EE2-12 Software Engineering 2: Object-Oriented Programming

Week 7 - Generic Programming: Templates, and the Standard Template Library (STL)

Sahbi Ben Ismail

Imperial College London, Department of Electrical and Electronic Engineering

Imperial College London

Autumn Term 2018-19

Module Syllabus

- Week 1 Classes and Objects I: Introduction
- Week 2 Classes and Objects II: Constructors, and Operator Overloading
- Week 3 More on Classes, Objects, and Operator Overloading
- Week 4 Objects and Dynamic Memory
- Week 5 Classes Relationships: Association, Aggregation/Composition and Generalisation (Inheritance)
- Week 6 Polymorphism and Virtual Functions
- Week 7 **Generic Programming: Templates, and the Standard Template Library (STL)**
- Week 8 Exceptions Handling
- Week 9 C++ to Java
- Week 10 Revision

Week 7 Intended Learning Outcomes (ILOs)

By the end of this week you should be better able to:

Lecture

1. Design and write code with **generic behaviour** using **function templates**
2. Design and write code with **generic properties/behaviour** using **class templates**
3. Understand and use some elements of the **Standard Template Library (STL)**: **containers**, **iterators** and **algorithms**

Lab

1. Apply **function and class templates** in generic code
2. Use **iterators** when looping on a container
3. Apply some **STL algorithms** such as sorting and searching to **STL containers** (`vector`, `list`)

Problematic: Generic programming, yet other mode of ‘polymorphism’

```cpp
#include <iostream>   //std::cout
#include <vector>     //std::vector
#include <algorithm> //std::sort

int main() {
  std::vector<int> vi;
  std::vector<Point> vp;

  std::vector<Warrior*> good_warriors;
  std::vector<Warrior*> bad_warriors;
  //same class vector<> with different types

  std::cout << vp.capacity() << std::endl;
  for(int i=0; i<vp.size(); i++) {
    std::cout << vp[i] << std::endl;
  }
  //...

  std::sort(vp.begins(), vp.end()); //sorts the vector using <
  std::sort(good_warriors.begins(), good_warriors.end());
  //same function sort() with different types

  return 0;
}
```

Outline

Outline

Bjarne Stroustrup - GoingNative 2013 - The Essence of C++

YouTube video https://www.youtube.com/watch?v=D5MEsboj9Fc

Week 4: Objects and Dynamic Memory (start at 12'00", until 15'20")

C++ in four slides:

- Map to hardware
- Classes
- Inheritance
- **Parameterised types** (at 16'09")

Week 7: GP: Templates, and the STL (start at 47'53", until 47'44")

https://youtu.be/D5MEsboj9Fc?t=2873

- *Extremely general/flexible*
- *Generic Programming: Templates*
- *Generic Programming: Algorithms*
- *Generic Programming is just Programming*

Week 6 Polymorphism and Virtual Functions I

- OOP “Big Four”: abstraction + encapsulation + **inheritance** + **polymorphism**

Polymorphism

syntax, etymology: poly+morphism = many forms/shapes

semantic: same function name, different behaviour(s)

Week 6 Polymorphism and Virtual Functions II

Polymorphism and Virtual Functions

- Binding: connecting a function call to a function body
- **Early (static)** binding: performed (by compiler and linker) before the program is run.
- **Late (dynamic)** binding: (partially) occurring at runtime, based on the actual type of the object.
- Keyword `virtual` instructs the compiler to perform **late binding** on that function.

Week 6 Polymorphism and Virtual Functions III

Examples of polymorphic functions

- `say_hello()` called on pointers to `Students`
- `attack()` called on pointers to Warriors (`Ninjas`, `Samurais`)
- `draw()` called on pointers to Shapes (`Triangles`, `Rectangles`, etc...)

Week 6 Lab - play a mini combat game I

- project files reorganisation in different folders (`src/` `lib/` `inc/` `bin/`)
- `Makefile`: more variables and compiler options (flags)
- `README.md`: documentation

```
# Project files organisation
The project files are split into:
- "README.md": documentation
- "Makefile"
- "inc/*.hpp": header files (declaration)
- "src/*.cpp": source files (implementation/definition) and the main program
- "lib/": external libraries (NOT RELEVANT HERE BECAUSE WE DO NOT USE ANY EXTERNAL
- "bin/*": object files (*.o) and the executable program

# Building the project
compilation and linking
 $make all

running
 $make run

cleaning
 $make clean
```

Building the application

```
# Note: compiling, linking and running without the Makefile:
1) compiling
g++ -I inc/ -c src/warrior.cpp -o bin/warrior.o
g++ -I inc/ -c src/ninja.cpp -o bin/ninja.o
g++ -I inc/ -c src/samurai.cpp -o bin/samurai.o
g++ -I inc/ -c src/game.cpp -o bin/game.o
g++ -I inc/ -c src/main_prog.cpp -o bin/main_prog.o

(note that
g++ -I inc/ -c src/warrior.cpp
generates warrior.o in the local directory, not in bin/ as we wish. That is why we added
 -o bin/warrior.o
 The -o flag here is not for linking, but for compiling)
)

2) linking
g++ bin/*.o -o bin/main_prog

3) running
./bin/main_prog
```

Makefile I

```
# -------- Variables --------
TARGET=bin/main_prog

OBJECTS=bin/warrior.o  bin/ninja.o  bin/samurai.o bin/game.o $(TARGET).o

# -------- Compiler --------
CC=g++
CCFLAGS=-Wall

# -------- Compiling options --------
INCFLAGS=-I ./inc
#LIBFLAGS=-L ./lib

# -------- Compiling --------
bin/%.o : src/%.cpp
	$(CC) $(CCFLAGS) $(INCFLAGS) -c $< -o $@

# -------- Linking --------
all: $(TARGET)

$(TARGET): $(OBJECTS)
	$(CC) $(OBJECTS) -o $(TARGET)

#note: if using an external library, placed in folder lib/, then use the variable LIBFLA
#$(CC) $(LIBFLAGS) $(OBJECTS) -o bin/$(TARGET)
```

Makefile II

```
# -------- Running --------
run:
	./$(TARGET)

# -------- Cleaning --------
clean:
	rm bin/*.o $(TARGET)


##########################
# Documentation/Reminders #
##########################
# file name: either makefile or Makefile
# (not Make, nor Makefile.txt, nor Makefile.mk nor Makefile.mak etc...)
#
#
#Format of a rule:
#
#target: dependency(ies)
#[TAB]  command(s)
#
#(the second line has to start with a TABULATION)
#
#Warning about TAB:
#when copying/pasting from a PDF file e.g, re-check ALL the tabulations
```

Makefile III

```
#
#
#User-defined variables:
#definition: VAR_NAME=value
#use: $(VAR_NAME)
#
#
#Built-in variables:
#'$@' the target name
#'$<' the first dependency
#'$^' list of all the dependencies (including $<)
```

The game (basic attacks)

```cpp
#include "warrior.hpp"
#include "ninja.hpp"
#include "samurai.hpp"
#include "game.hpp"

int main() {
  //good worriors: team of 2 ninjas and 2 samurais
  Warrior* gw1 = new Ninja("Bruce Lee");
  Warrior* gw2 = new Ninja("Jackie Chan");
  Warrior* gw3 = new Samurai("Jet Li");
  Warrior* gw4 = new Samurai("Chuck Norris");

  //bad worriors: team of 2 ninjas and 2 samurais
  Warrior* bw1 = new Ninja("Evil Ninja");
  Warrior* bw2 = new Ninja("Ninja Assassin");
  Warrior* bw3 = new Samurai("Evil Mind");
  Warrior* bw4 = new Samurai("Evil Sword");

  //game
  Game g;
  g.add_good_warrior(gw1);
  g.add_good_warrior(gw2);
  g.add_good_warrior(gw3);
  g.add_good_warrior(gw4);

  g.add_bad_warrior(bw1);
  g.add_bad_warrior(bw2);
  g.add_bad_warrior(bw3);
  g.add_bad_warrior(bw4);

  g.run();

  return 0;
}
```

```
/* Output:
  $ make
  g++ -Wall -I ./inc -c src/game.cpp -o bi
  g++ -Wall -I ./inc -c src/main_prog.cpp
  g++ bin/warrior.o  bin/ninja.o
bin/samurai.o bin/game.o bin/main_prog.o -

  $ make run
  ./bin/main_prog
  good_warriors.size()=4
  bad_warriors.size()=4

  Battle start!

  ############ Round 1 ############
  Bruce Lee (Ninja) attacks.
  Ninja Assassin (Ninja) attacks.

  ############ Round 2 ############
  Jackie Chan (Ninja) attacks.
  Evil Sword (Samurai) attacks.

  ############ Round 3 ############
  Chuck Norris (Samurai) attacks.
  Evil Mind (Samurai) attacks.

  ############ Round 4 ############
  Jackie Chan (Ninja) attacks.
  Ninja Assassin (Ninja) attacks.
*/
```

Polymorphism: *Deja vu?* I

- same function name, different behaviour(s): *Deja vu?*
- in **function overloading** (Week 2 Classes and Objects II: Constructors, and Operator Overloading)

```
#include <iostream>

class A {};

class B : public A {};

//function overloading:
//declarations (prototypes)
void print();
void print(int n);
void print(double d);

void print(A a);
void print(B b);

void print(A* aptr);
void print(B* bptr);
```

```
//function overloading:
//definitions (implementations)
void print() {
  std::cout << "print(void)" << std::endl;
}

void print(int n) {
  std::cout << "print(int)" << std::endl;
}
void print(double d) {
  std::cout << "print(double)" << std::endl;
}

void print(A a) {
  std::cout << "print(A)" << std::endl;
}
void print(B b) {
  std::cout << "print(B)" << std::endl;
}

void print(A* aptr) {
  std::cout << "print(A*)" << std::endl;
}
void print(B* bptr) {
  std::cout << "print(B*)" << std::endl;
}
```

Polymorphism: *Deja vu?* II

```
//main function
int main() {
  print();
  print(1);
  print(1.0);

  std::cout << std::endl;
  A a;
  B b;
  A* aptr;
  B* bptr;

  print(a);
  print(b);
  print(aptr);
  print(bptr);

  std::cout << std::endl;
  a = b;
  print(a);

  aptr = bptr;
  print(aptr);

  aptr = &b;
  print(aptr);

  aptr = new B();
  print(aptr);

  delete aptr;

  return 0;
}
```

```
/* Output:
  $ ./prog
  print(void)
  print(int)
  print(double)

  print(A)
  print(B)
  print(A*)
  print(B*)

  print(A)
  print(A*)
  print(A*)
  print(A*)
*/
```

Polymorphism modes

Function overloading

static binding (compile-time)

Inheritance and **virtual functions**

dynamic binding (runtime)

today: **Generic programming**

yet another mode of polymorphism
static binding (compile-time)

Insertion operator and inheritance/polymorphism

Problematic

```cpp
#include <iostream> //std::cout, std::endl, std::ostream

class A {
  public:
    A(int n1_in=0) : n1(n1_in) {}
  protected:
    int n1;
};

class B : public A {
  public:
    B() : A(), n2(0) {}
    B(int n1_in, int n2_in) : A(n1_in), n2(n2_in) {}

  private:
    int n2;
};

int main() {
  A a(1);
  B b(2, 3);

  std::cout << a << std::endl; // ERROR: no match for 'operator<<' ...
  std::cout << b << std::endl; // ERROR: no match for 'operator<<' ...

  return 0;
}
```

Insertion operator and inheritance/polymorphism

Solution 1: only one overload of operator<< (in the base class)

```
#include <iostream> //std::cout, std::endl, std::ostream

class A {
  public:
    A(int n1_in=0) : n1(n1_in) {}
    friend std::ostream& operator<<(std::ostream& os, A& a);

  protected:
    int n1;
};

class B : public A {
  public:
    B() : A(), n2(0) {}
    B(int n1_in, int n2_in) : A(n1_in), n2(n2_in) {}

  private:
    int n2;
};

std::ostream& operator<<(std::ostream& os, A& a) {
  os << a.n1;
  return os;
}

int main() {
  A a(1);
  B b(2, 3);

  std::cout << a << std::endl; //prints 1
  std::cout << b << std::endl; //prints 2 3? or only 2?

  return 0;
```

```
/* Output:
  $ ./prog
  1
  2
*/
```

Insertion operator and inheritance/polymorphism

Solution 2: two overloads of `operator<<` (in both base and derived classes)

```cpp
class A {
  //...
    friend std::ostream& operator<<(std::ostream& os, A& a);
};

class B : public A {
  //...
    friend std::ostream& operator<<(std::ostream& os, B& b);
};

std::ostream& operator<<(std::ostream& os, A& a) {
  os << a.n1;
  return os;
}

std::ostream& operator<<(std::ostream& os, B& b) {
  os << b.n1 << " " << b.n2;
  return os;
}

int main() {
  A a(1);
  B b(2, 3);

  std::cout << a << std::endl; //prints 1
  std::cout << b << std::endl; //prints 2 3? or only 2?

  A* aptr = new B(4, 5);
  std::cout << *aptr << std::endl; //prints 4 5? or only 4?

  delete aptr;
  return 0;
}
```

```
/* Output:
  $ ./prog
  1
  2 3
  4
*/
```

Insertion operator and inheritance/polymorphism

Solution 3: only one overload of `operator<<` (in the base class) + a polymorphic virtual function

```
class A {
  public:
    A(int n1_in=0) : n1(n1_in) {}
    virtual std::ostream& print(std::ostream& os); //virtual function. will
         ensure dynamic binding
    friend std::ostream& operator<<(std::ostream& os, A& a);

  protected:
    int n1;
};

class B : public A {
  public:
    B() : A(), n2(0) {}
    B(int n1_in, int n2_in) : A(n1_in), n2(n2_in) {}
    std::ostream& print(std::ostream& os);

  private:
    int n2;
};
```

Insertion operator and inheritance/polymorphism

Solution 3: only one overload of `operator<<` (in the base class) + a polymorphic virtual function

```
//std::ostream& operator<<(std::ostream& os, A& a) {
//  os << a.n1;
//  return os;
//}

std::ostream& operator<<(std::ostream& os, A& a) {
  return a.print(os); //call of the polymorphic virtual function
}

std::ostream& A::print(std::ostream& os) {
  os << n1;
  return os;
}

std::ostream& B::print(std::ostream& os) {
  os << n1 << " " << n2;
  return os;
}
```

Insertion operator and inheritance/polymorphism

Solution 3: only one overload of `operator<<` (in the base class) + a polymorphic virtual function

```
int main() {
  A a(1);
  B b(2, 3);

  std::cout << a << std::endl; //prints 1                /* Output:
  std::cout << b << std::endl; //prints 2 3? or only 2?      $ ./prog
                                                               1
  A* aptr = new B(4, 5);                                       2 3
  std::cout << *aptr << std::endl; //prints 4 5? or only 4?    4 5
                                                             */
  delete aptr;

  return 0;
}
```

Outline

1. Introduction
2. Templates
   - Function templates
   - Class templates
3. The Standard Template Library (STL)
4. Conclusion

Swapping (integers)

```
void my_swap(int& a, int& b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

Overloading my_swap

```
void my_swap(double& a, double& b){
    double c;
    c = a;
    a = b;
    b = c;
}
```

Further overloading

```
void my_swap(std::string& a, std::string& b){
    std::string c;
    c = a;
    a = b;
    b = c;
}
```

Thinking about overloading `my_swap`

- When does it end?
- Are the various versions of the function so different?
- For what types can it be overloaded successfully / meaningfully?

Generalising `my_swap`

```
void my_swap(<type>& a, <type>& b){
    <type> c;
    c = a;
    a = b;
    b = c;
}
```

- We can think of *generalising* `my_swap`.
- We factor out the common code – only the type declaration needs to be parameterised.
- As long as *the assignment operator (operator=)* is overloaded (correctly)!

Template functions syntax

```
template <class Type>
// or template <typename Type>
void my_swap(Type& a, Type& b){
      Type c;
      c = a;
      a = b;
      b = c;
}
```

Using template functions

```cpp
int main(){
    int i1 = 1, i2 = 2;
    double d1 = 1, d2 = 2;
    string s1("1"), s2("2");

    my_swap(i1, i2);
    my_swap(d1, d2);
    my_swap(s1, s2);

    cout << i1 << " " << i2 << endl;
    cout << d1 << " " << d2 << endl;
    cout << s1 << " " << s2 << endl;

    return 0;
}
```

```
/* Output:
  $ ./prog
  2 1
  2 1
  2 1
*/
```

- Nothing changes in the function call.

What happens with templates

- Templates are what the word suggests: blocks of code with placeholders.
- The compiler instantiates actual versions of the function when a function call with the specified parameters is encountered.
- In the previous example three versions are generated.
- Even with multiple calls, only one instantiation for each set of equal parameters (no code bloating).

Template functions: declaration and definition? I

- Does our usual way to separate declaration (in the header) and definition (in the implementation file) work?

```
//my_swap.hpp
#ifndef MY_SWAP_HPP
#define MY_SWAP_HPP

template <class Type>
void my_swap(Type& a, Type& b);

#endif
```

```
//my_swap.cpp
#include "my_swap.hpp"

template <class Type>
void my_swap(Type& a, Type& b){
    Type c;
    c = a;
    a = b;
    b = c;
}
```

Template functions: declaration and definition? II

```
#include "my_swap.hpp"

int main(){
    int i1 = 1, i2 = 2;
    // ...
    my_swap(i1, i2);
   // ...
}
```

- Compiling (g++ -c) each implementation file works.
- But g++ -o:

```
/* Output:
Error:
(.text+0x2a): undefined reference to ‘void my_swap<int>
collect2: ld returned 1 exit status
*/
```

Template functions: declaration and definition III

- A template declaration is not a full declaration.
- The compiler needs to create the actual function (the function is a 'template specialisation').
- The function is created (the process is 'template instantiation') when the compiler detects its use (call) (this is called '**point of instantiation**').
- At the point of instantiation the compiler will need not just the declaration but also the definition.

$\hookrightarrow$The most practical way to deal with this is to **fully specify template functions in the header file**.

Templates and subtyping I

```
class Base{
    public:
        Base(int b = 0) : base(b) {}
        // ...
    private:
        int base;
};

class Extend : public Base {
    public:
        Extend(int e = 0, int b = 0) : Base(b), extend(e) {}
        // ...
    private:
        int extend;
};
```

Templates and subtyping II

```
template <class Type>
void my_swap(Type& a, Type& b){
    Type c;
    c = a;
    a = b;
    b = c;
}

int main(){
    Base a;
    Extend b;
    my_swap(a,b);
  //Error: no matching function for call to
  //'my_swap(Base&, Extend&)'
}
```

- Type needs to be the same.
- (Would it make sense to pass arguments with subtyping in this case?)

Template parameter list I

```
template <class T1, class T2>
T2 my_add(T1 a, T2 b){
    return a+b;
}

int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) << endl;
    // ok (prints 25.15)

    return 0;
}
```

Template parameter list II

```
template <class T1, class T2>
T2 my_add(T1 a, T2 b){
    return a+b;
}

int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) << endl;
    // ok (prints 25.15)
    cout << my_add(n2, n1) << endl;
    // prints 25

    return 0;
}
```

Template functions and overloading v1

```
template <class T>
T my_add(T a, T b){
    cout << "using one parameter version" << endl;
    return a+b;
}

template <class T1, class T2>
T2 my_add(T1 a, T2 b){
    cout << "using two parameters version" << endl;
    return a+b;
}
```

```
int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) << endl;
    cout << my_add(n2, n1) << endl;
    cout << my_add(n1, n1) << endl;
    cout << my_add(n2, n2) << endl;

    return 0;
}
```

```
/* Output:
  $ ./prog
  using two parameters version
  25.15
  using two parameters version
  25
  using one parameter version
  20
  using one parameter version
  30.3
*/
```

Template functions and overloading v2 I

```
template <class T>
T my_add(T a, T b){
    cout << "using one parameter version" << endl;
    return a+b;
}

template <class T1, class T2>
T2 my_add(T1 a, T2 b){
    cout << "using two parameters version" << endl;
    return a+b;
}

int my_add(int a, int b){
    cout << "using int version" << endl;
    return a + b;
}
```

Template functions and overloading v2 II

```
int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) << endl;
    cout << my_add(n2, n1) << endl;
    cout << my_add(n1, n1) << endl;
    cout << my_add(n2, n2) << endl;

    return 0;
}
```

```
/* Output:
  $ ./prog
  using two parameters version
  25.15
  using two parameters version
  25
  using int version
  20
  using one parameter version
  30.3
*/
```

Template functions and overloading

- The **most specific** version is selected.
- Matching algorithm ranks versions (ambiguities can still occurr and are detected by the compiler).
- What if we want to force a version?

Template functions and overloading - forcing 1

```cpp
int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) << endl;
    cout << my_add(n2, n1) << endl;

    cout << my_add<int, int>(n1, n1) << endl;
  //forcing a version

    cout << my_add(n2, n2) << endl;

    return 0;
}
```

```
/* Output:
  $ ./prog
  using two parameters version
  25.15
  using two parameters version
  25
  using two parameters version
  20
  using one parameter version
  30.3
*/
```

Template functions and overloading - forcing 2

```
int main(){
    int n1 = 10;
    double n2 = 15.15;
    cout << my_add(n1, n2) <<
        endl;
    cout << my_add(n2, n1) <<
        endl;

    cout << my_add<int, int>(n1,
        n1) << endl;
  //forcing a version

    cout << my_add<int, int>(n2,
        n2) << endl;
  //forcing a version

    return 0;
}
```

```
/* Output:
  $ ./prog
  using two parameters version
  25.15
  using two parameters version
  25
  using two parameters version
  20
  using two parameters version
  30
*/
```

Class templates

```cpp
#ifndef MY_COMPLEX_HPP
#define MY_COMPLEX_HPP

template <class T>
class My_Complex{
    public:
        My_Complex(T i_r = 0, T i_img = 0):real(i_r), img(i_img){}
        // ...
    private:
        T real;
        T img;
};

#endif
```

- Declaration and definition in header file (like for function templates).

Class templates

```
template <class T>
class My_Complex{
    public:
        My_Complex(T i_r = 0, T i_img = 0);

    private:
        T real;
        T img;
};

template <class T>
My_Complex<T>::My_Complex(T i_r, T i_img): real(i_r), img(i_img){}
```

- Member function definition outside of class scope (with ::) is still possible.

Using the class template

```cpp
#include "My_Complex.hpp"

int main(){
    My_Complex<double> c;
    return 0;
}
```

Class templates: default parameters

```
template <class T = double>
class My_Complex{
    public:
        My_Complex(T i_r = 0, T i_img = 0);

    private:
        T real;
        T img;
};
```

```
template <class T = double>
class My_Complex{
    public:
        My_Complex(T i_r = 0, T i_img = 0);

    private:
        T real;
        T img;
};
```

- Originally available only in classes.
- New C++ standard allows them also in functions.

Class templates and friends

```cpp
template <class T = double>
class My_Complex{
    public:
        My_Complex(T i_r = 0, T i_img = 0);
        // ...
        template<typename T2>
        friend std::ostream& operator<<(std::ostream& out, My_Comp

    private:
        T real;
        T img;
};
```

```cpp
template<typename T>
ostream& operator<<(ostream& out, My_Complex<T> c){
    out << "(" << c.real << ", " << c.img << ")" << endl;
    return out;
}
```

Generic programming

- Function and class templates are the language features which enable *generic programming* in C++
- ***Generic programming***: approach software decomposition abstracting fundamental requirements on types from across concrete examples of algorithms and data structures.
  [D. Musser, A. Stepanov]
- For instance: 'types that can be swapped' (and one single function suitable for all of them).
- The aim, as usual, is writing 'abstract', 'general' code which can be re-used.
- Templates considered also as '**static polymorphism**', enabled at compile time (as opposed to the 'dynamic' one which happens at runtime).

Outline

The STL

- STL is the Standard Template Library
- Genericity of templates used to implement:
  - Container classes.
  - Iterators (to point at the containers content).
  - Algorithms (to process the data in the containers).
  - (Other features.)
- `vector` is a container class of the STL.

Traversing a vector

```cpp
#include <iostream>  //std::cout
#include <vector>    //std::vector
using namespace std;

int main() {
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  v.push_back(3);

  for(vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
  }

  return 0;
}
```

Iterators

- `vector<int>::iterator it;`
  declares a variable `it` of type 'iterator on vector of integers'.
- `vector<int>::iterator`
  is a *type* (not a member data variable).

Types in classes

```
class Container{
    public:
        // ...
        typedef size_t size_type;
        // this typedef really exists in vector.h
    private:
        // ...
};
```

```
Container::size_type cont_dim = 10;
cout << cont_dim << endl;
```

Iterators

- Iterators are a generalisation of pointers.
- The usual operations of pointers apply to `vector` iterators:
- Incrementing/decrementing (as in pointer arithmetic to point to the next/previos contiguous element).
- Dereferencing (to access the value of the pointed element).

Iterators in functions

```
template<class T>
void print_vector(const vector<T>& v){
    vector<T>::iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        cout << *it << endl;
    }
}
```

```
int main() {
  vector<int> v;

  v.push_back(1);
  v.push_back(2);
  v.push_back(3);

  print_vector(v);

  return 0;
}
```

Iterators as types I

```
template<class T>
void print_vector(const vector<T>& v){
    vector<T>::iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        cout << *it << endl;
    }
}
```

```
/* ERROR:
  In function 'void print_vector(const std::vector<T, std::
      allocator<_CharT> >&) [with T = int]':
  instantiated from here
  error: no match for 'operator=' in 'it = ((const std::
      vector<int, std::allocator<int> >*)v)->std::vector<_Tp
      , _Alloc>::begin [with _Tp = int, _Alloc = std::
      allocator<int>]()'
```

- Compiler can't disambiguate if iterator is a type or a member data variable
- vector<T> is something only partially defined which depends

Iterators as types II

```
template<class T>
void print_vector(const vector<T>& v){
    vector<T>::iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        cout << *it << endl;
    }
}

/* ERROR:
  In function 'void print_vector(const std::vector<T, std::
      allocator<_CharT> >&) [with T = int]':
  instantiated from here
  error: no match for 'operator=' in 'it = ((const std::
      vector<int, std::allocator<int> >*)v)->std::vector<_Tp
      , _Alloc>::begin [with _Tp = int, _Alloc = std::
      allocator<int>]()'
```

- Compiler assumes the latter holds (`iterator` as member data variable) and raises syntax error.
- But we are also given very clear advice.

const vectors - What's wrong now?

```
template<class T>
void print_vector(const vector<T>& v){
    typename vector<T>::iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        cout << *it << endl;
    }
}
```

```
/* ERROR:
  In function 'void print_vector(const std::vector<T, std::
      allocator<_CharT> >&) [with T = int]':
  instantiated from here
  error: no match for 'operator=' in 'it = ((const std::
      vector<int, std::allocator<int> >*)v)->std::vector<_Tp
      , _Alloc>::begin [with _Tp = int, _Alloc = std::
      allocator<int>]()'
*/
```

const vectors - const_iterators

```
template<class T>
void print_vector(const vector<T>& v){
    typename vector<T>::const_iterator it;
    for(it = v.begin(); it != v.end(); ++it){
        cout << *it << endl;
    }
}
```

```
/* Output:
  $ ./prog
  1
  2
  3
*/
```

- const_iterator can access but not change the value of the pointed element.

Inserting in a vector

```
int main() {
  vector<int> v(10);
    v.insert( (v.begin() + 3), 1);
    // now v[3] is 1 and v.size() is 11
}
```

- `iterator insert(iterator position, const T& x);`
- position needs to be between begin and end.
- Insertion happens *before* element in position.
- Returns iterator to newly inserted element.
- Elements following the newly inserted one (from position included, on) are moved of one place towards the end.
- Like for `push_back`: re-allocation (all pointers, references, iterators are invalidated).

Sorting a vector

```
#include <iostream>  //std::cout                  /* Output:
#include <vector>    //std::vector                  $ ./prog
#include <algorithm> //std::sort                    1
using namespace std;                                2
                                                    3
int main() {                                      */
  vector<int> v;
  v.push_back(3); v.push_back(2); v.push_back(1);

  sort(v.begin(), v.end()); //sorts the vector using <

  for(vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
  }
}
```

- One of STL algorithms. (Not a member function.)
- We can specify to sort also a limited range.

Other containers: `list`

- `list`, represents the usual (doubly) linked list.
- Sequential container like `vector`.
- But not random access.
- `sort` in `<algorithm>` cannot be used (has own version as member function).
- Inserting and deleting elements is more efficient than in `vector`.
- Insert and delete don't invalidate pointers and iterators (except for those pointing to a deleted element).

Sorting a list

```
int main(){
    list<int> l;
    l.push_back(1);
    l.push_back(2);
    l.push_back(3);
    list<int>::iterator it = l.begin();
//      l.insert(l.begin()+2, 15);
//      no, no random access
    it++;
    it++;
    l.insert(it, 15);
    for(list<int>::iterator it = l.begin(); it != l.end(); ++it){
        cout << *it << endl;
    }
//      sort(l.begin(), l.end());
//      no, no random access
    l.sort();
    for(list<int>::iterator it = l.begin(); it != l.end(); ++it){
        cout << *it << endl;
    }
    return 0;
}
```

Other containers

Sequence containers

- `array`
- `vector`, `list`
- `deque` (Double ended queue)

Container adaptors

- `stack` (LIFO)
- `queue` (FIFO)
- `priority_queue`

Associative containers

- `set`, `multiset`
- `map`, `multimap`

- ...
- The STL reference: http://www.cplusplus.com/reference/stl/

Outline

Summary

- Generic Programming in C++:
    - Templates: **Function templates** & **Class templates**
    - The STL: **Containers** & **Iterators** & **Algorithms**

- yet another mode of **polymorphism** (+ function overloading & inheritance/virtual functions)

What to do next?

Next Lab

- Swapping with several types
- Revisiting previous labs
  - using iterators
  - alternatives to `std::vector`
- Using `<algorithm>`

Next Lecture

Week 8 - Exceptions Handling