek++; };
int operator*() { return seek; };
```

Exercise 21

Make a primes class that can be ranged:

Code:

```
primegenerator allprimes;
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout << endl;
```

Output
[primes] range:

```
2, 3, 5, 7, 11, 13, 17, 19, 23,
```

Auto

Type deduction

In:

```
std::vector< std::shared_ptr< myclass >>*
  myvar = new std::vector< std::shared_ptr< myclass >>
              ( 20, new myclass(1.3) );
```

the compiler can figure it out:

```
auto myvar =
  new std::vector< std::shared_ptr< myclass >>
            ( 20, new myclass(1.3) );
auto result = someobject.somemethod();
```

Type deduction in functions

Return type can be deduced in C++17:

```
auto equal(int i,int j) {
  return i==j;
};
```

Type deduction in functions

Return type can be deduced in C++17:

```
class A {
private: float data;
public:
  A(float i) : data(i) {};
  auto &access() {
    return data; };
  void print() {
    cout << "data: " << data << endl; };
};
```

Auto and references, 1

auto discards references and such:

Code:

```
A my_a(5.7);
auto get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output [auto] plainget:

```
data: 5.7
```

Auto and references, 2

Combine `auto` and references:

Code:

```
A my_a(5.7);
auto &get_data = my_a.access();
get_data += 1;
my_a.print();
```

Output [auto] refget:

```
data: 6.7
```

Auto and references, 3

For good measure:

Code:

```
A my_a(5.7);
const auto &get_data = my_a.
    access();
get_data += 1;
my_a.print();
```

Output [auto] constrefget:

```
make[2]: *** No rule to make target ‘e
```

Lambdas

Lambda expressions

```
[capture] ( inputs ) -> outtype { definition };
```

Example:

```
[] (float x,float y) -> float {
  return x+y; } ( 1.5, 2.3 )
```

Store lambda in a variable:

```
auto summing =
  [] (float x,float y) -> float {
  return x+y; };
cout << summing ( 1.5, 2.3 ) << endl;
```

Capture parameter

Capture value and reduce number of arguments:

```
int exponent=5;
auto powerfunction =
  [exponent] (float x) -> float {
    return pow(x,exponent); };
```

Now powerfunction is a function of one argument, which computes that argument to a fixed power.

Code:

```
cout << "To the power "
     << exponent << endl;
for (float x=1.; x<=9.; x+=1.)
  cout << x << ":" << powerfunction(x)
    << endl;
```

Output [func] lambdait:

```
To the power 5
1:1
2:32
3:243
4:1024
5:3125
6:7776
7:16807
8:32768
9:59049
```

Lambda in object

```
#include <functional>
using std::function;
  /* ... */
class SelectedInts {
private:
  vector<int> bag;
  function< bool(int) > selector;
public:
  SelectedInts( function< bool(int) > f ) {
    selector = f; };
  void add(int i) {
    if (selector(i))
      bag.push_back(i);
  };
  int size() { return bag.size(); };
  std::string string() { std::string s;
    for ( int i : bag )
      s += to_string(i)+" ";
    return s;
  };
};
```

Illustration

Code:

```
cout << "Give a divisor: "; cin >>
  divisor; cout << endl;
cout << ".. using " << divisor <<
  endl;
SelectedInts multiples
  ( [divisor] (int i) -> bool {
      return i%divisor==0; } );
for (int i=1; i<50; i++)
  multiples.add(i);
```

Output
[func] lambdafun:

```
Give a divisor:
.. using 7
Multiples of 7:
7 14 21 28 35 42 49
```

Background Square roots through Newton

Early computers had no hardware for computing a square root. Instead, they used Newton's method. Suppose you have a value y and you want want to compute $x = \sqrt{y}$. This is equivalent to finding the zero of

$$f(x) = x^2 - y$$

where y is fixed. To indicate this dependence on y, we will write $f_y(x)$. Newton's method then finds the zero by evaluating

$$x_{\mathrm{next}} = x - f_y(x)/f_y'(x)$$

until the guess is accurate enough, that is, until $f_y(x) \approx 0$.

Exercise 22

Refer to 217 for background, and note that finding x such that $f(x) = a$ is equivalent to applying Newton to $f(x) - a$.

Implement a class `valuefinder` and its `double find(double)` method.

```
class valuefinder {
private:
  function< double(double) >
      f,fprime;
  double tolerance{.00001};
public:
  valuefinder
  ( function< double(double) > f,
    function< double(double) > fprime )
    : f(f),fprime(fprime) {};
```

used as

```
double root = newton_root.find(number);
```

Casts

C++ casts

Old-style 'take this byte and pretend it is XYZ':
`reinterpret_cast`

Casting with classes:

- `static_cast` cast base to derived without check.
- `dynamic_cast` cast base to derived with check.

Adding/removing const: `const_cast`

Syntactically clearly recognizable.

Const cast

```
int hundredk = 100000;
int overflow;
overflow = hundredk*hundredk;
cout << "overflow: " << overflow << endl;
size_t bignumber = static_cast<size_t>(hundredk)*hundredk;
cout << "bignumber: " << bignumber << endl;
```

Code:

```
long int hundredg = 100000000000;
cout << "long number:      "
     << hundredg << endl;
int overflow;
overflow = static_cast<int>(hundredg);
cout << "assigned to int: "
     << overflow << endl;
```

Output
[cast] intlong:

```
long number:      100000000000
assigned to int: 1215752192
```

Pointer to base class

Class and derived:

```
class Base {
public:
  virtual void print() = 0;
};
class Derived : public Base {
public:
  virtual void print() {
    cout << "Construct derived!"
         << endl; };
};
class Erived : public Base {
public:
  virtual void print() {
    cout << "Construct erived!"
         << endl; };
};
```

Cast to derived class

This is how to do it:

Code:

```
void f( Base *obj ) {
  Derived *der =
    dynamic_cast<Derived*>(obj);
  if (der==nullptr)
    cout << "Could not be cast to
    Derived"
         << endl;
  else
    der->print();
};
  /* ... */
    Base *object = new Derived();
    f(object);
    Base *nobject = new Erived();
    f(nobject);
```

Output
[cast] deriveright:

```
Construct derived!
Could not be cast to Derived
```

Cast to derived class, the wrong way

Do not use this function g:

Code:

```
void g( Base *obj ) {
  Derived *der =
    static_cast<Derived*>(obj);
  der->print();
};
  /* ... */
    Base *object = new Derived();
    g(object);
    Base *nobject = new Erived();
    g(nobject);
```

Output
[cast] derivewrong:

```
Construct derived!
Construct erived!
```

Tuples

C++11 style tuples

```
std::tuple<int,double> id = \
    std::make_tuple<int,double>(3,5.12);
double result = std::get<1>(id);
std::get<0>(id) += 1;
```

Function returning tuple

Return type deduction:

```
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple
      <bool,float>(true,sqrt(x)
    );
};
```

Alternative:

```
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

Catching a returned tuple

The calling code is particularly elegant:

Code:

```
auto [succeed,y] = maybe_root2(x);
if (succeed)
  cout << "Root of " << x
       << " is " << y << endl;
else
  cout << "Sorry, " << x
       << " is negative" << endl;
```

Output
[stl] tuple:

```
Root of 2 is 1.41421
Sorry, -2 is negative
```

Optional results (C++17)

The most elegant solution to 'a number or an error' is to have a single quantity that you can query whether it's valid.

```
optional<float> MaybeRootPtr(float x) {
  if (x<0)
    return {};
  else
    return sqrt(x);
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto root = MaybeRootPtr(x) ; root.has_value() )
      cout << "Root is " << *root << endl;
    else
      cout << "could not take root of " << x << endl;
```

More STL

Iterators outside a loop

First, you can use them by themselves:

Code:

```
vector<int> v{1,3,5,7};
auto pointer = v.begin();
cout << "we start at "
     << *pointer << endl;
pointer++;
cout << "after increment: "
     << *pointer << endl;

pointer = v.end();
cout << "end is not a valid element
: "
     << *pointer << endl;
pointer--;
cout << "last element: "
     << *pointer << endl;
```

Output
[stl] iter:

```
we start at 1
after increment: 3
end is not a valid element: 0
last element: 7
```

(Note: the `auto` actually stands for `vector::iterator`)

Iterators in vector methods

Methods `erase` and `insert` indicate their range with begin/end iterators

Code:

```
vector<int> v{1,3,5,7,9};
cout << "Vector: ";
for ( auto e : v ) cout << e << " ";
cout << endl;
auto first = v.begin();
first++;
auto last = v.end();
last--;
v.erase(first,last);
cout << "Erased: ";
for ( auto e : v ) cout << e << " ";
cout << endl;
```

Output
[stl] erase:

```
Vector: 1 3 5 7 9
Erased: 1 9
```

Note: end is exclusive.

Reduction operation

Default is sum reduction:

Code:

```
vector<int> v{1,3,5,7};
auto first = v.begin();
auto last  = v.end();
auto sum = accumulate(first,last,0);
cout << "sum: " << sum << endl;
```

Output
[stl] accumulate:

```
sum: 16
```

Reduction with supplied operator

Supply multiply operator:

Code:

```
vector<int> v{1,3,5,7};
auto first = v.begin();
auto last  = v.end();
first++; last--;
auto product =
  accumulate
    (first,last,2,multiplies<>());
cout << "product: " << product << endl;
```

Output
[stl] product:

```
product: 30
```

Templated functions for limits

Use header file `limits`:

```
#include <limits>
using std::numeric_limits;

cout << numeric_limits<long>::max();
```

Random number example

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```