C++ crash course for C programmers

Nexus Team

Teacher's introduction

MAHDI AKBARI ZARKESH

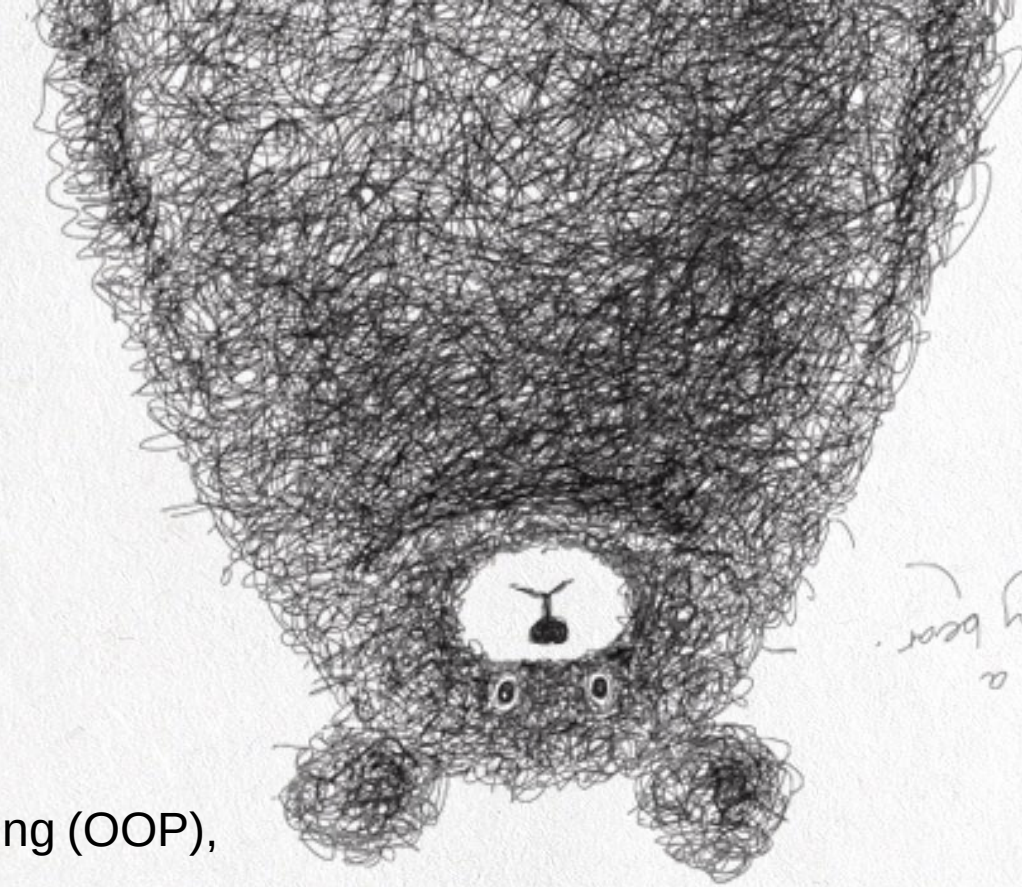
maz1377

CONTENTS

PART 01

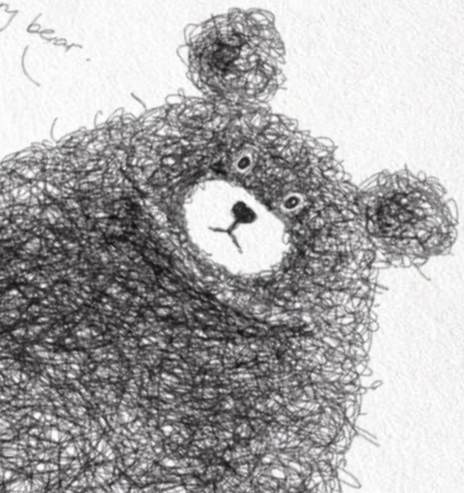
From C to C++

Even if C++ is slanted toward object-oriented programming (OOP), you can nevertheless use any c++ compiler to compile c code and benefits from some c++ goodies.



Input/Output

Prefer the use of <iostream> for input/output operations



```cpp
#include <iostream>

int main (int argc, char **argv)
{
    int i;
    std::cout << "Please enter an integer value: ";
    std::cin >> i;
    std::cout << "The value you entered is " << i  << std::endl;
    return 0;
}
```

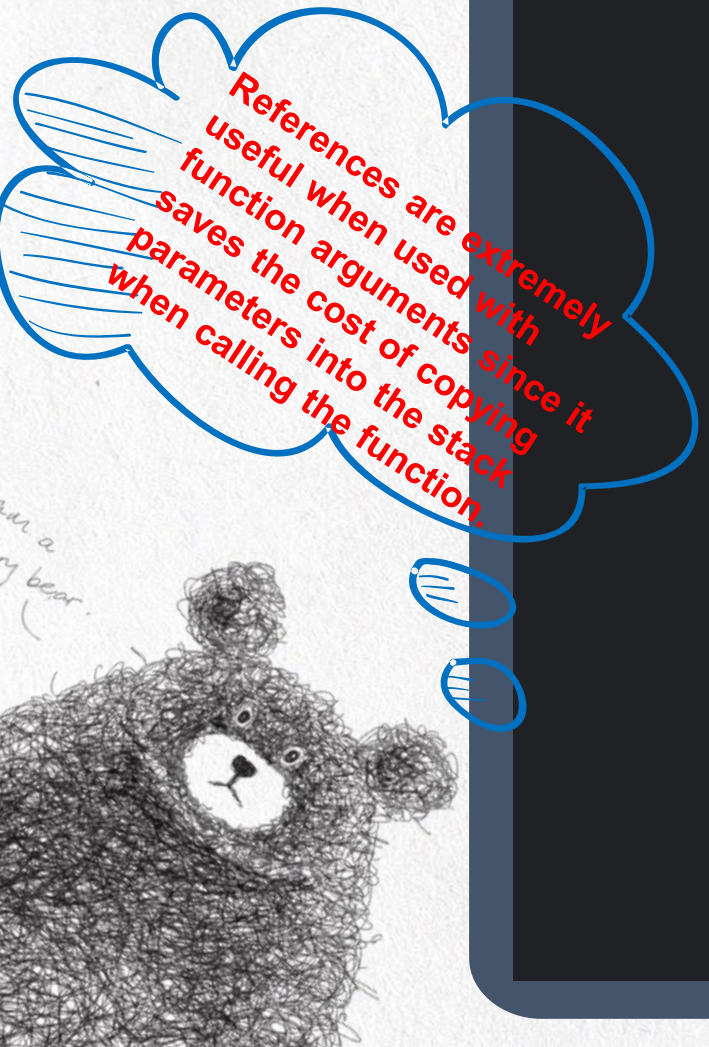
New/Delete

The new and delete keywords are used to allocate and free memory. They are "object-aware" so you'd better use them instead of malloc and free. In any case, never cross the streams (new/free or malloc/delete). the use of <iostream> for input/output operations


```
int *a = new int;
delete a;

int *b = new int[5];
delete [] b;
```

References

A reference allows to declare an alias to another variable. As long as the aliased variable lives, you can use indifferently the variable or the alias.

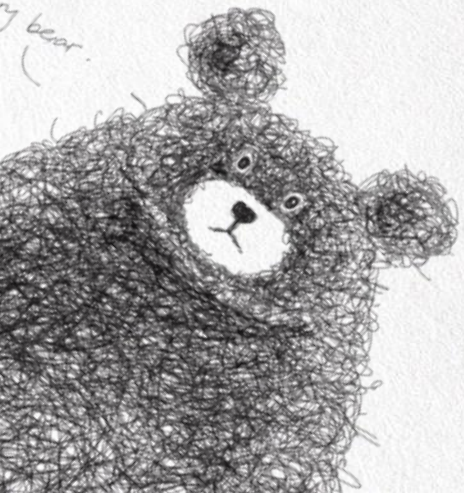



```cpp
int x;
int& foo = x;

foo = 42;
std::cout << x << std::endl;
```

Default parameters

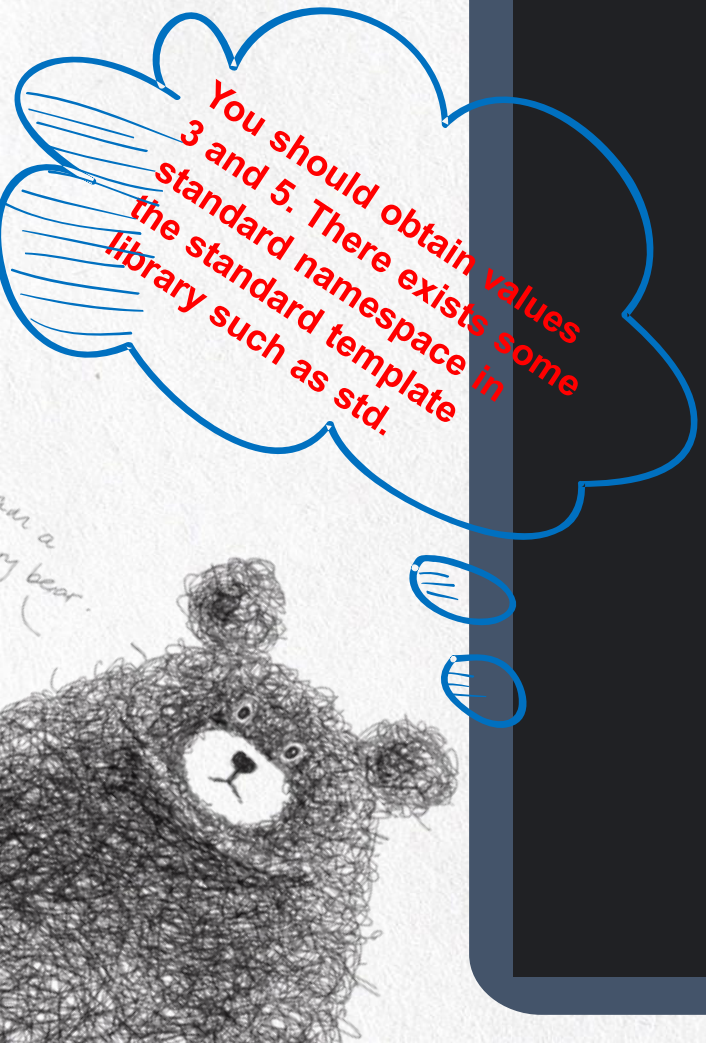
You can specify default values for function parameters. When the function is called with fewer parameters, default values are used.

```
float foo( float a=0, float b=1, float c=2 )
{
return a+b+c;
}

cout << foo(1) << endl
     << foo(1,2) << endl
     << foo(1,2,3) << endl;
```



Namespaces

Namespace allows to group classes, functions and variable under a common scope name that can be referenced elsewhere.




```
namespace first  { int var = 5; }
namespace second { int var = 3; }
cout << first::var << endl << second::var << endl;
```

Overloading

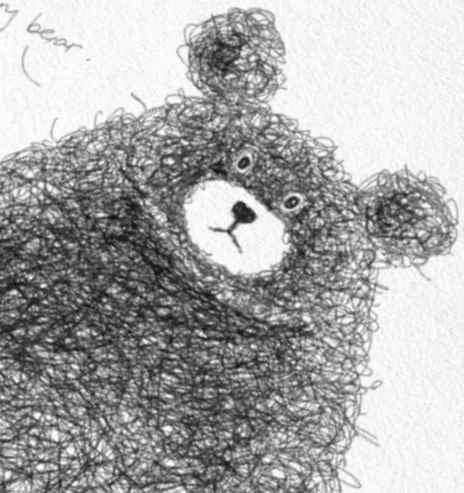
Function overloading refers to the possibility of creating multiple functions with the same name as long as they have different parameters (type and/or number).


```
float add( float a, float b )
{
return a+b;
}

int add( int a, int b )
{
return a+b;
}
```

Const & inline

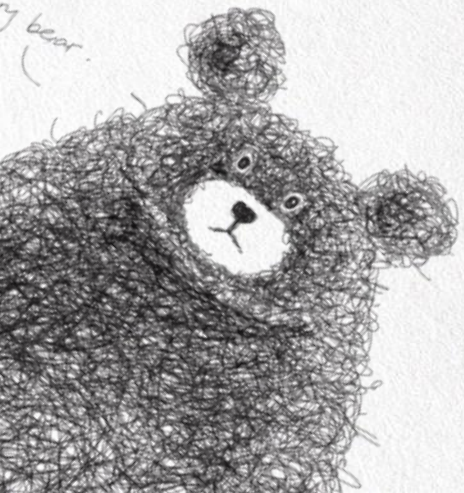
Defines and macros are bad if not used properly as illustrated (sample1)
For constants, prefer the const notation:
For macros, prefer the inline notation:

```
#define SQUARE(x) x*x

int result = SQUARE(3+3);

const int two = 2;

int inline square(int x)
{
    return x*x;
}
```

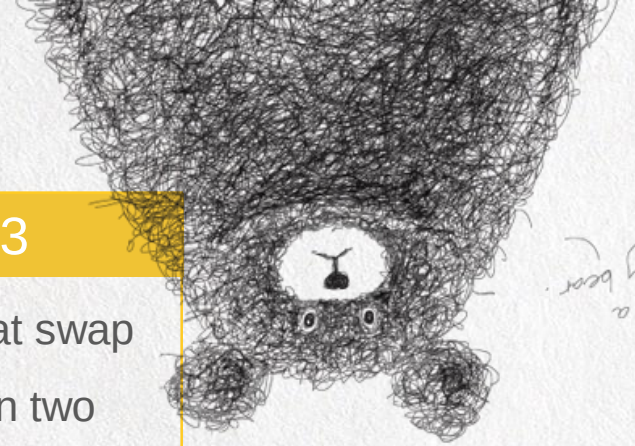


#ifdef #define #undef

(#ifdef, #ifndef, #if, #endif, #else and #elif)
These directives allow to include or discard part of the code of a program if a certain condition is met.



```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];

#pragma once
```



Exercises 1

Write a basic makefile for compiling sources ☺

Exercises 3

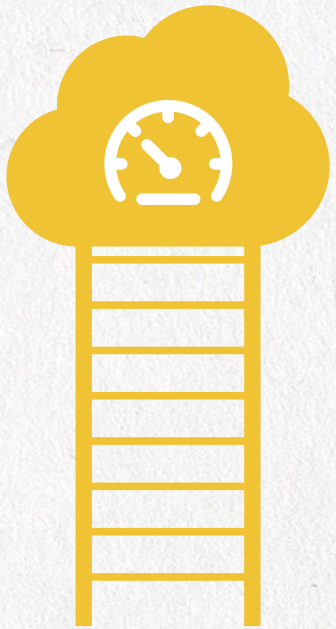
Write a function that swap two integers, then two pointers.

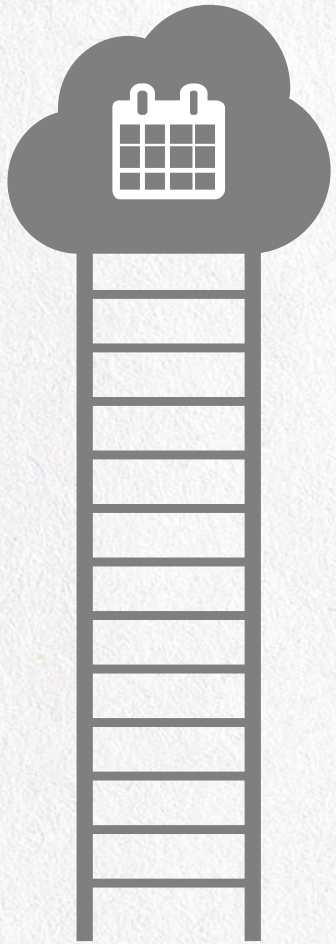
Exercises 2

Create a two-dimensional array of integers (size is n x n), fill it with corresponding indices (a[i][j] = i*n+j), test it and finally, delete it.

Exercises 4

What's the difference between int const* p, int* const p and int const* const p ?

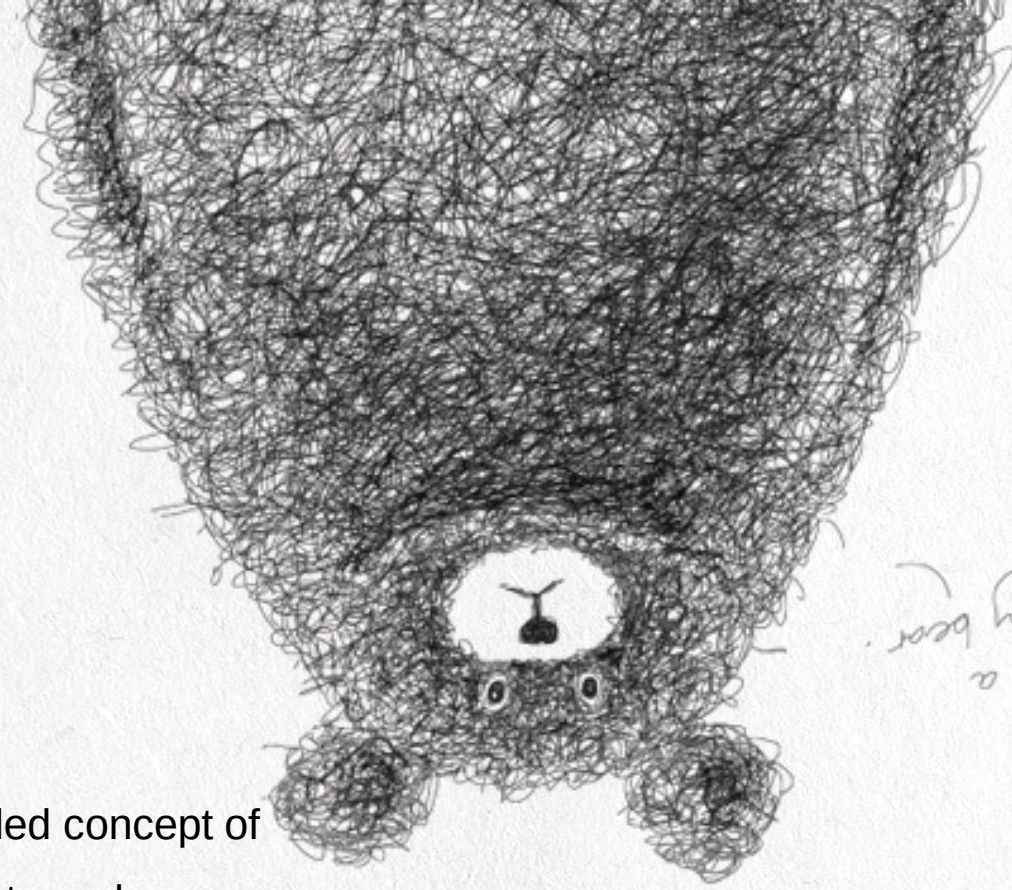




Exercises 5

Is this legal ?

```
int add( int a, int b ) { return a+b; }

int add( int a, int b, int c=0 ) { return a+b+c; }
```



PART 02

Classes

Even if C++ is slanted A class might be considered as an extended concept of a data structure: instead of holding only data, it can hold both data and functions. An object is an instantiation of a class. By default, all attributes and functions of a class are private (see below Access control). If you want a public default behavior, you can use keyword struct instead of keyword class in the declaration. object-oriented programming (OOP), you can nevertheless use any c++ compiler to compile c code and benefits from some c++ goodies.

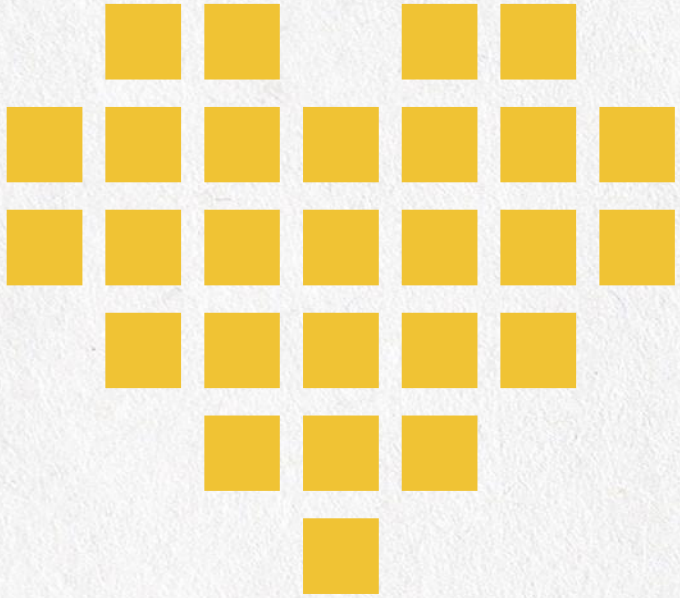
Why class

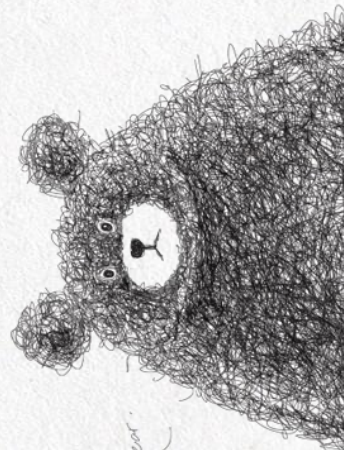
01 A class is a mechanism of binding data members and associating methods in a single unit.

02 A class can help make code cleaner, easier to use, more robust, and in a rare while, more flexible and extendible.

03 Classes are used an abstraction to real world entity.

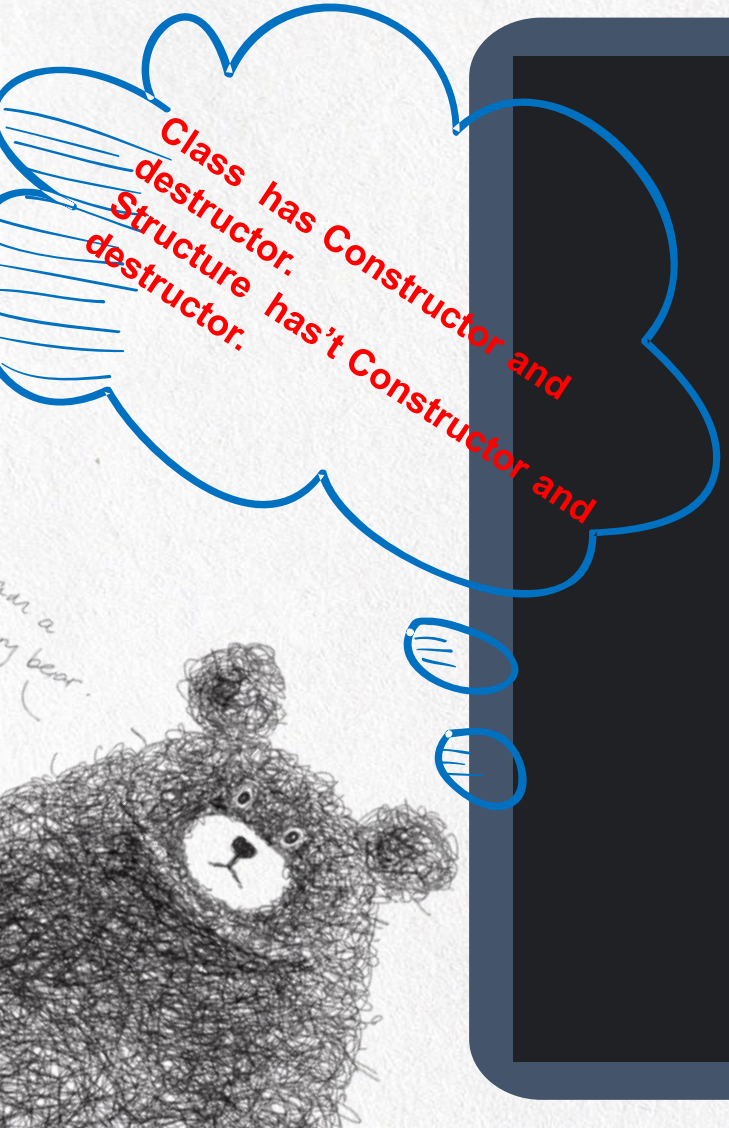
Note : An Object is an instance of a Class.





structure vs class

Members of a class are private by default and members of a struct are public by default.A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details

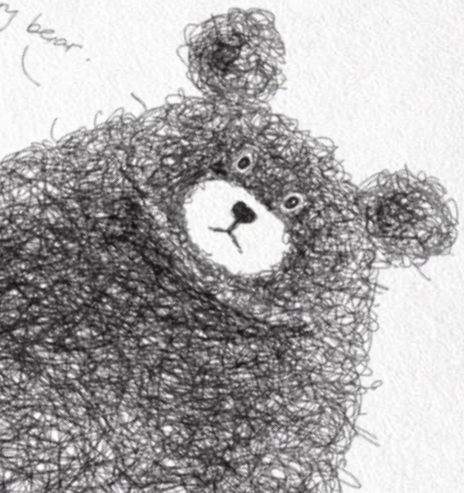



```
class Foo {
    int attribute;
    int function( void ) { };
};

struct Bar {
    int attribute;
    int function( void ) { };
};

Foo foo;
foo.attribute = 1; // WRONG

Bar bar;
bar.attribute = 1;  // OK
```

Constructors

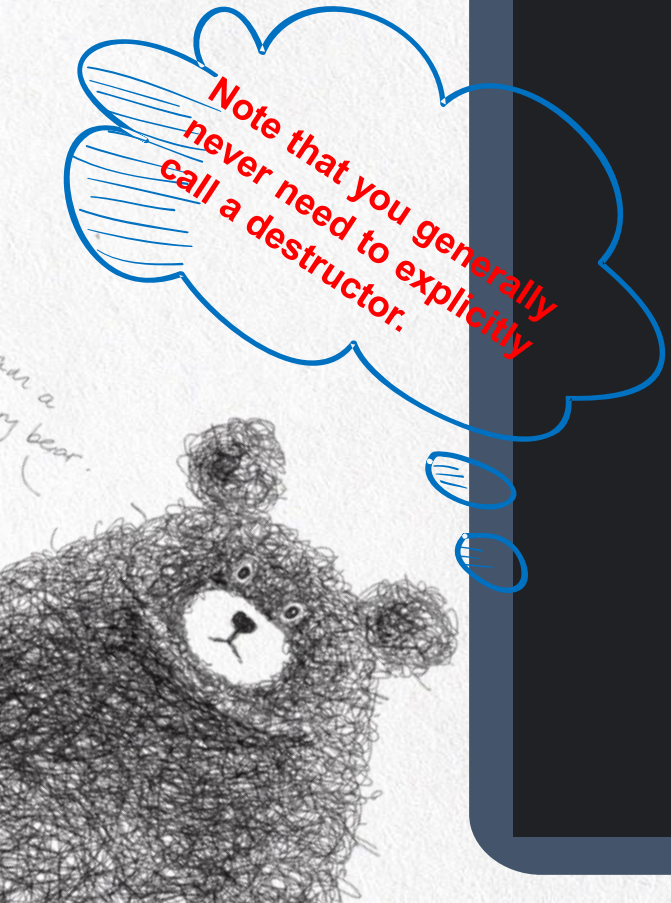
It is possible to specify zero, one or more constructors for the class.

```cpp
#include <iostream>

class Foo {
public:
    Foo( void )
    { std::cout << "Foo constructor 1 called" << std::endl; }
    Foo( int value )
    { std::cout << "Foo constructor 2 called" << std::endl; }
};

int main( int argc, char **argv )
{
    Foo foo_1, foo_2(2);
    return 0;
}
```



Destructor

There can be only one destructor per class. It takes no argument and returns nothing.




```cpp
#include <iostream>

class Foo {
public:
    ~Foo( void )
    { std::cout << "Foo destructor called" << std::endl; }
}
int main( int argc, char **argv )
{
    Foo foo();
    return 0;
}
```

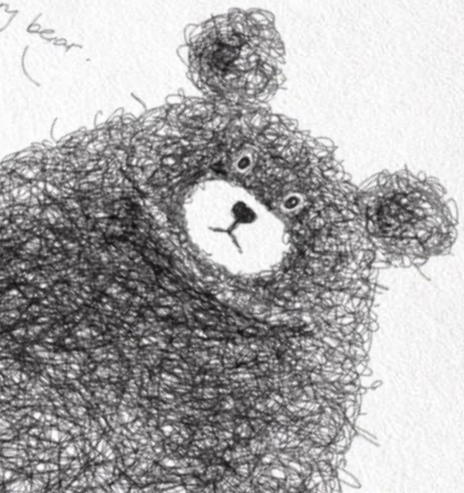
Access control

You can have fine control over who is granted access to a class function or attribute by specifying an explicit access policy:

public: Anyone is granted access

protected: Only derived classes are granted access

private: No one but friends are granted access



Initialization list

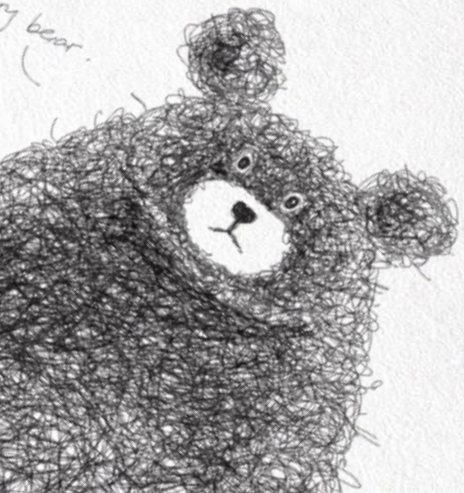
Object's member should be initialized using initialization lists


```
class Foo
{
    int _value;
public:
    Foo(int value=0) : _value(value) { };
};
```

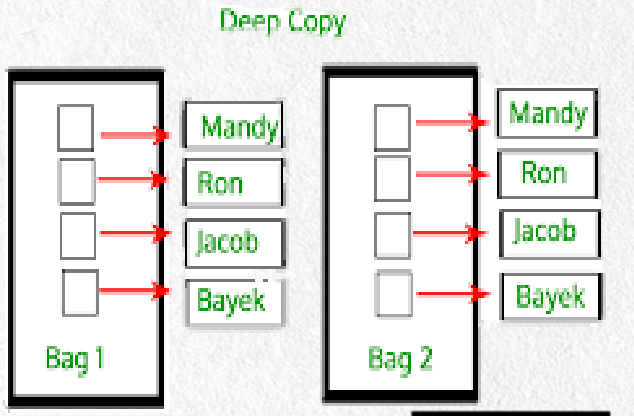
Operator overloading

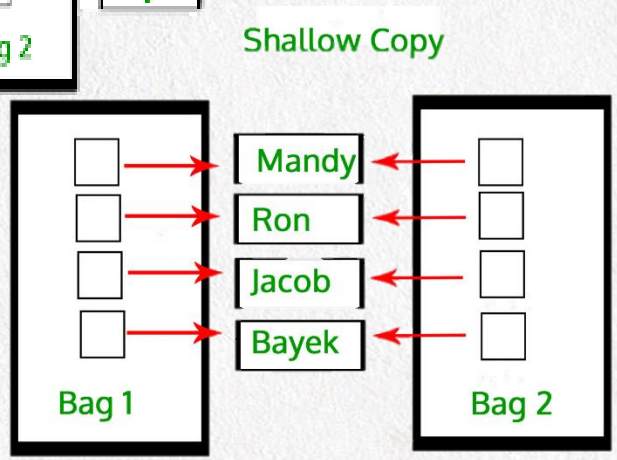
Customizes the C++ operators for operands of user-defined types.

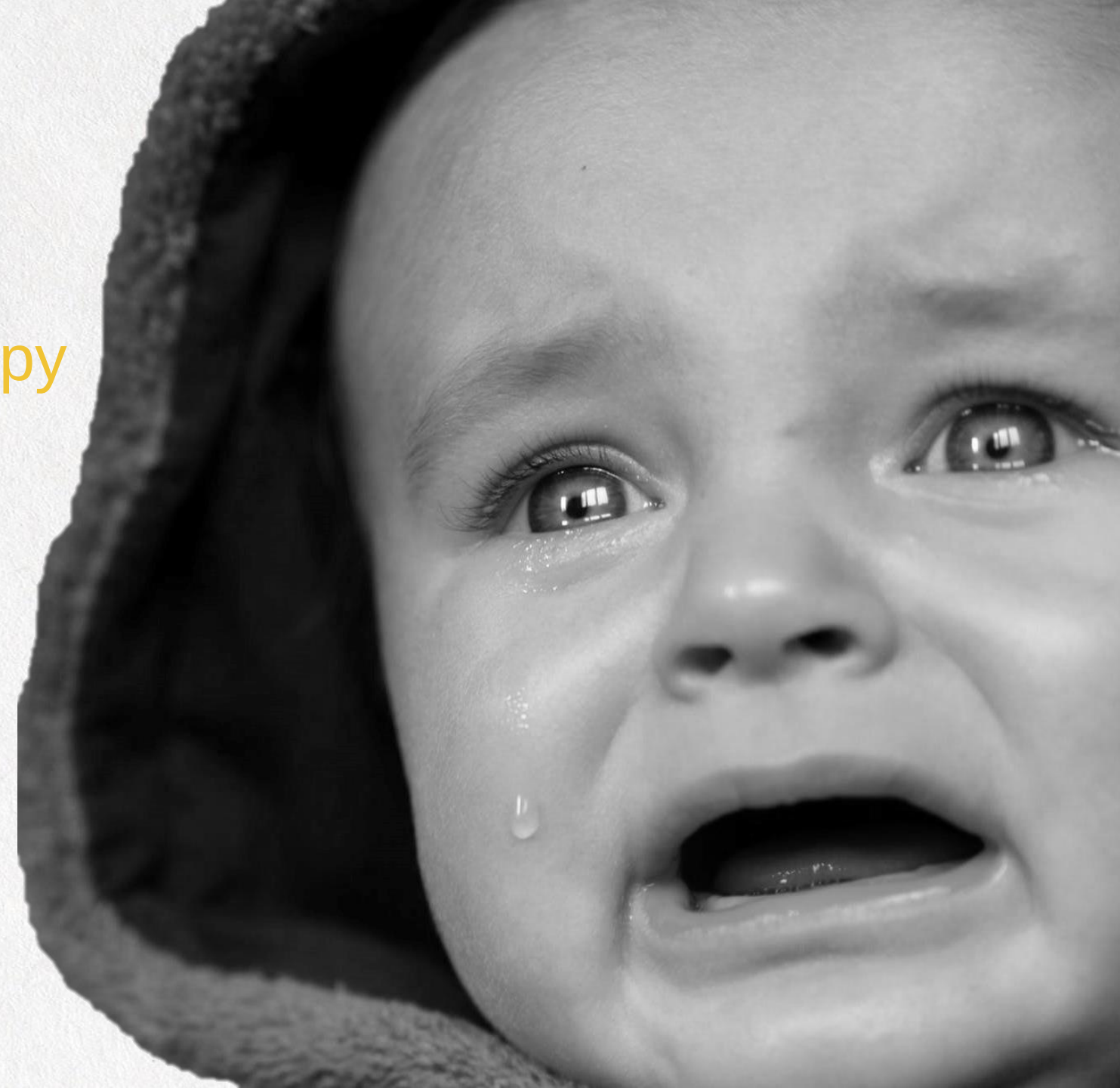
```
class Foo {
private:
    int _value;

public:
    Foo( int value ) : _value(value) { };

    Foo operator+ ( const Foo & other )
    {
        return Foo( _value+ other._value );
    }

    Foo operator* ( const Foo & other );
    {
        return Foo( _value * other._value );
    }
}
```



Shadow Copy vs Deep Copy and memory leak!!!

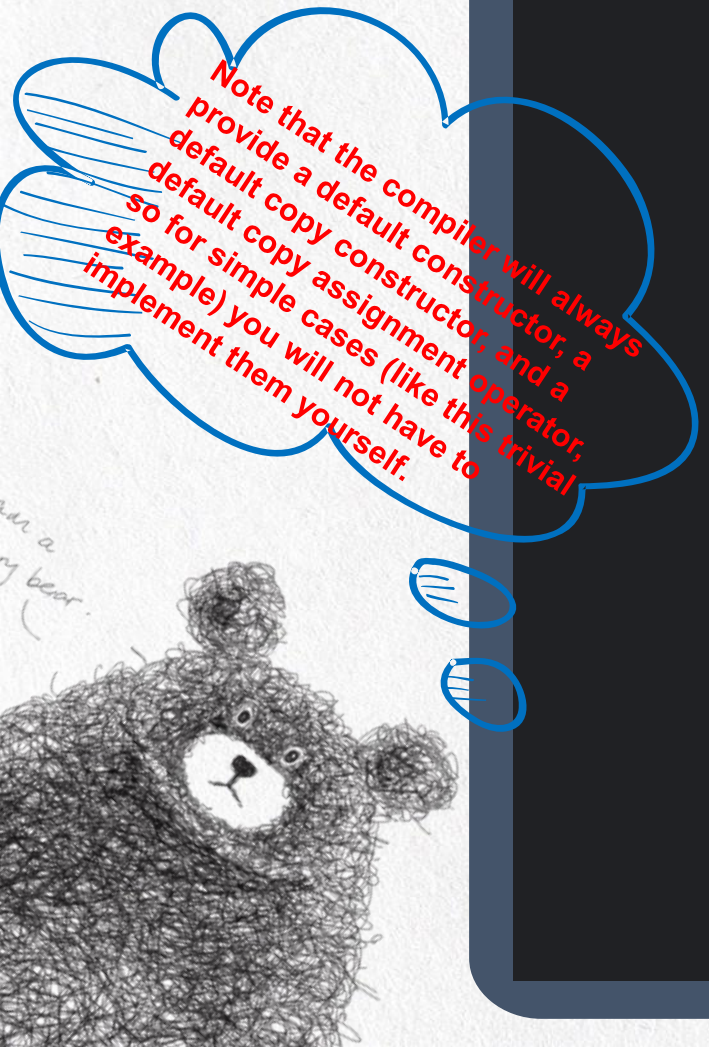






Copy Constructor and Copy Assignment

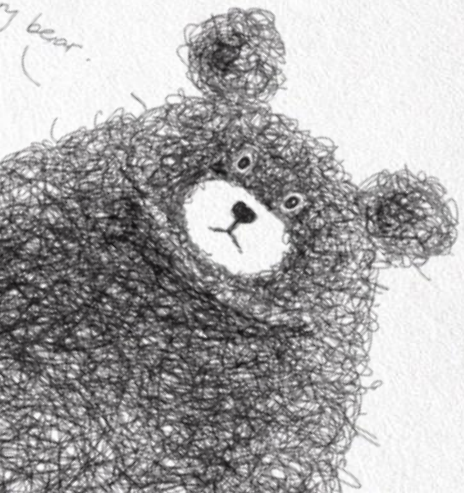
Copy constructors and copy assigment operators allow one object to be constructed or assigned a copy of another object directly:




```
Foo a(10);
Foo b(a);   // (1): Copy via constructor
Foo c = a;    // (2): Copy via assignment operator
class Foo {
private:
    int data;
public:
    // Default (no argument) constructor
    Foo() : data(0) {}
    // Single argument constructor
    explicit Foo(const int v) : data(v) {}
    // Copy constructor
    Foo(const Foo & f) : data(f.data) {}
    // Copy assignment operator
    Foo & operator=(const Foo & f) {
        data = f.data;
        return *this;
    }
};
```

Move Constructor and Move Assignment

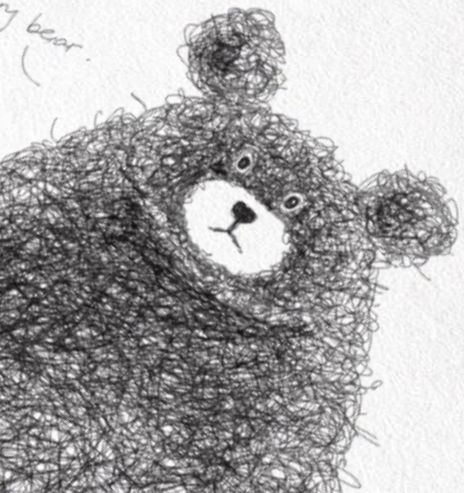
Sometimes instead of performing a copy you instead wish to completely move data from one object to another. This requires the use of a move constructor and move assignement operator.



```cpp
class Movable {
private:
    Foo * data_ptr;
public:
    Movable(Foo data) : data_ptr(new Foo(data)) {}
    // Move constructor
    Movable(Movable && m) {
        // Point to the other object's data
        data_ptr = m.data_ptr;

        // Remove the other object's data pointer by
        // setting it to nullptr
        m.data_ptr = nullptr;
    }
    // Move assignment operator
    Movable & operator=(Movable && m) {
        data_ptr = m.data_ptr;
        m.data_ptr = nullptr;
        return *this;
    }
    ~Movable() {
        delete data_ptr;
    }
};
int main() {
    Movable a(Bar());       // Using the move constructor
    Movable b = Bar();        // Using the move assignment operator
}
```

Friends

Friends are either functions or other classes that are granted privileged access to a class.



```cpp
#include <iostream>

class Foo {
public:
    friend std::ostream& operator<< ( std::ostream& output,
                                      Foo const & that )
    {
        return output << that._value;
    }
private:
    double _value;
};

int main( int argc, char **argv )
{
  Foo foo;
  std::cout << "Foo object: " << foo << std::endl;
  return 0
}
```

Exercises 1

Write a Point class that can be constructed using cartesian or polar coordinates.

Exercises 3

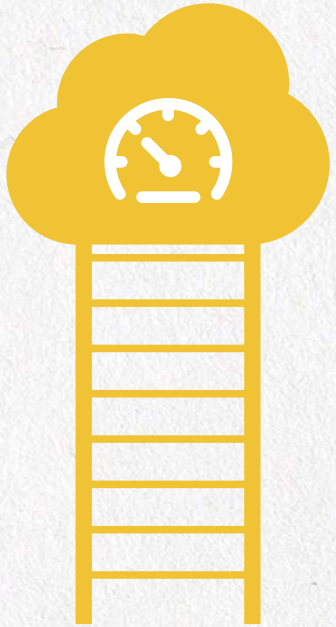
Write a Foo class and provide it with an input method.

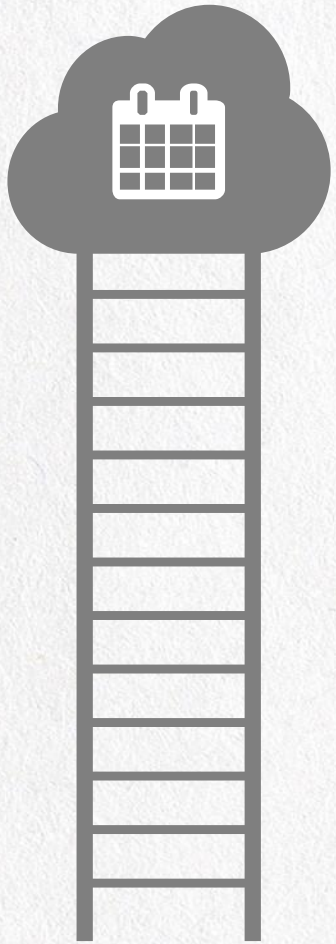
Exercises 2

Write a Foo class with default and copy constructors and add also an assignment operator. Write some code to highlight the use of each of them.

Exercises 4

Is is possible to write something like foo.method1().method2() ?





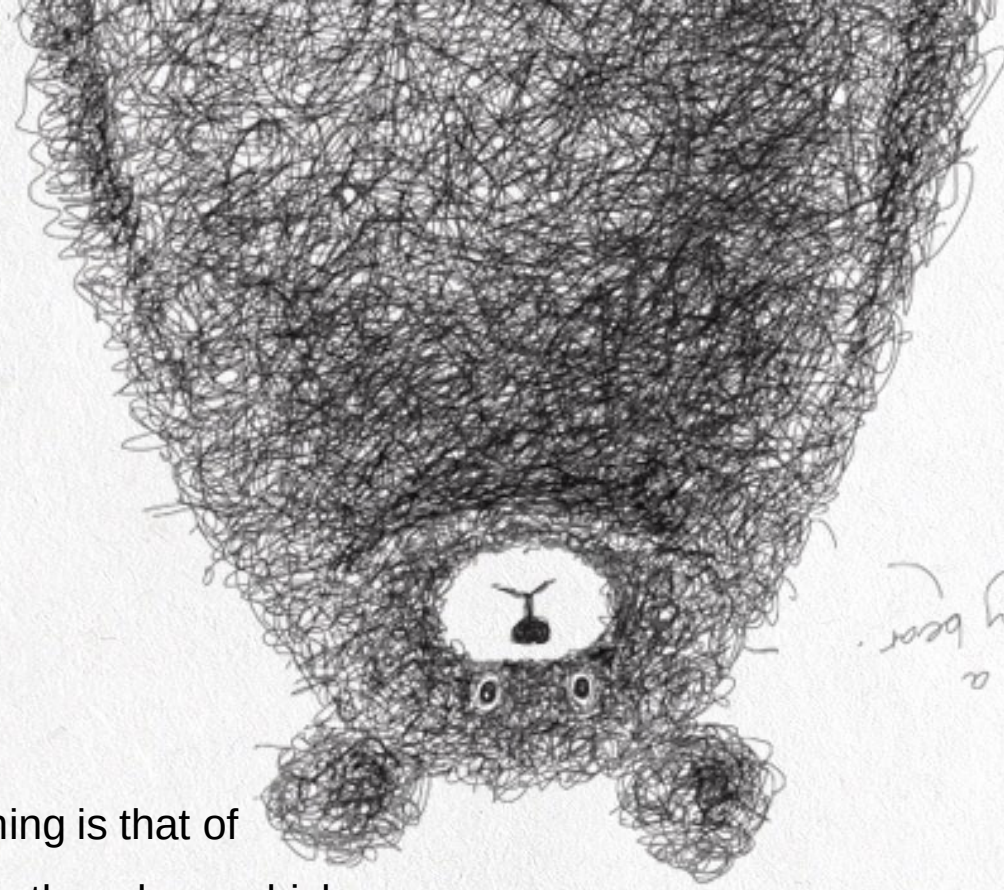
Exercises 5

Why the following code doesn't compile ?

```
class Foo { Foo () { }; };
int main( int argc, char **argv )
{
    Foo foo;
}
```

PART 03

Inheritance



One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

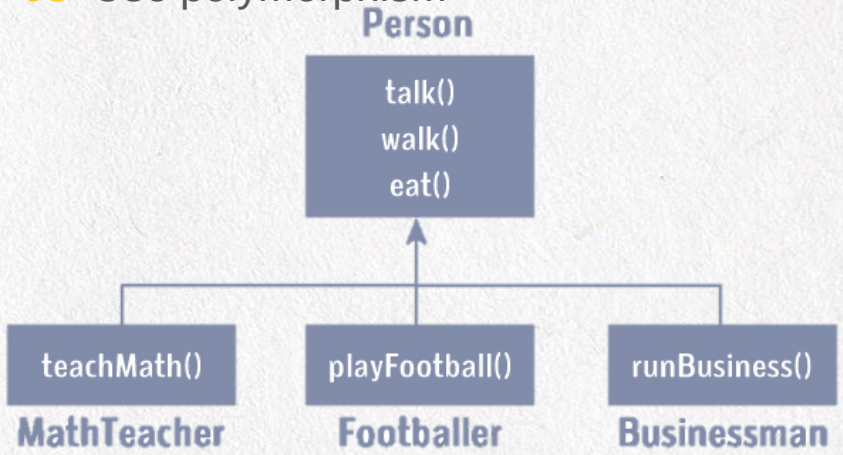
Why Inheritance

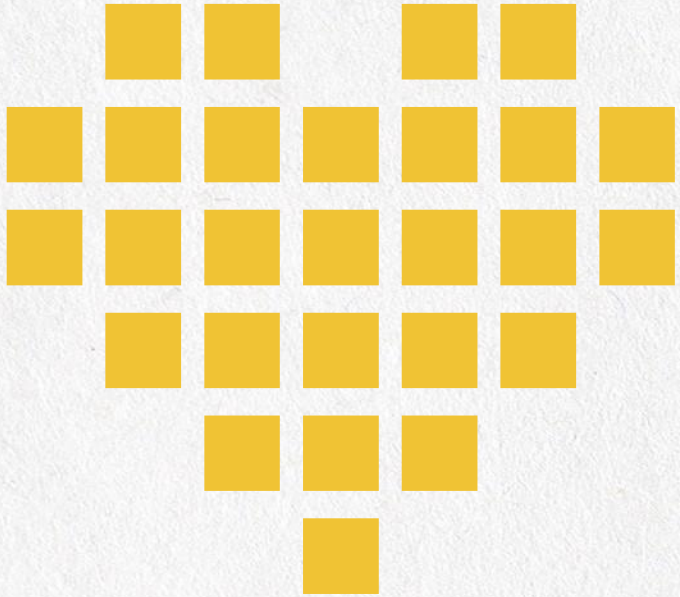
01 a specific section of your project's code reusable with the possibility of adding or removing certain features later.

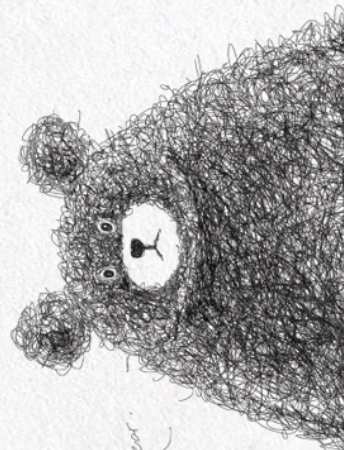
02 A class can help make code cleaner, easier to use, more robust, and in a rare while, more flexible and extendible.

reduce number of copy

03 Use polymorphism





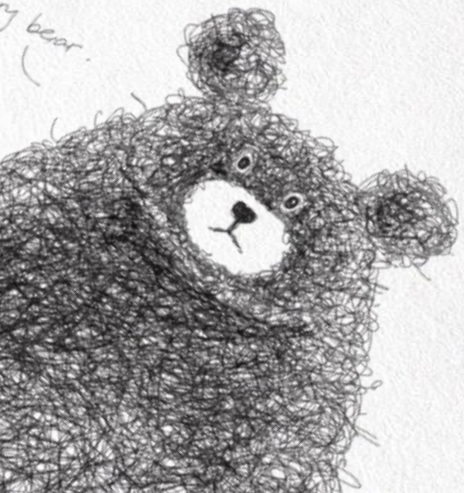


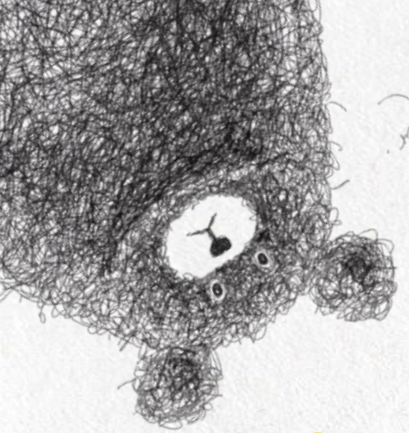
Basics

It is possible to specify zero, one or more constructors for the class.
Bar_public, Bar_private and Bar_protected are derived from Foo. Foo is the base class of Bar_public, Bar_private and Bar_protected.

```
class Foo                           { /* ... */ };
class Bar_public : public Foo       { /* ... */ };
class Bar_private : private Foo     { /* ... */ };
class Bar_protected : protected Foo { /* ... */ };
```

* In Bar_public , public parts of Foo are public, protected parts of Foo are protected
* In Bar_private , public and protected parts of Foo are private
* In Bar_protected , public and protected parts of Foo are protected



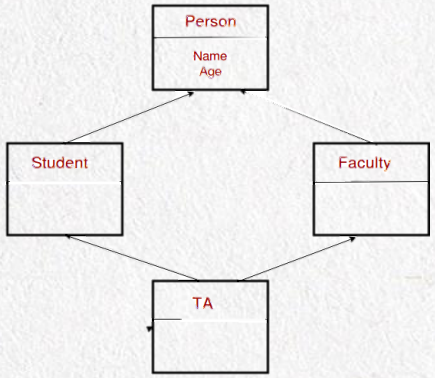


Order of call

C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in reverse order.

Multiple Inheritance Problem

The diamond problem occurs when two superclasses of a class have a common base class



Virtual methods

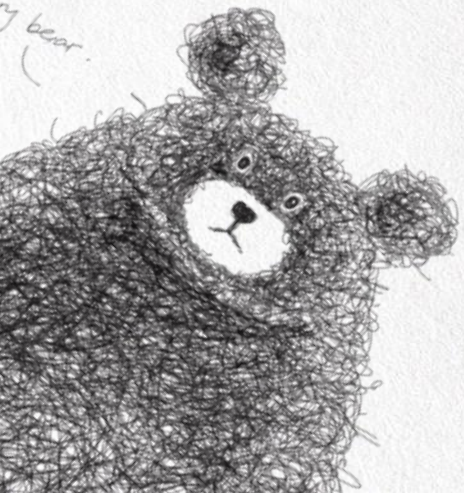
A virtual function allows derived classes to replace the implementation provided by the base class (yes, it is not automatic...). Non virtual methods are resolved statically (at compile time) while virtual methods are resolved dynamically (at run time).


```
class Foo {
public:
    Foo( void );
    void method1( void );
    virtual void method2( void );
};

class Bar : public Foo {
public:
    Bar( void );
    void method1( void );
    void method2( void );
};

Foo *bar = new Bar();
bar->method1();
bar->method2();
```

Multiple inheritance

A class may inherit from multiple base classes but you have to be careful:
In class Bar3, the data reference is ambiguous since it could refer to Bar1::data or Bar2::data. This problem is referred as the diamond problem. You can eliminate the problem by explicitly specifying the data origin (e.g. Bar1::data) or by using virtual inheritance in Bar1 and Bar2.

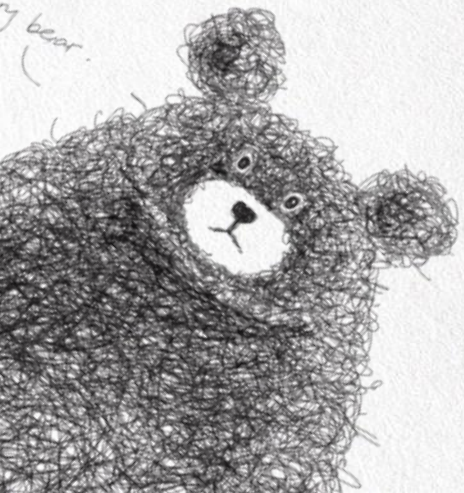
```
class Foo                  { protected: int data; };
class Bar1 : public Foo { /* ... */ };
class Bar2 : public Foo { /* ... */ };
class Bar3 : public Bar1, public Bar2 {
    void method( void )
    {
        data = 1; // !!! BAD
    }
};
```



Abstract classes

You can define pure virtual method that prohibits the base object to be instantiated. Derived classes need then to implement the virtual method.

```
class Foo {
public:
    Foo( void );
    virtual void method( void ) = 0;
};

class Bar: public Foo {
public:
    Foo( void );
    void method( void ) { };
};
```



Exercises 1

Write a foo function and make it called from a class that has a foo method.

Exercises 3

Write a Singleton class such that only one object of this class can be created.

Exercises 2

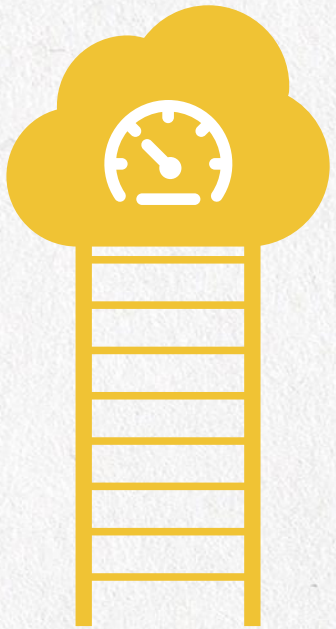
Write a Bar class that inherits from a Foo class and makes constructor and destructor methods to print something when called.

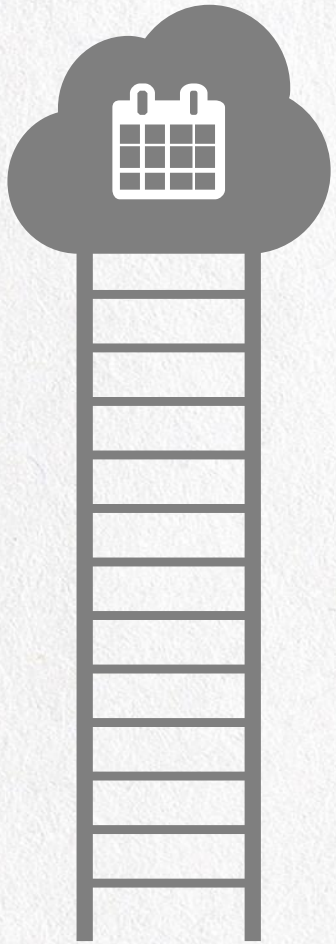
Exercises 4

Write a Real base class and a derived Integer class with all common operators (+,-,*,/)

Exercises 5

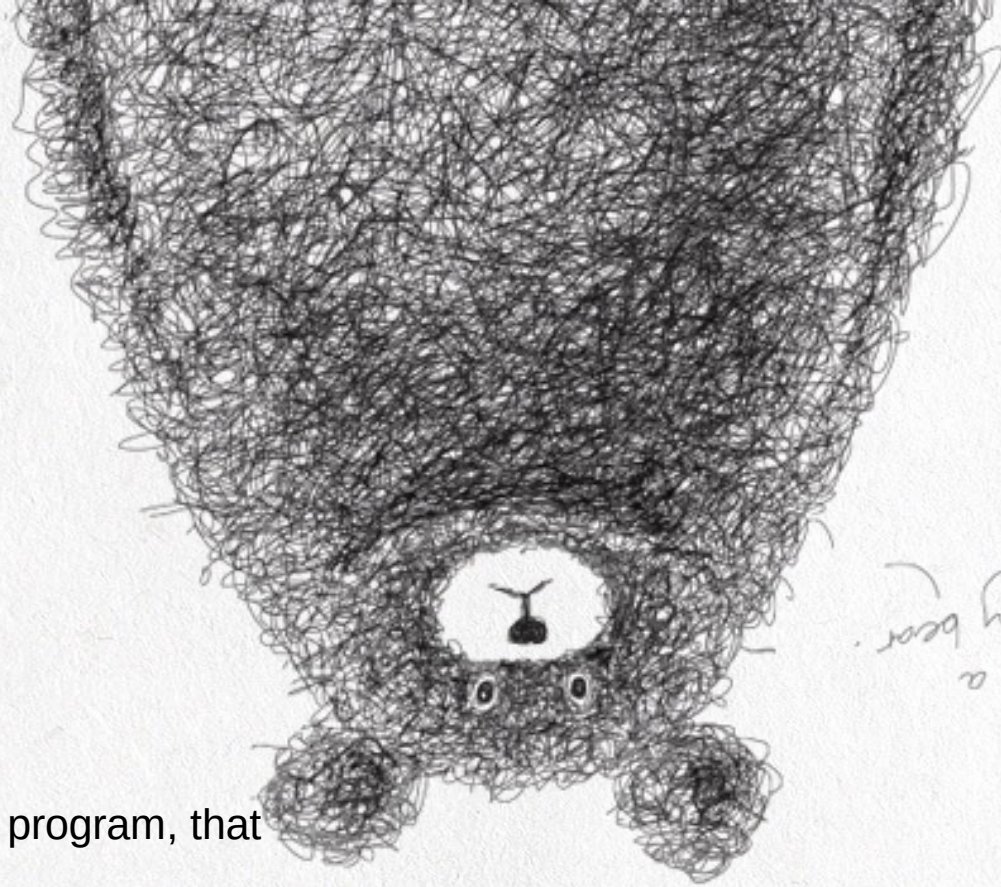
Write a functor class ☺





PART 04

Exceptions



An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions!

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

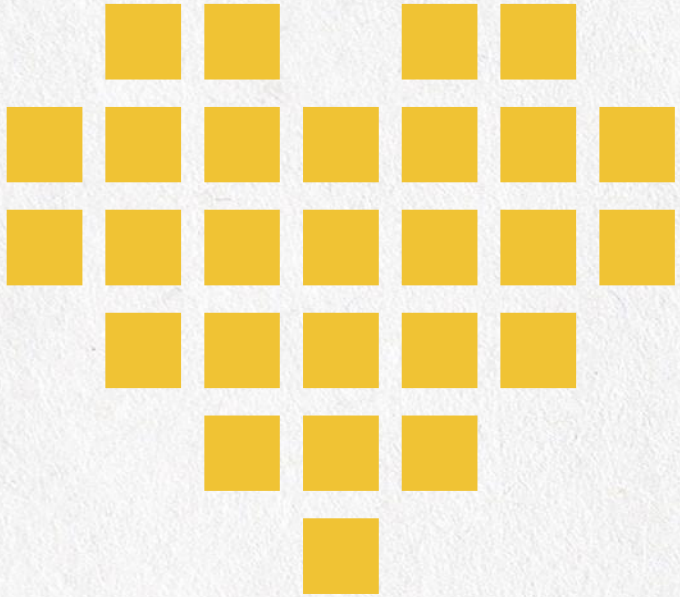
Although never is often better than right now.

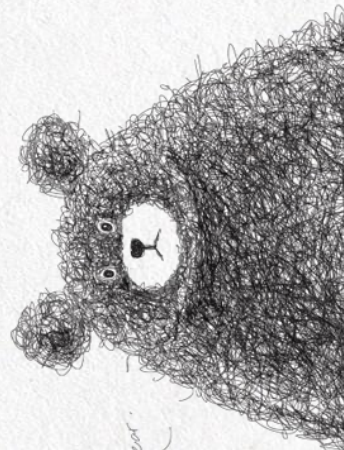
If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

(by Tim Peters)





Catch me if you can

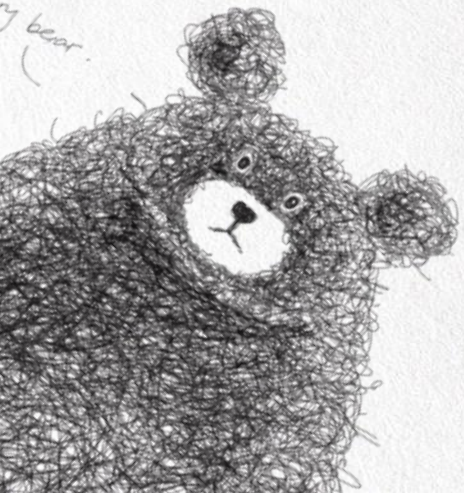
You can catch any exception using the following structure:


```
try
{
    float *array = new float[-1];
}
catch( std::bad_alloc e )
{
    std::cerr << e.what() << std::endl;
}
```

Creating your own exception

Creating a new exception is quite easy:

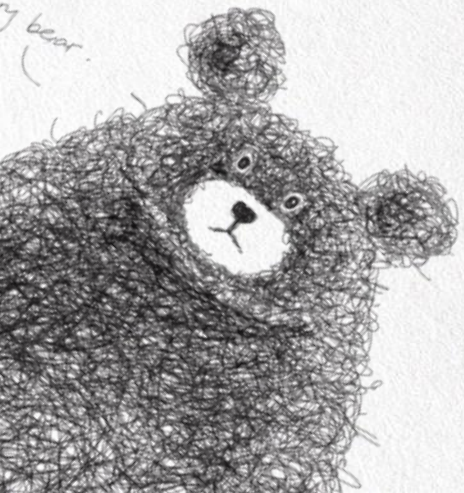
```cpp
#include <stdexcept>

class Exception : public std::runtime_error
{
public:
    Exception() : std::runtime_error("Exception") { };
};
```

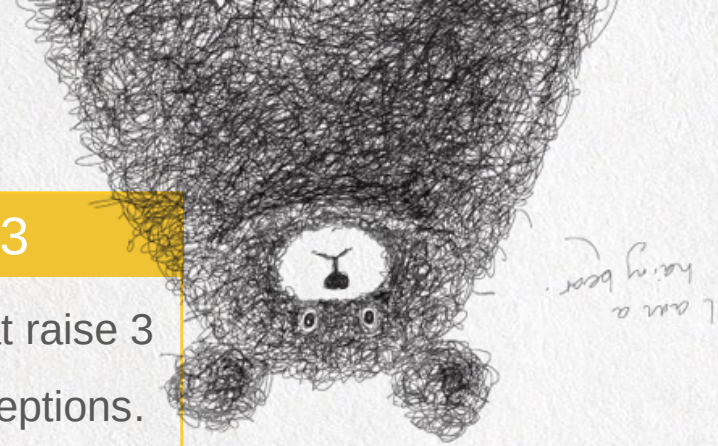


Standard exceptions

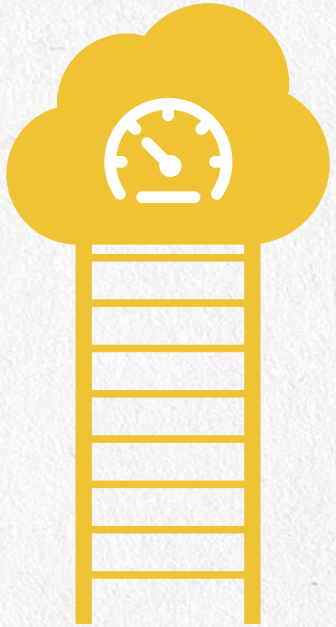
There exist some standard exceptions that can be raised in some circumstances:

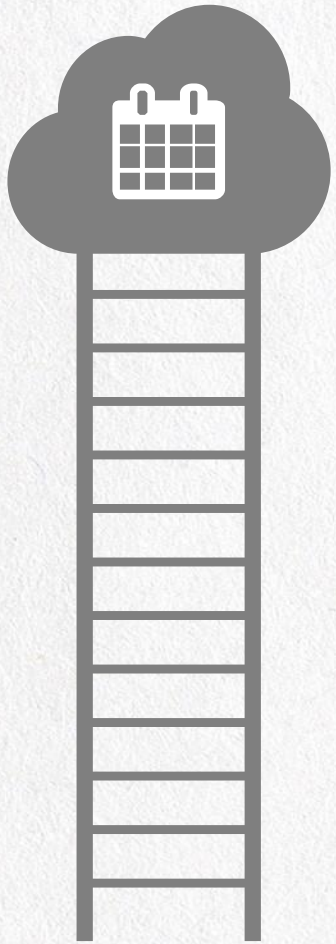
- bad_alloc
- bad_cast
- bad_exception
- bad_typeid
- logic_error
  - domain_error
  - invalid_argument
  - length_error
  - out_of_range
- runtime_error
  - range_error
  - overflow_error
  - underflow_error





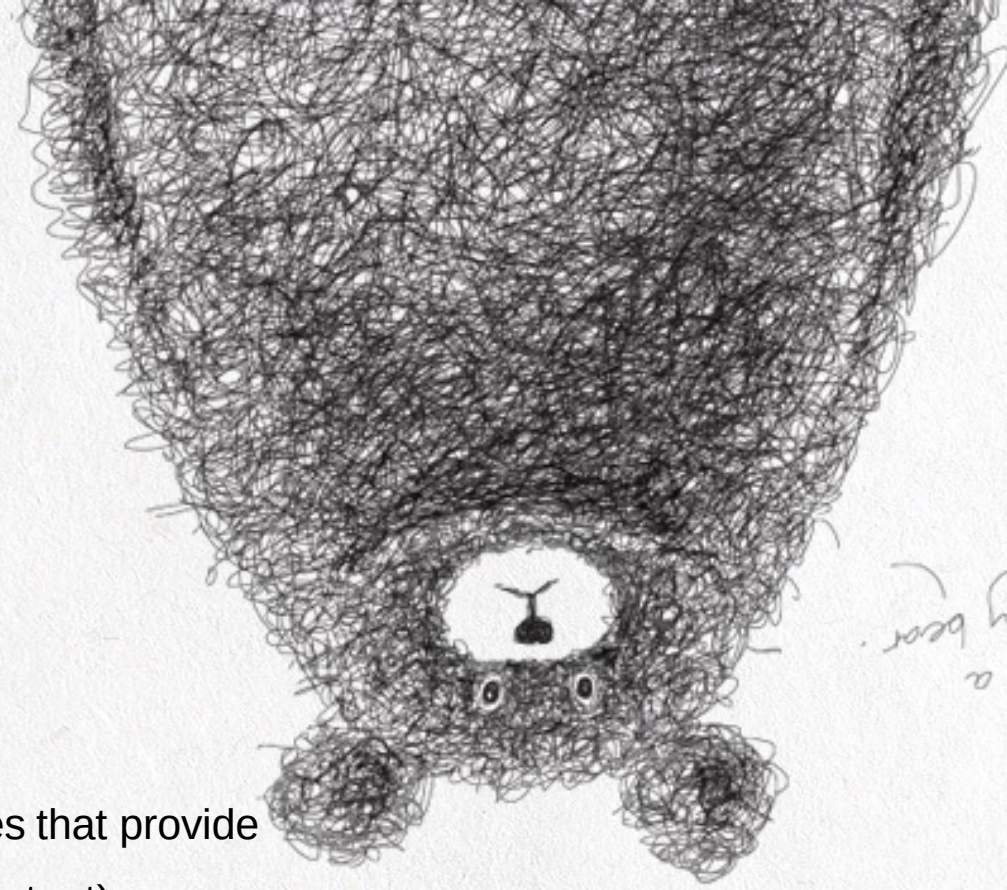
Exercises 1	Exercises 3
How to handle a constructor that fails ?	Write a program that raise 3 of the standard exceptions.
Exercises 2	**Exercises 4**
Write a correct division function.	Write a Integer (positive) class with proper exception handling (Overflow, Underflow, DivideByZero, etc.)





PART 05

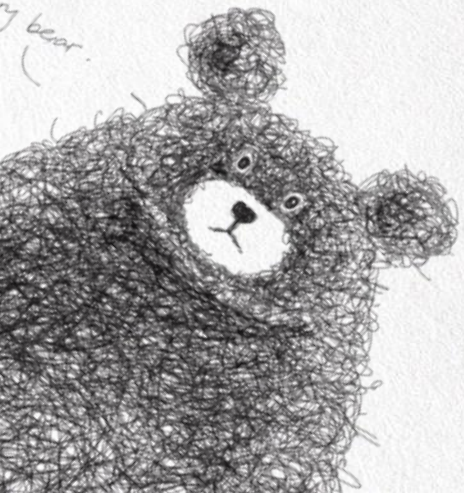
Streams



C++ provides input/output capability through the iostream classes that provide the stream concept (iXXXstream for input and oXXXstream for output).

iostream and ios

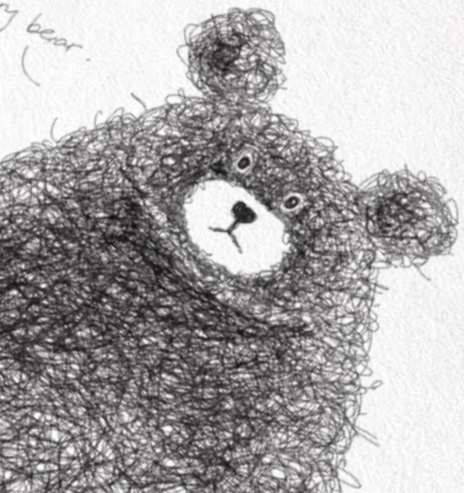
Screen outputs and keyboard inputs may be handled using the iostream header file:



```cpp
#include <iostream>

int main( int argc, char **argv )
{

    unsigned char age = 65;
    std::cout << static_cast<unsigned>(age)     << std::endl;
    std::cout << static_cast<void const*>(&age) << std::endl;

    double f = 3.14159;
    cout.unsetf(ios::floatfield);
    cout.precision(5);
    cout << f << endl;
    cout.precision(10);
    cout << f << endl;
    cout.setf(ios::fixed,ios::floatfield);
    cout << f << endl;

    std::cout << "Enter a number, or -1 to quit: ";
    int i = 0;
    while( std::cin >> i )
    {
        if (i == -1) break;
        std::cout << "You entered " << i << '\n';
    }
    return 0;
}
```

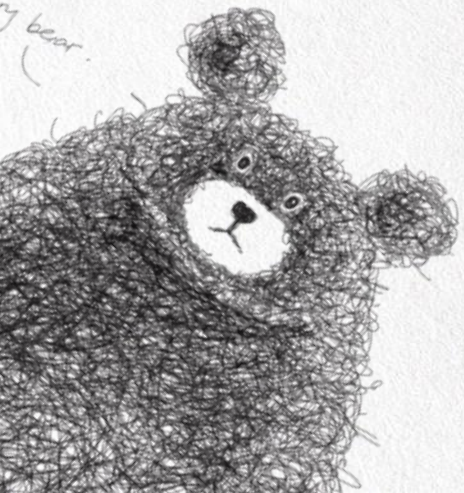
Class input/output

You can implement a class input and output using friends functions:



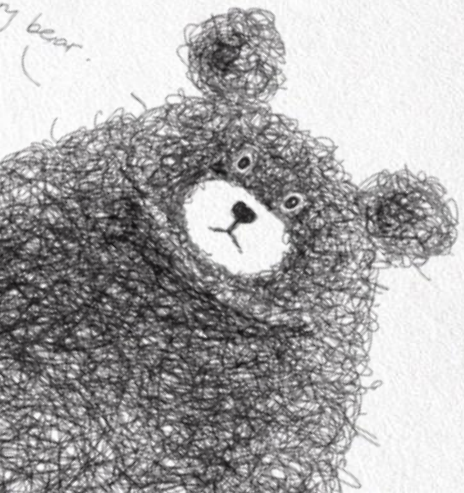
```cpp
#include <iostream>

class Foo {
public:
    friend std::ostream& operator<< ( std::ostream & output, Foo const & that )
    { return output << that._value; }
    friend std::istream& operator>> ( std::istream & input, Foo& foo )
    { return input >> fred._value; }

private:
    double _value;
};
```

Working with files

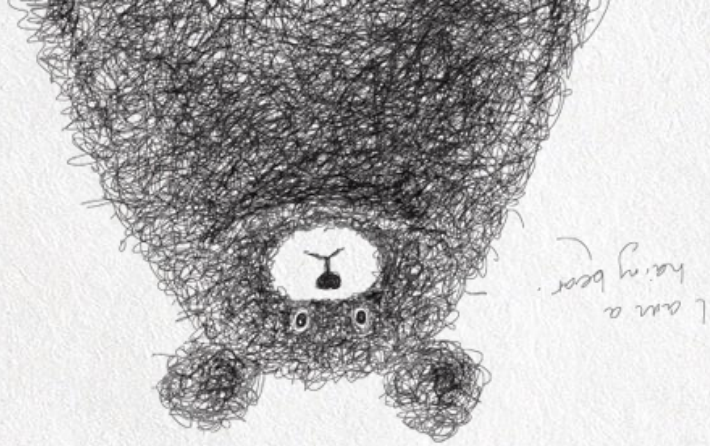


```cpp
#include <fstream>

int main( int argc, char **argv )
{
    std::ifstream input( filename );
    // std::ifstream input( filename, std::ios::in | std::ios::binary);

    std::ofstream output( filename );
    // std::ofstream output( filename, std::ios::out | std::ios::binary);

    return 0;
}
```

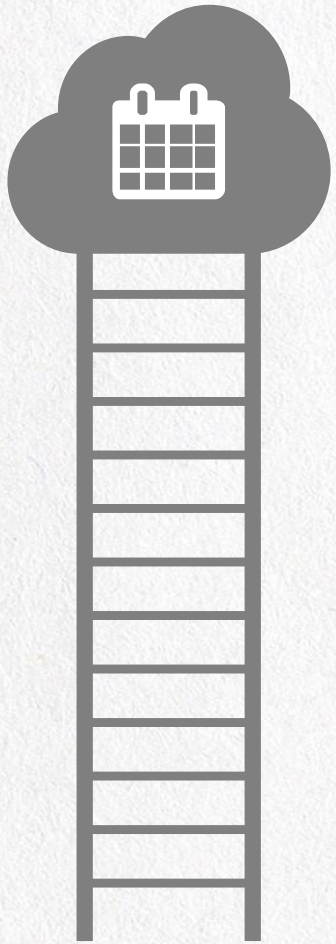
Working with strings



```cpp
#include <sstream>

int main( int argc, char **argv )
{
    const char *svalue = "42.0";
    int ivalue;
    std::istringstream istream;
    std::ostringstream ostream;

    istream.str(svalue);
    istream >> ivalue;
    std::cout << svalue << " = " << ivalue << std::endl;

    ostream.clear();
    ostream << ivalue;
    std::cout << ivalue << " = " << ostream.str() << std::endl;

    return 0;
}
```



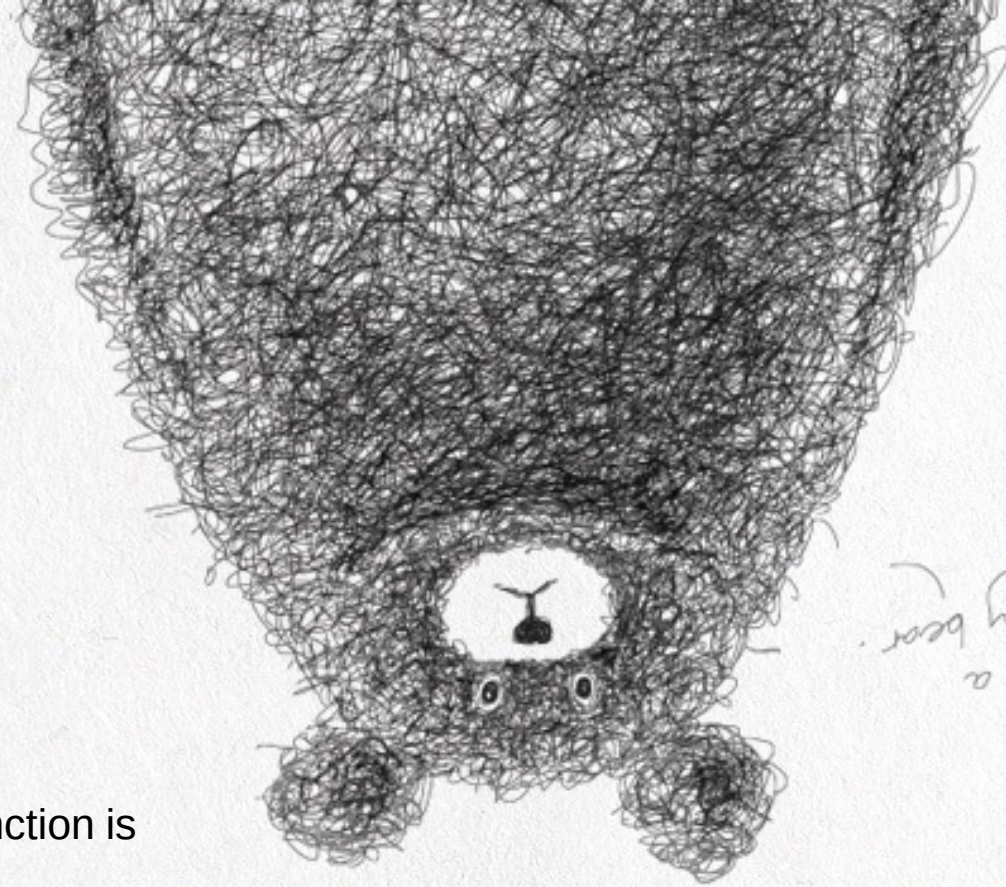
Exercises 1	Exercises 2
Write an itoa and an atoi function	Write a foo class with some attributes and write functions for writing to file and reading from file.



PART 06

Templates

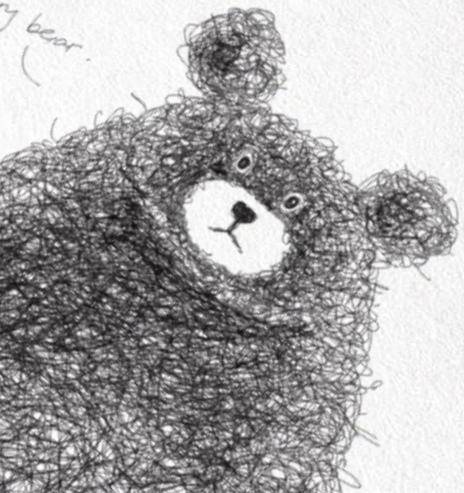
Templates are special operators that specify that a class or a function is written for one or several generic types that are not yet known.



Basic templetes

You can have several templates and to actually use a class or function template, you have to specify all unknown types:

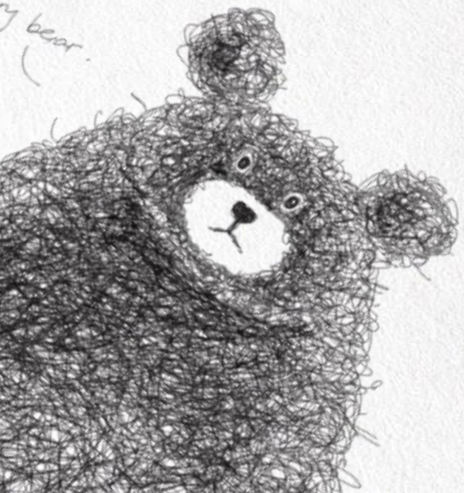
```
template<typename T1>
T1 foo1( void ) { /* ... */ };

template<typename T1, typename T2>
T1 foo2( void ) { /* ... */ };

template<typename T1>
class Foo3 { /* ... */ };


int a = foo1<int>();
float b = foo2<int,float>();
Foo<int> c;
```



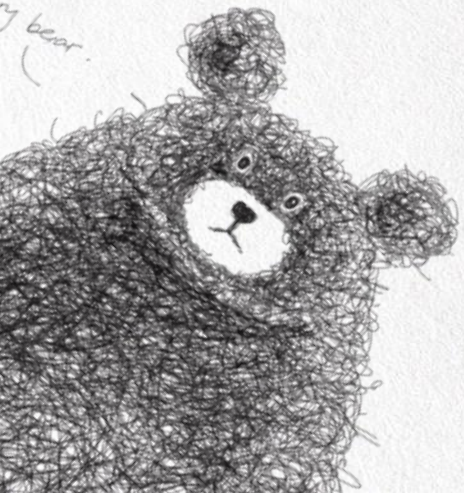
Template parameters

There are three possible template types:

```
//Type
template<typename T>  T foo( void ) { /* ... */ };
//Non-type
template<int N>  foo( void ) { /* ... */ };
//Template
template< template <typename T> > foo( void ) { /* ... */ };
```

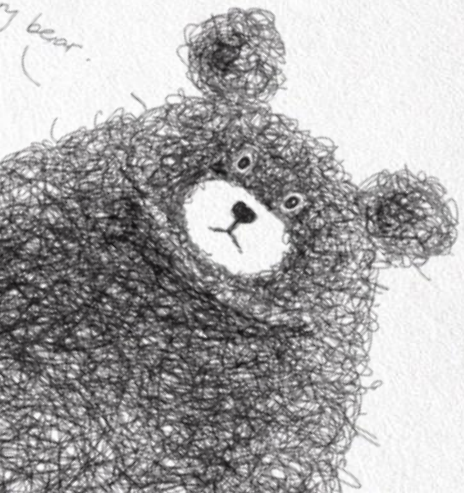


Template function

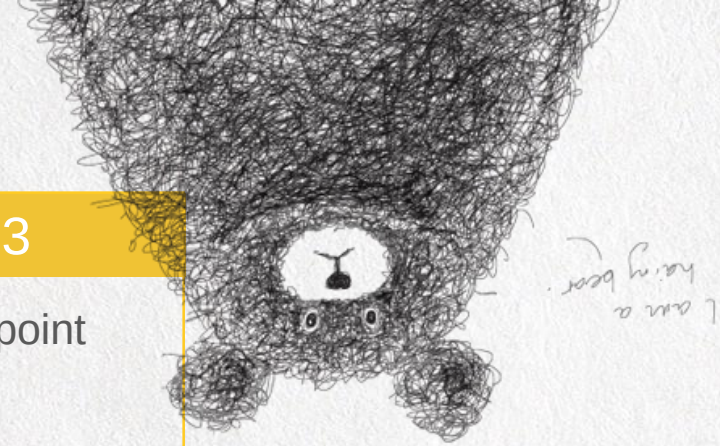


```cpp
template <class T>
T max( T a, T b)
{
    return( a > b ? a : b );
}

#include <sstream>

int main( int argc, char **argv )
{
    std::cout << max<int>( 2.2, 2.5 ) << std::endl;
    std::cout << max<float>( 2.2, 2.5 ) << std::endl;
}
```

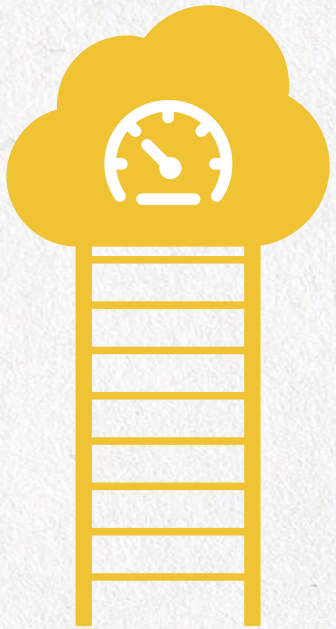
Template specialization

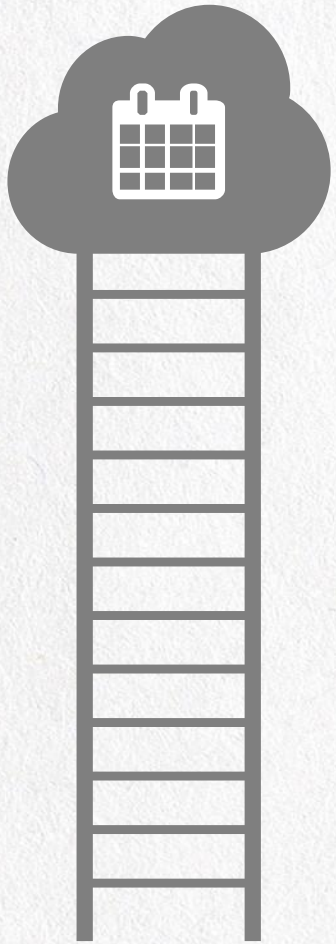


```cpp
#include <iostream>

template <class T>
class Foo {
    T _value;
public:
    Foo( T value ) : _value(value)
    {
        std::cout << "Generic constructor called" << std::endl;
    };
}

template <>
class Foo<float> {
    float _value;
public:
    Foo( float value ) : _value(value)
    {
        std::cout << "Specialized constructor called" << std::endl;
    };
}

int main( int argc, char **argv )
{
    Foo<int> foo_int;
    Foo<float> foo_float;
}
```



Exercises 1	Exercises 3
Write a generic swap function	Write a generic point structure
Exercises 2	**Exercises 4**
Write a smart pointer class	Write templated factorial, power and exponential functions (exp(x) = sum_n x^n/n!, exp(-x) = 1/exp(x))

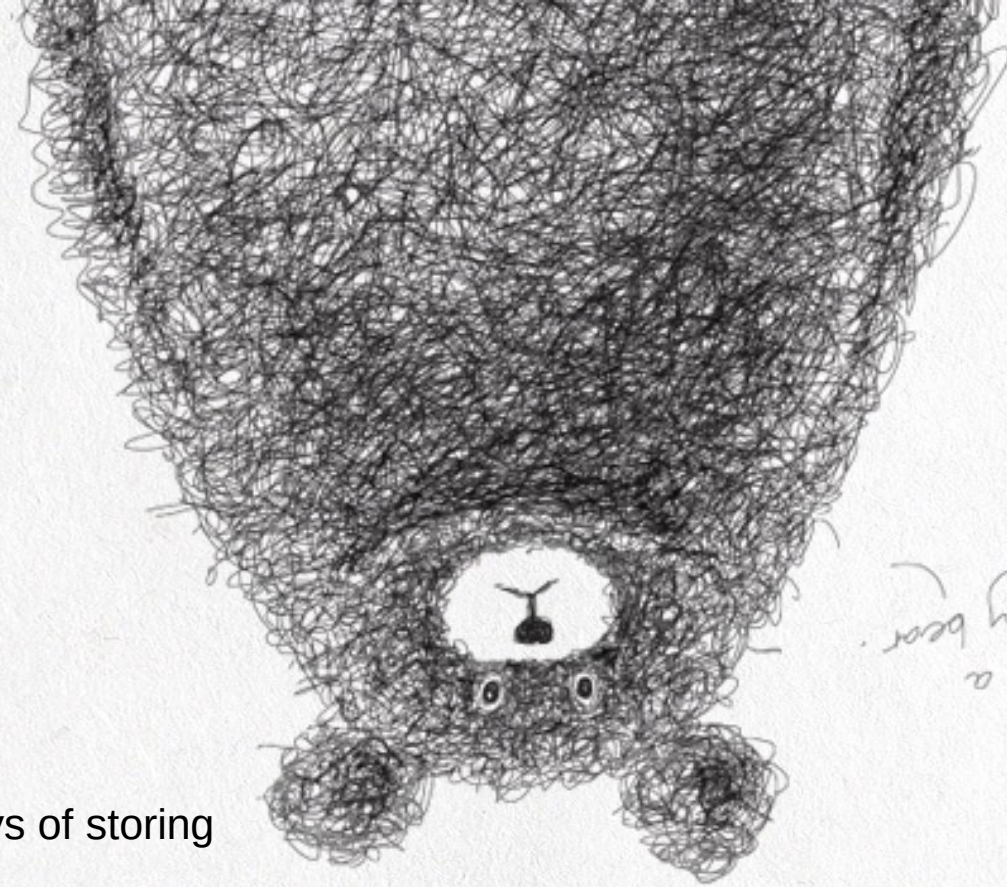




PART 07

Standard Template Library

STL containers are template classes that implement various ways of storing elements and accessing them.



Deeper view

01 Sequence containers:

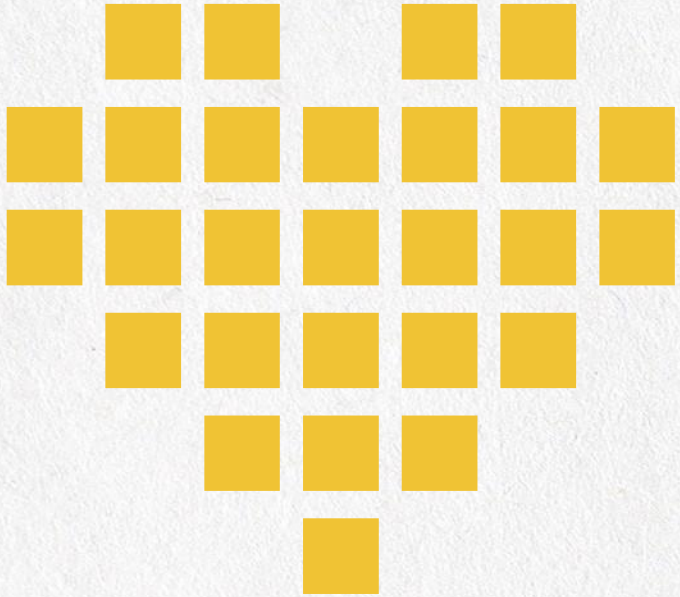
- vector
- deque
- list

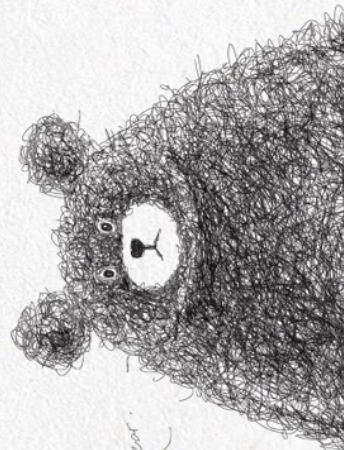
02 Container adaptors:

- stack
- queue
- priority_queue

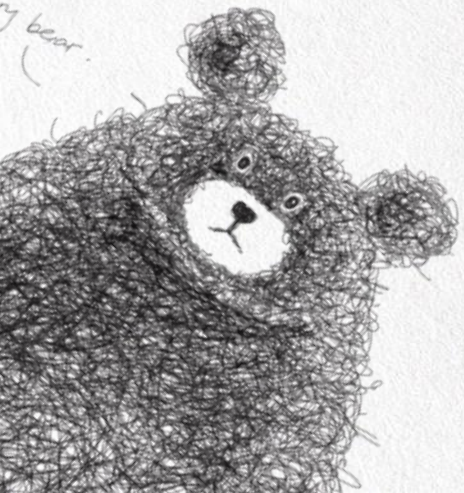
03 Associative containers:

- set
- multiset
- map
- multimap
- bitset





Sample of Vector

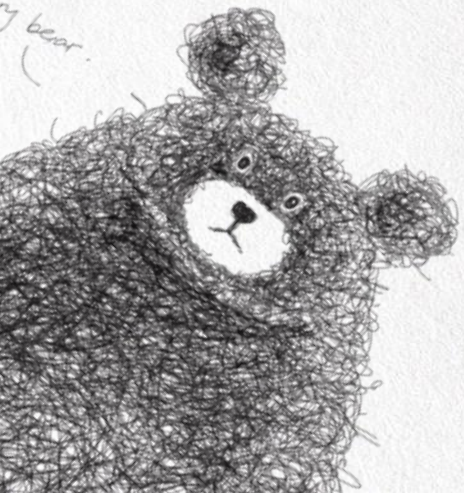


```cpp
#include <vector>
#include <map>
#include <string>

int main( int argc, char **argv )
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    std::map<std::string,int> m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    return 0;
}
```

Iterators

Iterators are a convenient tool to iterate over a container:

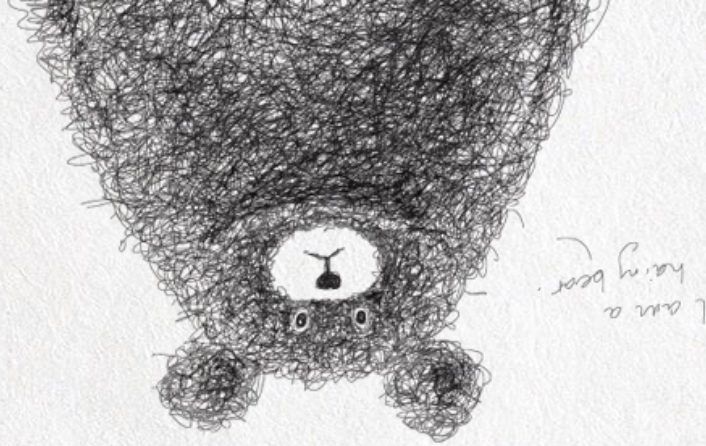
```cpp
#include <map>
#include <string>
#include <iostream>

int main( int argc, char **argv )
{
    std::map<std::string,int> m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    std::map<std::string,int>::iterator iter;
    for( iter=m.begin(); iter != m.end(); ++iter )
    {
        std::cout << "map[" << iter->first << "] = "
                  << iter->second << std::endl;
    }
    return 0;
}
```



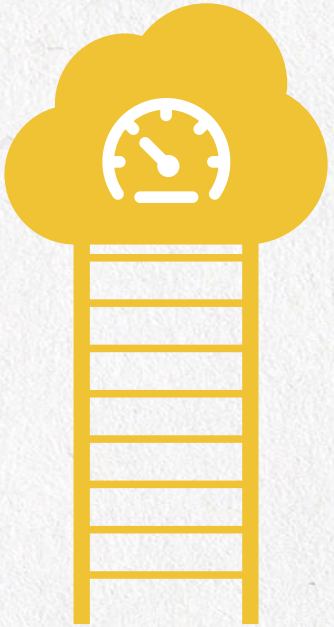
Algorithms

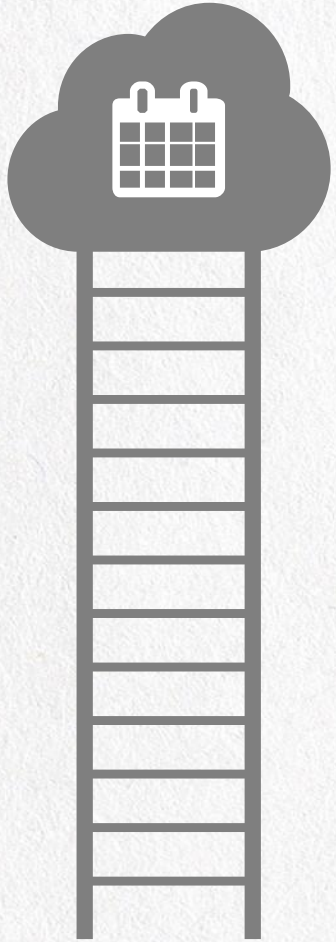
Algorithms from the STL offer fast, robust, tested and maintained code for a lot of standard operations on ranged elements. Don't reinvent the wheel !


```cpp
#include <vector>
#include <algorithm>

bool compare( const int & first, const int  & second )
{
    return (first < second);
}

int main( int argc, char **argv )
{
    std::vector<int> v(10);
    std::sort(v.begin(), v.end(), &compare);

    return 0;
}
```



Exercises 1	Exercises 2
Write a template stack class using the STL vector class	Write a generic vector class with iterators and benchmark it against the STL vector class



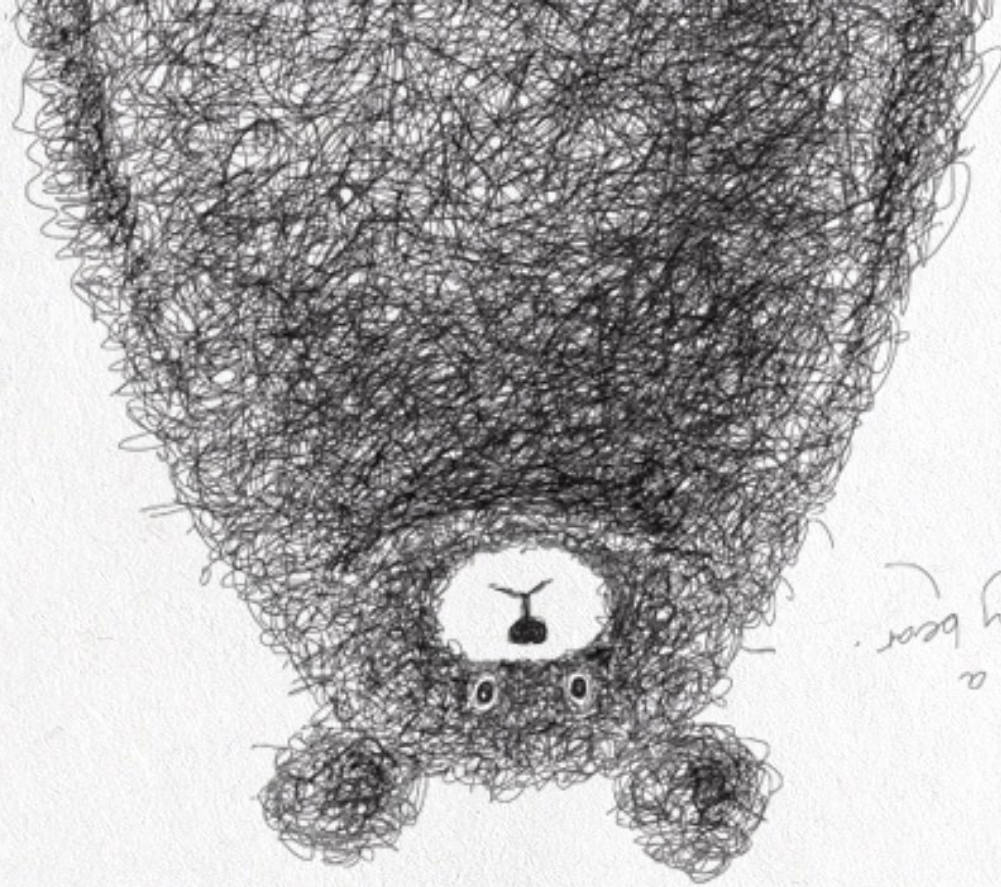


PART 08

review good thing

divide code to header and .cpp file

Pointers and Function Pointers

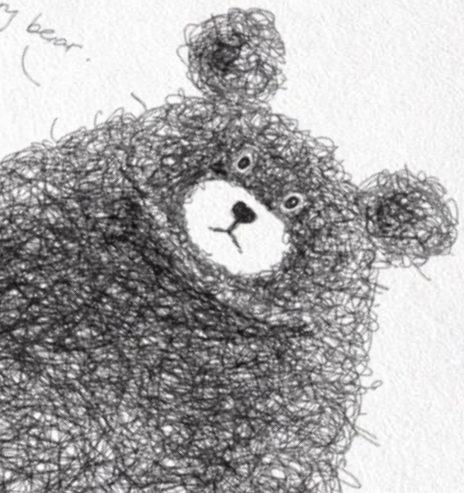


Class Declaration (.h file)

The class declaration typically goes in the header file, which has the extension .h (or, less commonly, .hpp to distinguish from C headers).

"there will be a thing and here's how it will look like"

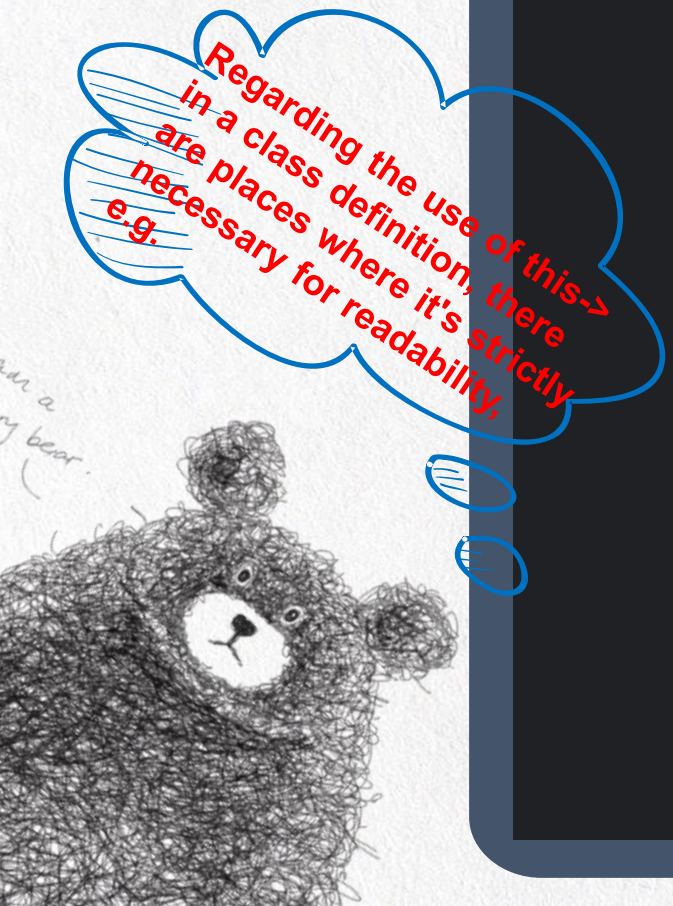
```
// File: polygon.h
#include <string>
class Polygon {
// Private members and methods are only accessible via methods in the class definition
private:
    int num_sides;        // Number of sides
// Protected members and methods are only accessible
//in the class definition or by classes who extend this class
protected:
    std::string name;   // Name of the polygon
// Public members and methods are accessible to anyone who creates an instance of the class
public:
    // Constructors
    Polygon(const int num_sides, const std::string & name);
    // <--- This constructor takes the number of sides and name as arguments
    // Getters and Setters
    int GetNumSides(void) const;
    void SetNumSides(const int num_sides);
    std::string & GetName(void) const;
    void SetName(const std::string & name);
}; // <--- Don't forget the semicolon!
```



Class Definition (.cpp file)

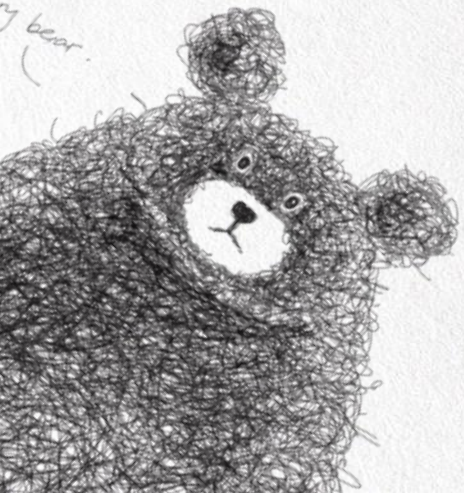
The class definition typically goes in the .cpp file. The definition extends the declaration by providing an actual implementation of whatever it is that you're building.

"Right, that thing I told you briefly about earlier? Here's how it actually functions".




```cpp
// File: polygon.cpp
#include <string> // <--- Required for std::string
#include "polygon.h" // <--- Obtains the class declaration
// Constructor
// You must scope the method definitions with the class name (Polygon::)
// Also, see the section on the 'explicit' keyword for a warning
//about constructors with exactly one argument
Polygon::Polygon(const int num_sides, const std::string & name) {
    this->num_sides = num_sides;     // 'this' is a pointer to the instance of the class.
    //Members are accessed via the -> operator
    this->name = name;             // In this case you need to use 'this->...'
    //to avoid shadowing the member variable since the argument shares the same name
}

// Get the number of sides
int Polygon::GetNumSides(void) const {
    // The 'const' here tells the compiler that you guarantee that you won't modify the object
    //when this function is called. This allows it to perform optimizations that
    //it otherwise may not be able to do
    return this->num_sides;
}

// Set the number of sides
void Polygon::SetNumSides(const int num_sides) {
    this->num_sides = num_sides;
}
// Get the polygon name
std::string & Polygon::GetName(void) const {
    return this->name;
}
// Set the polygon name
void Polygon::SetName(const std::string & name) {
    this->name = name;
}
```

Class Utilization (Another .cpp file)

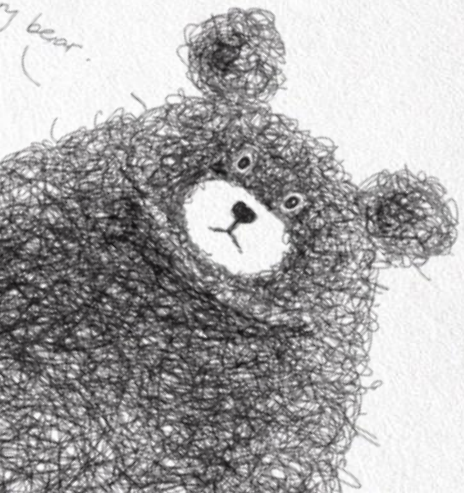


```cpp
// File: main.cpp

#include <string>
#include <iostream>

#include "Polygon.h"    // <--- Obtains the class declaration

int main(int argc, char * argv[]) {
    // Create a polygon with 4 sides and the name "Rectangle"
    Polygon polygon = Polygon(4, "Rectangle");

    // Check number of sides -- Prints "Rectangle has 4 sides"
    std::cout << polygon.GetName() << " has " << polygon.GetNumSides() << " sides"<< std::endl;

    // Change number of sides to 3 and rename to "Triangle"
    polygon.SetNumSides(3);
    polygon.SetName("Triangle");
}
```

Pointers

An array name is a CONSTANT pointer to the beginning of the array.

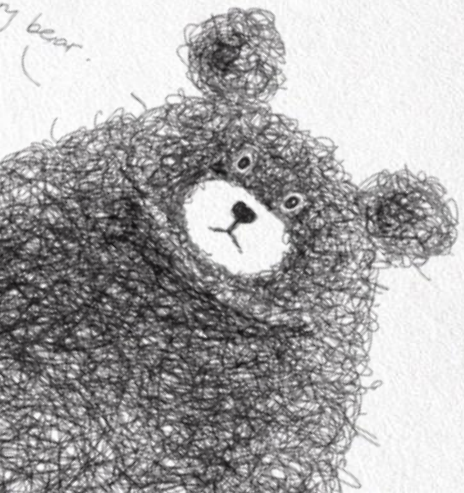
```
template <class TypeName>
void Proceed(TypeName *ArrayName)
{
    for (int i = 0 ; i < 6 ; i++)
    {
        cout << *ArrayName << endl;
        ++ArrayName;  // Switches to the next element of the array.
    }
    cout << endl;
}
int main()
{
    char String1[] = "STRING";
    int NumArray1[] = {1,2,3,4,5,6};
    Proceed (String1);
    Proceed (NumArray1);
    Proceed ("Test");
    return 0;
}
```



Pointers part 2

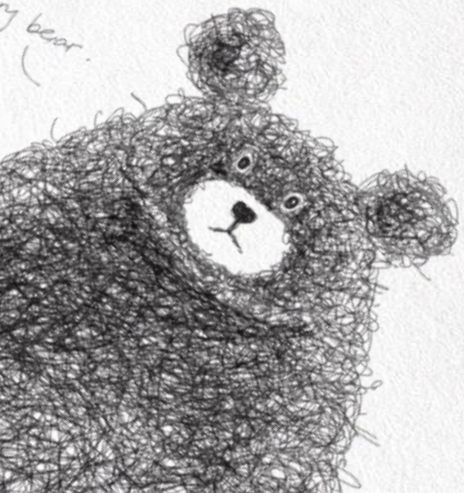
An array name is a CONSTANT pointer to the beginning of the array.

```
int x = 1;
int y = 2;
int z = 3;
// Read from right to left :
int *const Ptr1 = &y;  // Ptr1 is a constant pointer to an integer.
const int *Ptr2 = &x;  // Ptr2 is a pointer to an integer constant.
 const int *const Ptr3 = &z;  // Ptr3 is a constant pointer to an integer constant.

// Because a pointer holds an address so Ptr1 cannot hold any other ADDRESS.
// Ptr1 = &x;  is an error.
// *Ptr1 = 7;   completely acceptable.
cout << "x = " << x << endl;
*Ptr1 = 7;
cout << "New value of x is : " << x << endl;
// Ptr2 is not a constant pointer, so it can point to another address.
// But it points to an integer constant, so it cannot modify the value of the address it holds.
// *Ptr2 = 8;  is an error.
// Ptr2 = &z;  completely acceptable.
Ptr2 = &z;
cout << "The value of *Ptr2 is " << *Ptr2 << " and the value of z = " << z << endl;
// Ptr3 is a constant pointer to an integer constant, so it cannot change the address it
//    holds nor the value of the address it holds. So it's Read-Only.
// *Ptr3 = 10;  is an error.
// Ptr3 = &x;   is an error too.
cout << "The value of *Ptr3 is " << *Ptr3 << endl; // Read-Only
```



Pointers part 3

void pointers can point to any data type.
To dereference a void pointer it must be converted to appropriate pointer type using cast operator.
e.g. nNum1 = *(int *)(A_void_Ptr);

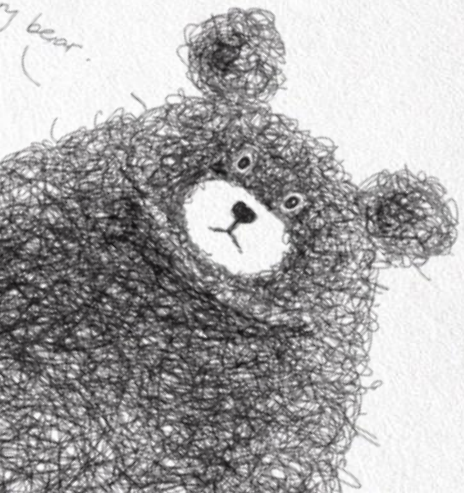


```
int main ( void )
{
    int n = 5;
    char c = 'c';
    double d = 54.4;
    void * vPtr;

    vPtr = &n;
    cout << *(int *)vPtr << endl;
    vPtr = &c;
    cout << *(char *)vPtr << endl;  // It could have been *(int*)... it's not a syntax error,
                                    //  but a logic error. Because char is 1B but int is
                                    //  normally 4B. so it will use 3 adjacent extra bytes.
    vPtr = &d;
    cout << *(double *)vPtr << endl;
    return 0;
}
```

Function Pointers

pointer to a function is dereferenced to execute the function.
Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

```
#include <iostream>
using namespace std;
void Function0 (int Num);
void Function1 (int Num);
void Function2 (int Num);
void Function3 (int Num);
int main( void )
{
    const int ArraySize = 4;
    void (*F[ArraySize])(int) = {Function0,Function1,Function2,Function3};
    // F is an array of 4 pointers to functions that each take an int as an argument and
    //return void.
    int Choice;
    cout << "Enter your choice from 0 to 3 : ";
    cin >> Choice;
    if (Choice >=0 && Choice < ArraySize)
        (*F[Choice])(Choice);
    else cout << "Out of Range!\n";
    return 0;
}
void Function0 (int Num){cout << "You entered " << Num << ", so Function0 was called.\n";}
void Function1 (int Num){cout << "You entered " << Num << ", so Function1 was called.\n";}
void Function2 (int Num){cout << "You entered " << Num << ", so Function2 was called.\n";}
void Function3 (int Num){cout << "You entered " << Num << ", so Function3 was called.\n";}
```



Lambda expressions

Constructs a closure: an unnamed function object capable of capturing variables in scope.

```cpp
// generic lambda, operator() is a template with two parameters
auto glambda = [](auto a, auto&& b) { return a < b; };
bool b = glambda(3, 3.14); // ok

// generic lambda, operator() is a template with one parameter
auto vglambda = [](auto printer) {
    return [=](auto&&... ts) // generic lambda, ts is a parameter pack
    {
        printer(std::forward<decltype(ts)>(ts)...);
        return [=] { printer(ts...); }; // nullary lambda (takes no parameters)
    };
};
auto p = vglambda([](auto v1, auto v2, auto v3) { std::cout << v1 << v2 << v3; });
auto q = p(1, 'a', 3.14); // outputs 1a3.14
q();
```

Boost

Boost free peer-reviewed portable C++ source libraries ☺

If you need:

- regex
- function binding
- lambda functions
- unit tests
- smart pointers
- noncopyable, optional
- serialization
- generic dates
- portable filesystem
- circular buffers
- config utils
- generic image library
- TR1
- threads
- uBLAS

looking forward to more learning

Persian movie:

- https://maktabkhooneh.org/course/%D8%A8%D8%B1%D9%86%D8%A7%D9%85%D9%87-%D9%86%D9%88%DB%8C%D8%B3%DB%8C-%D9%BE%DB%8C%D8%B4%D8%B1%D9%81%D8%AA%D9%87-mk187/

Step by step:

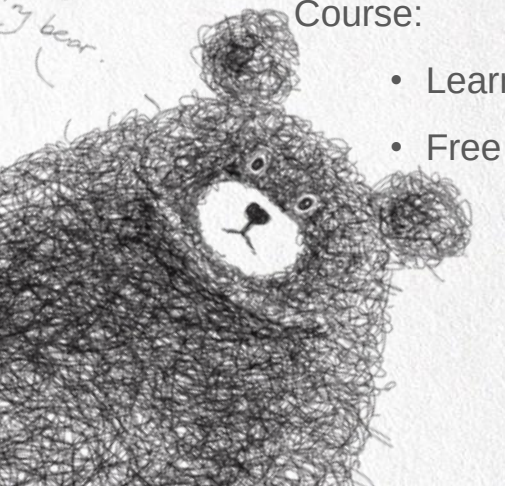
- https://github.com/caveofprogramming/advanced-cplusplus/wiki/Learn-Advanced-C-Plus-Plus-Tutorial
- https://github.com/Florianjw/cpp_tutorial

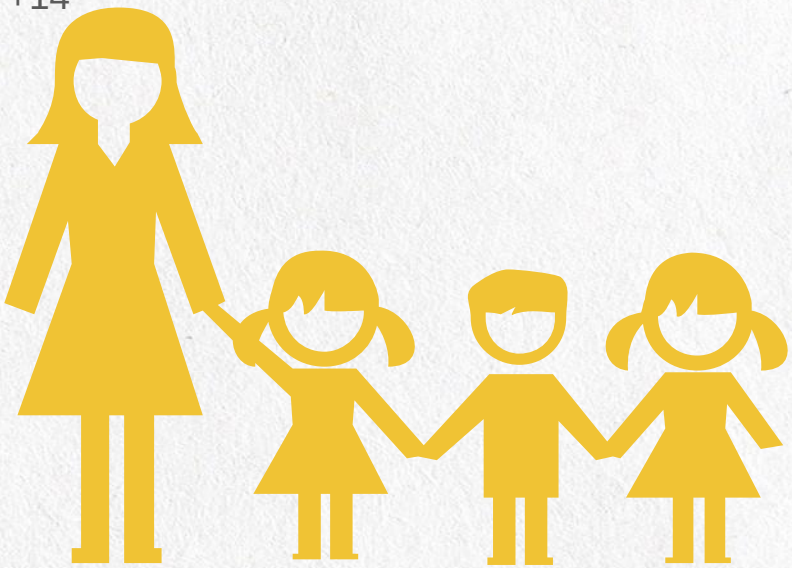
Book:

- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14
- C++ Primer 5th Edition

Course:

- Learn Advanced C++ Programming (Udemy)
- Free C++ Tutorial & Classes (Google)





THANKS