Part 8: C++, the better C

Contents

[DOCUMENT NOT FINALIZED YET]

Sources: Wikipedia, `http://www.cplusplus.com`

C++ Overview

Developed by Bjarne Stroustrup starting in 1979 at Bell Labs, C++ was originally named "C with Classes"

It added object oriented (OO) features, such as classes, and other enhancements to the C programming language

The language was renamed C++ in 1983, as a pun involving the increment operator

First standardized in 1998, and then subsequently in 2003, 2007, and recently in 2011

C++11 adds a multitude of features — such as lambda expressions, move semantics, and uniform initialization (see `http://en.wikipedia.org/wiki/C++11`)

Most of them were supported by g++ 4.7, and all by g++ 5.4 (which is installed on the lab machines)

In this section we give a brief overview of C++'s main features without going into much detail

C++ Design Philosophy

In "The Design and Evolution of C++" (1994), Bjarne Stroustrup describes some rules that he used for the design of C++:

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object oriented programming, and generic programming)
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is designed to be compatible with C as much as possible, therefore providing a smooth transition from C
- C++ avoids features that are platform specific or not general purpose
- C++ does not incur overhead for features that are not used (the "zero-overhead principle")
- C++ is designed to function without a sophisticated programming environment

Object Oriented Programming

C++ supports the following object oriented programming paradigms:

- Data encapsulation
- Inheritance
- Polymorphism

which can improve the design, structure, and re-usability of code. In C++, objects are values of a class type. As we have seen earlier, C++ classes consist of data members and member functions that act on the data, including constructors and a destructor which are automatically called when class objects are created or destroyed, respectively

Data encapsulation is the concept of bundling data with methods operating on that data and restricting access to some of the object's components. This prevents unauthorized parties from directly accessing data, and allows us to change implementation details without requiring application code adjustments

Data Encapsulation Example

```
class Rectangle
{
public:
  int x, y, width, height;
};

Rectangle r;
printf("%d", r.width); // works, but discouraged
r.height = 5; // direct access could corrupt object state

class Rectangle2
{
private:
  // hide implementation details
  int x, y, width, height;
public:
  // public interface
  void set_x(int v) { x = v; }
  int get_x() const { return x; }
  ...
};

Rectangle2 r2;
r2.set_y(5); // preferred
int w = r2.get_width();
// Implementation of get_width can be changed
// without users having to adjust their code!
// E.g. int get_width() const { return abs(x1-x0); }
```

Inheritance

In object oriented programming, inheritance is a way to establish an "is-a" relationship between objects

It is often confused as a way to reuse existing code which is not a good practice because inheritance leads to tight code coupling

Rather, re-usability of code is achieved through composition, in which new objects contain sub-objects whose member functions are re-used

In classical inheritance which defines objects by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, super-classes, or parent classes

The resulting classes are known as derived classes, sub-classes, or child classes

Derived classes are more specialized than base classes. Often, data and function members are added. A typical example is a collection of shape types in a graphics library (Square, Triangle, Circle, etc.) that all derive from a common base class Shape

Inheritance Example

```
struct Shape
{
private:
  Color color;
  Point location;

public:
  void draw() { printf("undefined"); exit(10); }
  void set_color(Color c) { color = c; }
};

// Circle is a sub-class of Shape.
// Circle inherits all Shape members,
// adds radius, and overrides Shape's draw function.
// Shape is the base class or super-class of Circle
// Meaning: "Circle is-a Shape"
struct Circle : public Shape
{
private:
  int radius;

public:
  void draw() {  // draw circle ...
  }
};

Circle c; // has color, location, radius, draw, set_color
c.set_color(0); // calls Shape::set_color
c.draw();       // calls Circle::draw
```

In C++, class member functions can be declared `virtual`, which makes them polymorphic

This means that values of different data types can be handled using a uniform interface. Example:

```
class B                 // base class
{
  virtual void foo() { printf("B"); }
};

class X : public B   // X "is-a" B
{
  void foo() override { printf("X"); }
};

// X::foo() is overriding B::foo()
// annotation "override" is optional - it reminds the reader
// that we are overriding a function and instructs
// the compiler to check that we actually do

class Y : public B   // Y "is-a" B
{
  void foo() override { printf("Y"); }
};

X *px = new X; px->foo();     // prints X
Y *py = new Y; py->foo();     // prints Y
B *pb = px;                   // works, because X is a B
pb->foo();                    // prints X
pb = py;                      // works, because Y is a B
pb->foo();                    // prints Y
// polymorphism at work: same invocation syntax, different
// results. The runtime system knows the type of *pb ...
```

Polymorphism is very useful. In a graphics library, for instance, one could store pointers to all shape objects in an array and call the virtual draw function on each object to draw all shapes like so:

```
Shape **objects = new Shape*[N];
objects[0] = new Circle;
objects[1] = new Rectangle;
...
for (int i=0; i < N; ++i) {
  objects[i]->draw();
}
```

Each sub-class of Shape implements its own draw function, and C++'s polymorphism mechanism ensures that the right functions are called. I.e., Circle::draw for circles and Rectangle::draw for rectangles, etc. Elegant, indeed!

The only requirement for this code (which may be stored in a pre-compiled library) to work with new user-defined shape types is that they are derived from Shape

Unlike Java, polymorphism is optional in C++, because it incurs runtime cost (a pointer is added to each object, and virtual function dispatch takes longer)

Operator Overloading

C++ allows programmers to use operators in new contexts:

```
Matrix A, B, C;
C = A + B * B;   // matrix operations!

Complex a, b, c;
c = a + b * b;   // complex number arithmetic!

Rational u, v, w;
w = u + v * v;   // rational arithmetic!

ofstream os;     // output stream
ifstream is;     // input stream
os << "foo";    // write to output stream
is >> x;         // read from input stream
```

This is a useful feature. For instance, one could take a numerical algorithms library that works with double floating point numbers, replace all instances of `double` by `Number`, and then use `typedef Rational Number;` to let the library work with exact rational numbers instead of floating point values that are plagued by rounding errors. THIS IS COOL!

All operators can be redefined in the context of new types

This includes assignment operators and binary relations such as == and <

Example:

```
struct Point {
  int x, y;
  void operator=(const Point &u) // assignment
  {
    x = u.x; y = u.y; // copy all components
  }
};

// returns true iff point a is lexicographically
// smaller than point b
bool operator<(const Point &a, const Point &b)
{
  if (a.x < b.x) { return true; }
  return a.x == b.x && a.y < b.y;
}

Point a, b;
a = b;       // invokes assignment operator
if (a < b)  // invokes less-than operator
```

Generic Programming (1): Function Templates

Consider the following code:

```
int max(int x, int y) {
  return x > y ? x : y;
}

char max(char x, char y) {
  return x > y ? x : y;
}

double max(double x, double y) {
  return x > y ? x : y;
}
```

Do we really have to go on and provide identical implementations for all types we want to compute the maximum value for? Looks silly. The compiler should do the work for us. It does — when using function templates:

```
template <typename T>
T max(T x, T y) {
  return x > y ? x : y;
}
```

T is a type parameter, which when calling `max` is bound to the actual type we are using

After deducing type `T` by matching the actual function argument type, the compiler compiles function template `max` for that specific type and emits code that calls the specific function

Example:

```
float x, y;
float m = max(x, y);  // compiler generates call
                      // to float max(float x, float y);

Point p, q;
Point r = max(p, q);  // compiler generates call
                      // to Point max(Point x, Point y);

typedef const char *ccptr;
ccptr s = "foo", t = "bar";
ccptr u = max(s, t);  // compiler generates call
                      // to ccptr max(ccptr x, ccptr y);
```

The only requirement on type `T` is that it supports >

What if it doesn't? In this case you'll see a compiler error message

In such a scenario, one could either implement the > operator for the type in question, or provide an explicit function definition

If such an explicit definition exists, the compiler will choose it over any matching function template

```
template <typename T>
T max(T x, T y) {
  return x > y ? x : y;
}

// assuming type Foo doesn't support >
Foo max(Foo x, Foo y)
{
  return is_greater_than(x, y) ? x : y;
}

Foo x, y;
x = max(x, y); // calls Foo max function, rather
               // than function template max
```

Exercise: use this function preference rule to fix max for C-strings on the previous page, which doesn't work as intended

Generic Programming (2): Class Templates

The idea of parameterizing types is also very useful for classes

For instance, consider container types such as vectors, lists, stacks, and queues, whose implementation doesn't depend on the type of elements we store

In C, you would have to either duplicate code (list of integers, list of Points, list of Foos, etc.) or create a type-unsafe generic version using `void*` pointers, similar to `qsort` we have seen in the labs

C++'s class templates are much more elegant and type-safe

The way you create class templates for container classes is straight-forward:

- Start with an implementation that stores ints, say
- Then, whenever you refer to the element type, replace `int` by type variable `T`, say
- Finally, add `template <typename T>` in front of your class definition

Example: Queue Class Template

```
template <typename T>
class Queue
{
public:
  // initializes empty queue with maximal c elements
  Queue(int c) {
    capacity = c;
    data = new T[capacity];
    reset();
  }

  ~Queue() { delete [] data; }

  // empties queue
  void reset() { head = tail = n = 0; }

  // return true iff queue is empty
  bool empty() const { return n == 0; }

  // return true iff queue is full
  bool full() const { return n >= capacity; }
```

```
  // add element to queue (at tail)
  // pre-condition: not full
  void add(const T &x) { // const& for safety and speed
    assert(!full());
    data[tail++] = x;
    tail %= capacity; // wrap around
    ++n;
  }

  // remove and return head element
  // pre-condition: not empty
  T remove() {
    assert(!empty());
    const T &x = data[head++]; // a bit faster (no copy)
    head %= capacity; // wrap around
    --n;
    return x;
  }

private:
  int capacity;   // maximum number of elements
  T *data;        // pointer to element array
  int head, tail; // current remove/add locations
  int n;          // actual number of elements stored
};
```

Application

How can we put our new shiny Queue class template to work?

We need to tell the compiler what type of Queue we want by explicitly instantiating the class template with the element type enclosed in < > :

```
Queue<int> iq(100);          // int queue
iq.add(1); iqe.add(2);

Queue<double> dq(50);        // double queue
dq.add(2.0); dq.add(3.5);

Queue<Point*> dp(20);        // Point pointer queue
dp.add(new Point); dp.add(new Point);
```

For each different instantiation, the compiler creates a new specific queue class. For this to work, all template code must reside in header files. Compiling classes multiple times slows down the compilation process. However, the fact that class templates only need to be written once and then can be applied to arbitrary types outweighs this disadvantage

Exercise: Transform the `Vector` and `SList` classes we have seen in Part 4 into class templates, so that we can use them like so:

```
Vector<int> iv(10);      // int vector
iv.set(1, 99);
int i = iv.get(0);

Vector<double> dv(10);   // double vector
dv.set(1, 3.5);
double d = dv.get(0);

SList<int*> pl;          // list of int pointers
pl.add_head(new int);
```

There is much more to say about function templates, class templates, and other C++ features we haven't covered in this introduction

To learn more, fire up your browser and search for C++ tutorials (or take CMPUT 350)

In the remaining course time we will look at the Standard Template Library (STL) which offers numerous useful container types and algorithms

C++ Libraries

Most programming languages have an associated core library, which is sometimes called "standard library", especially if it is included as part of the published language standard

Such libraries are conventionally made available by all implementations of the language

Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output

The availability of high-quality implementations of standard data structures and algorithms is important, because it frees programmers from the tedious and error prone process of re-inventing the wheel. The time thus saved can be put into solving the problems at hand

Throughout this course, we have encountered a few C library functions (`fopen, fclose, fputc, sqrt, qsort, ...`), a C++ library function (`std::swap`), and `pthread` functions

Compared to Java, though, the C++ standard library is still small, but its growth is accelerating

In 1994 Alexander Stepanov and Meng Lee introduced the Standard Template Library (STL) for C++, which to this day is the core of the C++ standard library

The STL is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed

In addition, the Boost C++ library collection has become rather popular in recent years (`www.boost.org`). It can be considered a C++ library development playground, which provides free peer-reviewed portable C++ source libraries, which after intensive testing and improvements often make it into the C++ standard

In the remaining time we can only give a brief STL overview

Standard Template Library Overview

The STL contains the following elements:

Container class templates

- Sequence containers
  - elements have predefined locations
  - `vector, forward_list, list, deque, string, ...`
- Associative containers
  - element location depends on element key
  - `set, map, unordered_set, unordered_map...`

Algorithms

- Container independent (majority)
- `sort, find, merge, random_shuffle, ...`

Iterators

- pointer-like types used for traversing STL containers
- interface between algorithms and containers

Examples

```
#include <vector>
#include <algorithm>
using namespace std; // can refer to standard library
        // elements without having to use std:: prefix,
        // like std::vector or std::generate
int main()
{
  // CONTAINERS + ALGORITHMS
  vector<int> v(10);   // vector of 10 ints, fill with...
  generate(begin(v), end(v), rand); //... random values
  v[0] = 6; // access like array

  // loop through vector using ITERATORS
  vector<int>::iterator it = begin(v), end = end(v);
  int sum = 0;
  for (; it != end; ++it) { sum += *it; }

  // equivalent C++11 code : ``range-based for''
  for (int x : v) { sum += x; }

  // ALGORITHM: shuffle elements randomly
  random_shuffle(begin(v), end(v));

  // MEMBER FUNCTIONS: if non-empty, erase first element
  if (!v.empty()) {
    v.erase(begin(v));
  }
}
```

STL Overview (Continued)

STL is part of the C++ standard library

Several implementations exist

g++'s is based on SGI's

Good Web Sites:

- `http://www.cplusplus.com/reference/stl/`
- `www.sgi.com/tech/stl`

Good Books

- Meyers: "Effective STL"
- Josuttis: "The C++ Standard Library"

Sequence Containers

Positions of elements in container are fixed

`vector<T>`

- Vector class template, dynamic array functionality
- Element type is `T`
- `#include <vector>`

`list<T>`

- Doubly linked list class template
- Data associated with node is `T`
- `#include <list>`

`deque<T>` ("deck");

- double-ended queue; supports random access: `d[i]`
- inserting/deleting at both ends takes amortized constant time
- inserting/deleting in the middle: linear time
- `#include <deque>`

Sequence Containers (Continued)

basic_string<T>

- sequence of characters
- `std::string = basic_string<char>`
- similar to vector
- many member functions: insert, append, erase, find, replace...
- C-string replacement
- `#include <string>`

Examples:

```
std::string s = "test";
std::string t = s + " case";  // concatenation works!
const char *cs = s.c_str();   // conversion to C-string
```

Important std::vector Member Functions

```
iterator begin()    : returns iterator to first element
iterator end()      : returns iterator to passed-the-end element
                      (global functions begin(v), end(v) preferred)
size_type size()    : number of elements in vector
bool empty() const  : true iff vector is empty

void push_back(const T&) : inserts new element at the end
                           (amortized constant time)
void pop_back()     : removes last element

reference operator[](size_type i): returns element i (starts with 0)
reference back()    : returns reference to last element, assumes !empty

void clear()               : remove all elements
void erase(iterator pos)   : removes element at position pos
void reserve(size_type n)  : allocates memory for n elements if needed
bool operator==(const vector&, const vector&) : equality
```

Iterators

Generalization of pointers

Often used to iterate over ranges of objects

- iterator points to object
- the incremented iterator points to the next object

Central to generic programming

- interface between containers and algorithms
- algorithms take iterators as arguments
- container only needs to provide a way to access its elements using iterators
- allows us to write generic algorithms operating on different containers such as vector and list using the same code

Each container type has its own specific iterator type which can be used like so:

```
vector<int> v;
// step through v
vector<int>::iterator it = begin(v), end = end(v);
// iterator is a type defined in class template vector
// see /usr/include/c++/5.4.0/bits/stl_vector.h
for (; it != end; ++it) {
  printf("%d ", *it); // print element iterator points to
}

list<int> l;
// step through l
list<int>::iterator it = begin(l), end = end(l);
for (; it != end; ++it) {
  printf("%d ", *it); // print element iterator points to
}
```

Polymorphism at work: The code for stepping through both containers is identical, but `++it` does different things:

For vectors, the overloaded `++` operator simply increments an internal pointer wrapped by the iterator object (vector elements are laid out consecutively in memory, like arrays). For lists, `++it` is equivalent to following the node successor pointer

std::vector Examples

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

int main()
{
  const int N = 1000;
  vector<int> v;        // empty integer vector

  v.reserve(N);         // reserve memory for N elements
  // saves time and memory because v doesn't need to grow;
  // v.size() still 0

  // append N elements
  for (int i=0; i < N; ++i) { v.push_back(i); }

  // add up all elements, array syntax
  int s = 0;
  for (size_t i=0; i < v.size(); ++i) { sum += v[i]; }

  // alternative: use iterator to step through vector;
  // a bit faster because above v.size() is called multiple times
  s = 0;
  vector<int>::iterator it = begin(v), end = end(v);
  for (; it != end; ++it) { s += *it; }

  // remove all elements one by one back to front
  while (!v.empty()) { v.pop_back(); }
  assert(v.empty());

  // when leaving main v is destroyed here

  // However, if v contains pointers,
  // destructors are *NOT* called on the objects pointers point to!
}
```

Important std::list Member Functions

```
iterator begin(): returns iterator to first element
iterator end()  : returns iterator to end (last element+1)
             (global functions begin(v), end(v) preferred)

size_type size()  : number of list elements (linear time!)
bool empty() const: true iff list is empty

void push_front(const T&): inserts new element at front
void push_back(const T&) : inserts new element at end
void pop_front() : removes first element
void pop_back()  : removes last element

iterator insert(iterator pos, const T&) :
                             inserts element in front of pos
void erase(iterator pos) : removes element at position pos

void reverse()           : reverses list (linear time!)
void splice(iterator pos, list<T> &x) :
                       inserts x in front of pos, clears x
```

std::list Examples

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

int main()
{
  list<int> list;

  list.push_back(0);
  list.push_front(1);
  list.insert(begin(list), 2);  // = list.push_front(2)
  // list now 2 1 0

  list<int> x(3, 10);           // x is list of 3 tens
  // insert x after 2
  list.splice(begin(list)+1, x);  // x now empty

  // print list
  // note: code identical to vector code! polymorphism ...
  list<int>::iterator it = begin(list), end = list.end();
  for (; it != end; ++it) {
    printf("%d ", *it);
  }
  // output: 2 10 10 10 1 0

  // list destructor called here
}
```

Associative Containers

Support efficient retrieval of elements based on keys

Support insertion/removal of elements

Difference to sequence containers: no mechanism for inserting elements at specific locations. Object location depends on key

Here we briefly discuss

- `std::set<T>` : set of elements of type `T`
- `std::map<U,V>` : maps elements of type `U` to elements of type `V`

STL's set and map implementations are based on balanced binary search trees, leading to worst-case runtime $\Theta(\log n)$ for insert, delete, and find operations, where n is the number of stored elements

std::set Example

Sets contain unique elements

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

// set of integers
set<int> s;

// populate s
s.insert(0); s.insert(2); s.insert(1); s.insert(0);
// note: inserting 0 the second time
// doesn't change the set

// visit all elements stored in set in ascending order
// again, same syntax!
set<int>::iterator it = begin(s), end = end(s);
for (; it != end; ++it) {
  printf("%d ", *it);
}

// this also works with all STL int containers:
for (int x : s) { printf("%d ", x); }

if (s.find(0) != s.end()) {
  printf("found element");
}

// output: 0 1 2 found element
```

std::map Example

Maps contain pairs (u, v), with unique first components. (u, v) in map means "u is mapped to v"

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

typedef map<int,double> i2d;
i2d m;  // maps integers to doubles

// populate m using array index notation
m[0] = 1;  // insert pair (0, 1); same as m.insert({0, 1});
m[1] = 2;  // insert pair (1, 2)
m[2] = 9;  // insert pair (2, 9)

// visit all pairs stored in map
i2d::iterator it = begin(m), end = end(m);
for (; it != end; ++it) {
  printf("%d mapped to %f\n", it->first, it->second);
}

// find mapping for 2
it = m.find(2);
if (it != end(m)) {
  // != end(m) => found
  // *it is a pair (key, data)
  printf("found (%d,%f)\n", it->first, it->second);
}

output:
0 mapped to 1.000000
1 mapped to 2.000000
2 mapped to 9.000000
found (2,9.000000)
```

What else is there in STL?

Algorithms

- for setting, copying, swapping, shuffling, finding, and sorting container elements ...
- use iterators to access container elements

Hashed associative containers (`unorderd_set/map`)

- organized as hash tables
- faster than standard tree-based containers — $O(1)$ rather than $\Theta(\log n)$
- but need more space

See `http://www.cplusplus.com/reference/stl` for more information.

STL Algorithm Examples

```
#include <algorithm>
#include <functional>
#include <vector>

std::vector<int> v;

// ascending order; faster than C-lib's quicksort
std::sort(begin(v), end(v));

// descending (need to pass comparison object)
std::sort(begin(v), end(v), greater<int>());

int A[N];
std::sort(A, A+N); // also works! pointers are
                   // iterators ...

// sets all elements
std::fill(A, A+N, 314159);

// reverses sequence container
std::reverse(begin(v), end(v));

// shuffles sequence container
random_shuffle(A, A+N);
```

Wrapping Up ...

C and C++ are powerful programming languages that give programmers low-level control over memory and hardware devices

C++ adds high-level features such as object oriented and generic programming support, which allows us to scale up project sizes considerably

The upside of this flexibility is fast program execution and transparent resource allocation

The potential downside is undefined program behaviour which lurks in many corners if programmers are not diligent, but tools like debuggers and `valgrind` can help identify such problems quickly

My advice to aspiring programmers is to never stop learning about programming languages, efficient algorithms and data structures — and to work on as many programming side-projects as you can. With practice comes success. Good luck! M.B.

The End