# CS 247 — Software Engineering Principles

Kevin James

Spring 2014

## Contents

# 1 Object-Orientation

Code which follows **object-oriented** patterns has better affinity between user-defined types and real-world types, can be more easily reused (inheritance, composition, as-is, polymorphism), allows for encapsulation and information hiding, decomposes extremely effectively (and thus separates concerns), and allows for abstraction through ADTs (Abstract data types), interfaces, etc.

## 1.1 ADTs

An ADT is a user-defined data type. They are composed with a range of legal values and functions that maniplate variables of a given type. By providing compiler support for these type restrictions, we turn programmer errors into type errors (which are checked by the compiler).

One of the main motivations for designing an ADT is to ensure safety of any client code. Other motivations include evolvability and scalability. ADTs also tend to improve code efficiency by limiting range checks to constructors and mutators.

An ADT constructor initializes the new object to some legal value and throws an error if the passed-in value is illegal. Accessors and mutators provide restricted read/write access to one of the values in the object. It is best practice to ensure legality within each mutator and use `const` references whenever possible.

We use the **outside-in** method fo development: first we determine what the user wants out of it, then we design and implement it.

**Function Overloading** is syntactic sugar which allows us to define a function with the same name as some other function, so long as this new function has different parameters. It is generally best practice to only overload functions when the purpose of the function is the same for each. For example: `void print(int)` and `void printfloat`.

Another option is to use **default arguments**. Defualt arguments tend to be used when a given argument should be optional. For example: `void print(int, outputStream=cout)`. Default arguments must appear only in the function declaration.

We can also overload operators. For example, we could define the sum of two classes as some other class, some modified version of one class, or even a standard data type of some sort with `MyClass operator+(const MyClass&, const MyClass&)`. Though widely used, this practice should be used sparingly, if at all.

**Nonmember functions** are critical ADT functions which are declared outside of the class. This leads to ebtter encapsulation and more flexible packaging. Additonally, certain functions are required to be nonmember functions (e.g. `operator>>`). Streaming operators should be nonmember functions so that they can accept a stream as a first operand and thus chain stream operations.

A class can have `private`, `protected`, and `public` data members and functions (in addition to several less-often used flags). It is best practice to use the most secure flag as possible (generally: `private`); though the `public` flag is sometimes necessary, the `protected` flag should be avoided. We can also use the `friend` flag to create a `private` method which is accessible to a given other class.

## 1.2 Inheritance

**Inheritance** allows us to create classes (**derived classes**) which include the data members and functions of a **base class**.

## 1.3 Polymorphism

When we create a derived class, that class can inherit pointers from its parent.

**Object slicing** occurs when only base class fields are copied from a subclass. It can occur as a side-effect of intereactions between a subclass and a superclass: passing the sublcass by value, assigning the subclass to the superclass, or suing the superclass assignment operator between subclass instances.

**Function binding** occurs at compile time. When a base class pointer is given a reference to a subclass after its declaration, the subclasses methods will not be called.

**Virtual functions** are a method of getting around this. A virtual function delays function lookup to runtime, and thus will ensure the subclass function will be called even when using a pointer to the base class.

When having an instance of a base class doesn't make sense, we can make a **pure virtual function**. Pure virtual functions need no definitition and prevent class instatiation. They also ensure all subclasses must have a definition for that function.

## 1.4 Overloading

**Overloading** occurs when two functions with the same name have a different set (amount or type) of parameters.

## 1.5 Overriding

**Overriding** occurs in subclasses, when a subclass function has the same signature as a superclass function. When this function is called from the derived class, the derived class' version of the function will be called.

## 1.6 Friendship

The **friend** command allows us to declare a class or function which can access the private members of the class containing the friend declaration.

For example:

```
class Base {
    int priv;
public:
    friend void print(Base);
};

void print (Base esab) {
    cout << esab.priv << endl;
}
```

### 1.7 Helper Functions

The best practice is to hide helper functions as `private` methods or within a namespace. Helper functions modularize our code, but should not pollute the gloal namespace.

## 2 Entity vs Value Objects

**Entity Objects** are the digital embodiment of a real-world entity. Each object has a distinct identity and objects with the same attribute values are not equal. **Value Objects** simply represent a value of an ADT. Value objects with the same attribute values are considered to be identical.

An operation on an entity object should reflect a real-world event: copying is not meaningful—though cloning may be, entities referred to by pointers are useless (due to the no-copy rule), and computation on entities are not meaningful (overload `new` and `delete`, maybe `operator<`). Virtual functions and inheritance are uncommon for value-based ADTs, though equality and computations are useful.

### 2.1 Singleton Design Pattern

The **Singleton Design Pattern** ensures that only one object of our ADT exists. By using a structure of the form

```
class ADT {
    static ADT a;
    ADT() {}
    ADT(const ADT&);
public:
    static ADT* instance() {
        return &a;
    }
};
```

we can ensure that the singleton is created the first time, then *referenced* every time thereafter.

## 2.2 Essential Methods

Some C++ member functions are necessary for all object definitions and will thus be inserted by default if no implementation is provided. These include

- the default constructor (if and only if we do not define any other constructor)
- the destructor
- the copy constructor
- the assignment operator

in addition, the equality operator should be defined for all objects, though it has no default implementation.

If we do not declare a copy constructor (a constructor which copies class instances), the default copy constructor will be created based on **memberwise initialization** such that

- simple data members are bitwise copied
- pointer members are bitwise copied
- member objects are copied using their copy constructors
- inherited members are copied using thir copy constructors

The assignment operator follows the same pattern as the above.

The default constructor follows a slightly different set of default values (**memberwise initialization**)

- simple data members are uninitialized
- pointer members are uninitialized
- member objects are initialized using their default constructors
- inherited members are initialized using their default constructors

The deconstructor is similar since it uses **memberwise destruction**

- simple data mambers are deallocated
- pointer members deallocate the pointer
- member objects call their destructor
- inherited members call their destructor

## 2.3 Mutable vs Immutable Objects

Entity objects are **mutable**, that is, their objects can change via mutators or other functions. Value objects, though, are often **immutable**: their objects can not change value, a new object must be created instead.

It is possible, though, to design a mutable value-based ADT:

```
Person myPerson("David", new Date(1, "May", 1990));
cout << myPerson.DOB;

Date myDate = myPerson.DOB;
myDate.monthIs(myDate + 1);
cout << myPerson.DOB;
```

This is generally a bad design: this ADT has a mutable date field if and only if the constructor did not create a **deep copy** of the input date and the accessor did not return a deep copy. If either of these do not create a copy, the original, mutable object continues to be accessible externally.

### 2.3.1 Copy Types

A deep copy creates an entirely new instance of an object.

```
class ADT {
    Date d;
public:
    void setDate(const Date din) {
        d = new Date(din);
    }
}
```

A **shallow copy** simply creates a pointer to the copied object.

```
class ADT {
    Date d;
public:
    void setDate(const Date din) {
        d = din;
    }
}
```

## 3 Polymorphism

The compiled byte code contains executable code for every function and method definition. The compiler can see at compile time which method should be executed. When we add virtual functions, each class with such a function will have a **vtable** of pointers to these functions and a pointer to its vtable. This allows us to call the same functions from multiple subclasses without reimplementing them.

The general convention as to whether a public function is to be made virtual is made for the class as a whole instead of for each individual function. This is generally based on whether the class should be polymorphic or not.

We can use the assignment operator to assign derived class instances to base class instances.

```
int main (void) {
    Base b(42, 'x');
    Derived d(10, 'c', 8.1);
    b = d;
}
```

This assignment will not copy the derived class' extra data members or vpointers. We call this **object slicing**.

If there exists any virtual function, the class is polymorphic and should thus have a virtual destructor. If we do not implement this manually, the compiler will default to creating a non-virtual constructor.

In order to print a polymorphic function, we generally

- create a `virtual void print(std::ostream& os) const;` function which does the actual printing
- create a non-member function `std::ostream& operator<<(std::ostream& os, const ADT& adt);` which takes an instance of the class and calls the `print()` member function

The overloaded streaming operator generally looks similar to

```
std::ostream& operator<<(std::ostream& os, const ADT& adt) {
    adt.print(os);
    return os;
}
```

In general, we have

```
class ADT {
public:
    ADT();
    ADT(ADT& adt);
    virtual ~ADT();
    void operator=(ADT& adt);
    virtual bool operator==(ADT& adt);
    virtual void print(std::ostream& os);
}
```

## 4 Compilation

We use **header files** to contain our declarations and **code files** to contain their implementations. Header files may be `#include`'d by other modules so that they can use the interface without relying on a specific implementation.

To ensure that header files are not included multiple times, we use the following **header guard**:

For the file `sample.h`, we do:

```
#ifndef SAMPLE_H
```

```
#define SAMPLE_H

// contents of header file

#endif
```

We can use a similar process within files when we need to reference `class A` from `class B` and vice-versa. To avoid a circular dependency, we can use **forward declarations** as such

```
// A.h
class B;

class A {
    // declarations involving B
}
```

and

```
// B.h
class A;

class B {
    // declarations involving A
}
```

Since compiling and linking in C++ are separate operations, we can compile each file separately. This allows us to recompile only those files that have changed.

Note that if a file changes, we need to recompile it as well as all other files which depend on it. This would be much easier with an automated build system.

## 4.1 Make, an Automated Build System

`make` is a UNIX command which uses build instructions and file dependencies provided in a `Makefile` as well as file timestamps to ensure it only recompiles changed files and their children.

We can create Makefiles with macros, implicit rule, and automatic dependency derivations.

```
CXX_FLAGS=-g -Wall

GTEST_DIR=~/gtest
GTEST_FLAGS=-isystem $(GTEST_DIR)/include -pthread
GTEST_LIBS=$(GTEST_DIR)/lib/*


.SUFFIXES:
.SUFFIXES: .o .cpp

.cpp.o:
    ${CXX} $(CXX_FLAGS) -c $<
```

```
all: program1 program2

test: program1_test program2_test


program1: Program1.o Program1Dependency.o Common.o
    ${CXX} $(CXX_FLAGS) $^ -o $@

program1_test: Program1Test.cpp Program1.o Program1Dependency.o Common.o
    ${CXX} $(GTEST_FLAGS) $(CXX_FLAGS) $^ $(GTEST_LIBS) -o $@;
    ./$@

program2: Program2.o Program2Dependency.o Common.o
    ${CXX} $(CXX_FLAGS) $^ -o $@

program2_test: Program2Test.cpp Program2.o Program2Dependency.o Common.o
    ${CXX} $(GTEST_FLAGS) $(CXX_FLAGS) $^ $(GTEST_LIBS) -o $@;
    ./$@


clean:
    rm -f *.o
    rm -f program1 program1_test
    rm -f program2 program2_test
```

This reasonably complicated Makefile should be named `Makefile` (no extension) and placed in the development directory. It assumes we are uses `gtest` as a test suite and have the following files in the current folder: `Program1.cpp`, `Program1.h`, `Program1Dependency.cpp`, `Program1Dependency.h`, `Program1Test.cpp`, `Program2.cpp`, `Program2.h`, `Program2Dependency.cpp`, `Program2Dependency.h`, `Program2Test.cpp`, `Common.cpp`, `Common.h` .

It relies on the `CXX` environment variable which **SHOULD NOT BE OVERRIDEN IN A MAKEFILE** since it is defined for each user by the specific compiler they use for C++ files (e.g. g++, clang++, . . . ).

It has defined the `.cpp.o` suffix, which auto-creates non-existing or out-of-date object files from their particular source file. This will be called implicitely by declaring object files as dependencies to various targets (e.g. `all`, `test`, `clean`, `program1`, . . . ). Note that I have not defined the dependencies for each object file (e.g. `Common.o` is based on `Common.cpp` and `Common.h`, but I have not explicitely declared this). Any object file will by default be built from a file of the same name with the `cpp` extension. If we want to change these dependencies, we can do so with

```
...
test: program1_test program2_test


Common.o: Common.cpp First.cpp Second.cpp
```

```
program1: ...
```

`all` is the standard name given to the set of commands which should be run by default. It should be responsible for compiling and linking all program components and will be run by default by calling `make` on the command-line.

`test` is the standard for running a test-suite during development. `check` should be used for self-testing once a program is installable (which should be with the `install` target).

`clean` should be used to remove all files generated by the Makefile, i.e. files which can be rebuilt.

Further standard targets can be found at `http://bit.ly/1nSOZPG`.

## 5 Exceptions

An **exception** is an unusual event or situtation which prevents a function from completing normally. In C++, we handle exceptions in a separate section from our normal code: risky and risk-free code is separated, different areas of our program can handle exceptions in a different manner, and errors can not be silently ignored.

We handle errors in a block of code with

```
try {
    // risky code
} catch(exceptionType1& e) {
    std::cout << "Generated exception 1" << std::endl;
    system.exit(-1);
} catch(exceptionType2& e) {
    std::cout << "Generated exception 2" << std::endl;
    system.exit(-2);
} catch(...) {
    std::cout << "Generated unspecified exception" << std::endl;
    throw;
}
```

An exception is handled by the "nearest" handler whose argument matches its type. A **local exception** is one which is handled in the same routine as it is thrown (e.g. through an alternative computatino or return value). Otherwise, the exception is caught by the dynamically nearest matching catch-clause whose try-block encloses the throw. If there is no matching handler, the program aborts.

A function may declare a **throw list** of potential exceptions it can throw. An empty list implies it does not throw errors, a missing list implies it may or may not throw. If a function throws an error not in its throw list (if it has one), that exception is considered `unspecified` and will crash the program (unless the `unspecified()` function has been overloaded).

In order to fully support blind inheritance and allow libraries to change, it is best practice not to use throw lists.

## 6 Resource Acquistion is Initialization (RAII)

The **RAII** idiom equates resource management with object lifetimes. The resource is allocated in the objects constructor and deallocated in its destructor. The standard function signatures are `resourceType* allocate(...)` and `void release(resourceType*)`

### 6.1 Smart Pointers

A **smart pointer** is an ADT which simulates a pointer while using RAII to provide automatic deallocation. In C99, the `auto_ptr<type>` object is used for this. In C++11, we have `unique_ptr<type>` for single-owner transferable references and `shared_ptr<type>` for multiple-owner automatic deallocation upon lack of references.

An `auto_ptr` is not a substitute for a standard pointer.

- the `operator=` is different
- we cannot assign an `auto_ptr` to a non-`auto_ptr`
- we cannot pass an `auto_ptr` to a function parameter that is not an `auto_ptr`
- we cannot use `auto_ptr`s in STL containers

To use an `auto_ptr`, we must `#include <memory>`

An `auto_ptr` requires less explicit management by the programmer, since all standard tasks are dealt with automatically. However, since only one `auto_ptr` can refer to any unique object, the `auto_ptr` may not always be useable.

We can use an `auto_ptr` as follows

```
#include <memory>

std::auto_ptr<int> pointer;

std::cout << *pointer.get() << std::endl;
// prints ‘‘0’’

pointer.reset(42);
std::cout << *pointer.get() << std::endl;
// prints ‘‘42’’
```

## 7 Development Practices

There are many problem areas in program development: code ownership, testing, managing complex requirements, ...

## 7.1 Waterfall Model

The waterfall model imitates standard engineering procedures: the requirements are developed, then a design plan is created, development is completed, and finally the system is tested. This model is commonly accepted to be flawed: writing software is not the same as architecting a building.

## 7.2 Agile Programming

There are several methodologies which can be described as agile programming: **scrum**, **extreme**, ...

### 7.2.1 Extreme Programming

**eXtreme Programming (XP)** is a modern software development process models which allows the programmer to lead the process. The goal is to produce high-quality software by "keeping programming fun". Though the name suggest high-risk and haphazard development, the opposite is true.

XP encompasses several business ideas, but the main three are pair programming, **design simplicity**, and **automated testing**.

The basic rule of **pair programming** is "code is never written or modified unless two people are sitting side-by-side in front of one computer". Programming becomes a dialogue: only one person types, but both people analye, design, program, and test. Pairing is not a long-term commitment; groups only need to remain together for the duration of a single task.

Design cimplicity ensures that a short-term pairing can be productive. By establishing coding standards and writing test code together, the pair can avoid trivial misunderstandings and reach a common understanding.

Pair programming also fosters collective ownership. Any programmer may change any portion of the code at any given time. System integration takes place as often as possible to identify conflicting changes. A working version always exists.

The basic rule of **design simplicity** is "build the simplest thing that works". Simple deigns lead to code that can easily be modified or replaced as requirements evolve. "Embrace change" is the motto of XP.

Pair programming forces an immediate "simplicity check" as design ideas are explained to the second programmer. Automated testing gives programmers the confidence that design changes have been correctly implemented. An overall system metaphor guides the design process.

A simple design has no duplicated code or logic. **Refactoring** is the process of simplifying code to avoid or eliminate duplication. For example: code may be refactored to combine two similar classes or to extend them from the same base class.

The baseic rule of **automated testing** is "write the tests first". Automated tests should become an integral component of the program, which can be allowed to self-test at any time. System integration takes place as often as possible. Automated testing ensures that a working version of the program always exists.

Design simplicity makes it easy to write tests. Each programmer thinks of ways to test their partner's ideas. A successful test provides the gratification which programmers crave.

# 8 Design Patterns

**Design patterns** are codified solutions that put design principles into practice to improve the modularity and flexibility of our code.

## 8.1 Template Method Pattern

**Problem:** duplicate code. Multiple sublcass methods have simila algotihm structure.

**Solution:** localize duplicate code structure in abstract class. Abstract class defines a template method (of common code) that calls pure virtual subroutines. Subclasses override the subroutines.

Alternatively, a template method is a bse-class method that defines a common code structure but includes primitive operations (holes) to be defined by subclass methods. It is essential that the template method be nonvirtual and the primitive operations be virtual functions in the base class.

## 8.2 Adaptor Pattern

**Problem:** interface mismatch between two modules. If you want to reuse an existing class though its interface does not match what is needed. Alternatively, if the interface of one of our modules changes and we don't want to make major changes to the existing code.

**Solution:** define an adaptor class that maps one interface to another.

## 8.3 Facade Pattern

**Problem:** complex interface. Client of a subsystem interacts with multiple complex classes.

**Solution:** create a single, simplified interface (class). Restrict, simplify client's interactions with subsystems' classes.

## 8.4 Strategy Pattern

**Problem:** want to vary an algorithm at runtime.

**Solution:** encapsulate the algorithm decision. Define algorithm to a component object and use subclassing to specialize the algorithm in different ways.

## 8.5 Observer Pattern

**Problem:** we have a set of data which we want to display or use in multiple places.

One way to solve this would be to create a `updateAll()` function. However, having to maintain a dependency chart within a member function is generally not a good idea: we would need to change code in multiple places to add new data and all state-changing operations would have to call this update function. Furthermore, changes at runtime may not be properly propogated.

**Solution:** alternatively, we could ensure that all classes relying on our changed data ("Observers") are within a collection in the data ("Subject") class and derive from a base class with an `update(state)` method. Then, the Subject class simply needs to iterate through this collection and call the `update()` method on each of them.

We can extend this second option by adding member functions to the Subject class which will add and remove displays from the collection. Traditionally, we use the Subject member functions `subscribe()`, `unsubscribe()`, and `change()`.

Any classes derived from the Subject class may provide methods for getting and settings state information; these, of course, must call the `change()` method of the base class. Any clases derived from the Observer class must simply reimplement the `update(state)` function.

The observer pattern described above is the **push** form: data is pushed along with the `update` call to each observer. Instead, we can send empty `update` method calls and instead have each observer access the data from our Subject class upon update. This can be useful when various observers care about different state information.

Either implementation of the observer pattern minimizes coupling between Subjects and Observers: the resulting classes are thus easier to reuse and the Observers can easily change at runtime.

## 8.6 Model-View-Controller Pattern

The **MVC** design pattern is a collection of design patterns which decouple UI code from application code. The Model is our backend application code, the View is the frontend UI code, and the Controller is the interface between the two which takes in user input and translates that to an application code call.

MVC makes heavy use of observers (the view is an observer of the model), strategies (the controller varies the model), and composites (throughout).

The main function for a MVC program (more specifically, for a GTK-based project) would be written as

```
int main(int argc, char *argv[]) {
    Gtk::main kit(argc, argv);

    Model model;
    Controller controller(&model);
    View view(&controller, &model);
    Gtk::main::run(view);

    return 0;
}
```

## 9 UML Modelling

A model is an abstraction of something for the purpose of understanding it before building it, communicating it to others, and answering questions about it.

**Unified Modelling Language** is a collection of notations for representing different views of a software design.

Structural Diagrams:

- class diagram
- component diagram
- composite structure diagram
- deployment diagram
- object diagram
- package diagram
- profile diagram

Behaviour Diagrams:

- activity diagram
- communication diagram
- interaction overview diagram
- sequence diagram
- state diagram
- timing diagram
- use case diagram

### 9.1 UML Class Diagrams

A box represents a class and defines the class name, the set of attributes and initial values, and the set of operations and signatures. These can be expressed at different levels of abstraction: just the name, the name and public attributes, or everything along with their individual private/public/virtual etc. status.

An **association** between two classes indicates that there exists a physical or conceptual link between objects of those classes.

**Multiplicity** annotations constrain the number of allowable links in association. We can either give specific values (`x`), ranges of values (`x...y`), or "greater-than" values (`x...*`)—note that we can use `*` to denote "any number". If we have no annotation, the multiplicity is unspecified.

A class association represents link attributes: properties of the link which cannot be attributed to either object. A composition is a stronger "part of" relation between a composite object and its components. A part does not exist without its composite (and belongs to at most one composite) and the composite is responsible for creating and destroying members.

The UML uses the term **generalization** for the subtype relationship between a base class and its derived classes. Every member of a derived class is a member of its base class. Attributes and associations of the base class are attributes and associations of the derived class.

### 9.2 UML Object Models

An **object model** is a runtime instance of a class model: every object is an instantiation of a specific class and every link between two objects is an instantiation of a specific association.

The value of each assigned data member is represented, which allows us to easily analyse the program state.

### 9.3 UML Sequence Diagram

A **UML Sequence Diagram** is a graphical model of communication events between objects, as exhibitied in one execution trace. It is similar to a timeline view of your program specific to several classes / functions.

## 10 Object Oriented Design Principles

Object Oriented Design Principles are a set of guidelines which improve program quality and modularity.

### 10.1 Open-Closed Principle

A module should be open for extension but closed to modification. For example, we prefer to have client code depend on an abstract class (which could be extended) rather than a concrete one.

It is possible to inherit both interfaces or implementations. The abstract class designer determines which parts of a member function the dervied class inherits: the interface, the interface and default implementation, or the interface and non-overridable implementation.

## 10.2 Inheritance vs Composition

When defining a new class that includes attributes and capailities of an existing class, should our new class inherit for the base class or include the class as a complex attribute? In generally, we prefer composition: composition can change at run-time (rather than being a static relationship between classes) and can not break encapsulation; we only prefer inheritance when we are subtyping or using an entire interface. Basically, composition is black box reuse where inheritance is white box reuse.

## 10.3 Delegation

Delegation in object composition stimulates inheritance-based object method reuse. The composite object delegates to the component object and can pass itself as a parameter to let the delegated operation refer to the composite object.

The benefits of composition are maximized when the component is an abstract type which can be concretized in multiple ways.

## 10.4 Dependency Inversion

High-level modules should depend on abstractions rather than on concrete classes. If we change the standard inheritance paradigm ("client extends server") to "client and server both inherit from a common interface", we can ensure that the two modules never have any problems interfacing with each other.

## 10.5 Single Responsibility Principle

Encapsulate each changeable design decision in a separate module. The **single-responsibility principle** offers guidance on how to decompose our program into cohesive modules.

## 10.6 Liskov Substitutability Principle

The **Liskov substitution principle** enforces strong behavioural subtyping, which is basically the following: if $S$ is a subtype of $T$, then any properties provable of $T$ must also be proveable of $S$. Less formally, any base class instance should be replaceable by a subclass instance.

## 10.7 Law of Demeter

The Law of Demeter is the principle of least knowledge and suggests loose coupling. It states that

- each unit should have only limited information about other units (e.g. only those which are very related)

- each unit should only interact with friends
- each unit should prefer only interacting with immediate friends

## 11 Templates

Suppose we want to create our own generic classes and functions. A **function template** describes this sort of family:

```
template<typename T>
int compare(const T &lhs, const T &rhs) {
    if(lhs < rhs) {
        return -1;
    }
    if(lhs > rhs) {
        return 1;
    }
    return 0;
}
```

We can use this code as `compare(3, 4);`, `compare(3.14, 4.2);`, etc.

If we would like to compare different types, we could use

```
template<typename T1, typename T2>
int compare(const T1 &lhs, const T2 &rhs) { ... }
```

whether or not we would find such a thing useful (this allows for code such as `compare(‘‘whale’’, 42);`) is left as an exercise to the reader.

We can also return templates with

```
template<typename T>
T compare(const int &lhs, const int &rhs) { ... }
```

at which point we must make explicit our desired return type, e.g. `compare<float>(3,4);`.

Note that in the case of code such as

```
template<typename T1, typename T2, typename T3>
T1 compare(const T2 &lhs, const T3 &rhs) { ... }
```

this explicitness works as follows: `compare</*T1 = */float, /*T2 = */int, /*T3 = */float>(3, 3.14);`, i.e. the templates must be assigned in order.

We can also create templated class definitions:

```
template<typename T>
class Stack {
    T _items[STACK_SIZE];
public:
    Stack();
    void push(const &T);
    T pop();
}

template<typename T>
void Stack<T>::push(const T &elem) { ... }
```

The template handle also allows us to provide default but compile-time editable variables as follows:

```
template<typename T, int size = 100>
class Stack {
    T _items[size];
    ...
}
```

which can be used as `Stack<T> myStack;` to create a Stack of default size 100 or as `Stack<T, 42> myStack;` to create a Stack with a non-default size (e.g. 42).

If we make assumptions within our templated code, that can place restrictions on which types may be templated. For example:

```
template<typename T>
T mumble(T val) {
    val.speak();
    std::cout << val << std::endl;
    return "Frog";
}
```

will only compile if there exists some type or class which has a `speak()` method, an output stream operator, and can be typecase from a string.

## 11.1 Friends

There are three types of friends declarations which may appear in a class templat. Each type of declaration describes friendship to one or more entities:

1. a friend declaration for an ordinary nontemplate class or function, which grants friendship to a specific named class or function
2. a friend declaration for a class template or function template, which grants access to all instances of the friend
3. a friend declaration which grants access only to a specific instance of a class or function template

# 12 STL

The **C++ Standard Template Library** is a major component of the Standard Library. It contains **containers** for primitive data types, **algorithms**, and **iterators**.

These containers know nothing about the elements they contain and function in the same way no matter what type the elements are. Mostly, containers focus on membership (`insert`, `erase`, etc). Containers also know nothing about any associated algorithms and can define their own iterators.

The algorithms know nothing about the data structures they operate on and almost nothing about the contained elements. Instead, algorithms use container iterators to apply their functionality.

Some algorithms overwright elements values in an existing container. We must take care to ensure that the destination sequence is large enough for the number of elements being written. For example:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);
```

requires that `result` be at least the size of the input container.

Algorithms never directly change the size of containers; only container operators can add or remove elements. Instead, algoriths rearrange elements—sometimes by placing undesirable elements at the end of the container and returning an iterator past the last *valid* element.

Some algorithms apply operations to the elements in their input range, e.g. `transform()`, `sort()`, etc. Some algorithms also accept a predicate which is applied to all elements and is used to restrict the set of data elements upon which our algorithm operates. For example:

```
bool gt20(int x) { return 20 < x; }

remove_copy_if(input.begin(), input.end(), output.begin(), gt20);
```

If we need a function which refers to data other than iterated elements, we need to define a **function object**:

```
class gt_n {
    int value;
public:
    gt_n(int val) : value(val) { }

    bool operator()(int n) { return n > value; }
}

int main() {
    gt_n gt4(4);
    std::cout << gt4(3) << std::endl;  // prints 0
    std::cout << gt4(5) << std::endl;  // prints 1
}
```