COMP 322

Table of Contents

Books

C++ Programming Language, official (link)
C++ Primer recommended by prof (link)

Other books:
Data structures in C++ (link)
C++ fundamentals (link)
Hands-On System Programming with C++ (link)

Summary

[L1] Intro to C++

C++ from C

	C++	C
Author	Bjarne Stroustrup	Dennis Ritchie
Purpose	OOP with performance of C	Development of UNIX OS
Memory management	new & delete	malloc & free
Operator & function overloading	Possible	Not possible

Namespaces	Exist	Don’t exist
Reference variables	Exist	Don’t exist
Generic programming paradigm	Possible via Templates	Not possible
OO programming paradigm	Classes	Functions and data are separate

C++ from Java

	C++	Java
Author	Bjarne Stroustrup (DEN)	James Gosling (CAN)
	Compiled (runs as native binary on target)	Interpreted (runs through VM)
	Compatible with C	Require JNI to run C/C++ code (Java Native Interface)
Multiple programming paradigms	Yes	Yes, but favors OOP
Inheritance	Multiple	Single
Memory management allowed?	Allows via pointers	No pointers, garbage collector deals with objects

Compiling & Running C++ Code

C++	Java
Single or multiple files Separate header files .h with declarations from implementation .cpp	One file per class, matching names

C++ Basics:

Syntax:

# using namespace std; main() return 0	preprocessor command entrypoint code ended succesfully

Data types:

Same as C: int, float, double, char
C++ only: string, bool, auto

Operators:

- Assignment: =
- Math: + - * / %
- Relational: == != < > <= >=
- Logical: && || !
- Bitwise: & | ~ ^ << >>

Flow control:

Expressions can use logical and relational operators

- Conditional: if (expression) and switch (integer or char - condition limited to equality); ternary (cond)?x:y
- Loops: for(init, condition/termination, inc/dec); while (condition); do {...} while(condition);

[L2] C++ Basics

Standard input & output

- #include <iostream> → streams
- cout == printf() == system.out.print() → stdout is default
    - cout << var << endl;
    - cout << getAge() << endl;
    - cout << "hello" << endl;
- cin == scanf() == scanner class in Java → keyboard is default
    - cin >> stringVariable; → only one word
    - getline(cin, stringVariable); → get a whole line

Functions

- Same as in C and Java
- Declaration (prototype/signature): return type name arguments type
- Return type void if no return is needed

Recursion:

Function can call itself. Tail-recursive is more performant because compiler can optimize (like iterative)

Performance:

Performance of code (how many calls are needed?), performance of memory (what uses less?)

Function Overloading:

Function that receives an int can also get a float, the compiler will turn it into a int → implicit conversion
Functions can have same name but have to have different aspects:

- if have the same argument types, but different return types → doesn't work
- if have the same return types, but different argument types → works

Scope and lifetime of variables

- Variables have name, type, address, scope & life span

Variable type	Scope	Life Span	Errors
local	within the closest { }	From when declared to automatically destroyed when out of scope	Compiler error if called outside scope
static local → memoization in ruby	within the closest { } → scope is same	From when declared to end of the program → lifespan is extended	
global → avoid using	available everywhere (across files)	From when declared outside any functions to end of program	Can cause name collision
static global	only available in the file where they are declared → scope is reduced	From when declared outside any functions to end of program → lifespan is the same	
dynamically allocated	follows the same rules as the ones above	From when allocated (new) until we deallocate (delete), may live even after out of scope	Can cause memory leak if never destroyed. Compiler error if try to use out of scope

Namespaces

- Help avoid name collision (ie. same var name in different modules), but also makes it easier to fix (by not having to change the var name everywhere)
- in QuebecTemp.h: namespace QC { double getTemp() { … } }; in main.cpp: #include "QuebecTemp.h"... QC::getTemp()

Static (extra not in slides)

Usage:

- declarations of namespace members with static storage duration and internal linkage
- definitions of block scope variables with static storage duration and initialized once
- declarations of class members not bound to specific instances

[L3] Pointers and References

Pointers

Testing pointers: http://cpp.sh/6cc6p
Testing arrays: http://cpp.sh/4rkue (sizeof(arr)= number of elements * size of element types)

```
ptr == &var → address of var
*ptr == var → value stored in var
```

- Locations in memory: address is assigned when declared, accessible by var name

- Operators:
    - * dereference operator
    - &
- Type of pointer should match type of variable it's pointing to (ie. `int var = 1; int *ptr = &var;`)
- Void pointers:
    - cannot use a void* to operate on the object it addresses
    - can use it to compare it to another pointer, we can pass it to or return it from a function, and we can assign it to another void* pointer
- Pointer types: can't point a double pointer to an int pointer (vice-verse)
- Double pointer: one practical use case is, whenever you are changing a pointer itself in a function passed as an argument, it has to be passed as a pointer to pointer (or double pointer) - you want to change the value of the pointer passed to a function as the function argument, to do this you require pointer to a pointer

Pointer Arithmetic

- Only addition and subtraction
- Goes to the previous/next memory location taking into account the size of the data it holds → important to know for when working with dynamic memory allocation

Pointer mistakes

- Dereferencing invalid pointers: uninitialized pointers point to some random location → memory corruption. Fix: always assign to nullptr (int *ptr = nullptr;)

```
int *ptr; → not initialized, pointing to random place
*ptr = 12; → memory corruption
```

- Mixing operator precedence rules: (++ higher precedence than *)
    - *++ptr: increments the pointer address
    - ++*ptr: increments value of data
- Check if pointer is NULL before accessing it. ie. if (ptr != NULL)
- Don't return a pointer to a local variable as part of function because that address won't be for that var anymore outside that scope

When to use pointers

- no need to statically reserve a huge array in advance
- to implement complex data structures
- when dynamically allocating memory (ie. creating an array we don't know the size yet)
- passing functions as params

Problems with Pointers

Memory leaks, dangling pointers, buffer overflow

Const Pointers, Variables and Methods

Pointers: You can't assign a different memory location to this pointer → compiler error

```
const int *ptr = NULL;
ptr = &i; → not allowed
```

Variables: makes the variable constant, its value can't be changed later on

Method: a const method prevents object data members to be changed from within that method (ie. can't change state of object). → mentioned in slides of Exceptions (inheriting from class exceptions)

References (aka Aliases)

Testing: http://cpp.sh/8ujwk

- Multiple names to the same variable → not creating a new variable
- Can't be rebound to refer to another object (example), so must always be initialized
- References are not objects, so we can't define a reference (pointer) to a reference
- Usually the type of var and reference must match (some exceptions)

Passing Arguments by References

Passing by value	Passing by reference	Passing by pointer
Compiler creates another two vars and then copies the value into them	No new variables are created, calculation directly in initial value	Creates new pointer variables to store the pointers to the vars
Uses twice more memory because has to store the values inside and outside function	Much more memory efficient because only stores the values once	Uses more memory because has to store the values and the pointers

References	Pointers
must initialize when define	can declare without initializing
can't change to reference another variable	can point to a different variable
can't be NULL	can be NULL
shares memory address with the variable it references	has its own memory address
safer	less safe
can't be used with dynamic memory	can be used with dynamic memory

[L4] Memory Management

Automatic/Static memory vs Dynamic memory

Automatic (aka Static)	**Dynamic**
Amount of memory needs to be known in advance	No need to know in advance the amount
All normal variable declarations use this type	Use `new` or `new[]` to allocate this type of memory
Automatically liberated when variable out of scope	Use `delete` or `delete[]` to deallocate this type of memory, no GC

Allocated in the stack	Allocated in the heap (aka free store)

Dynamic Memory

C++ vs C Dynamic Memory

C++	**C**
int *x = new int;	int *x = (int*)malloc(sizeof(int));
new is type aware, no need to do type casting	malloc is type void, not type aware, needs type casting & size of type

Dynamic memory issues

- Memory leaks: never deletes a memory
- Dangling pointers: pointer that points to invalid data or to data which is not valid anymore (dereferencing a pointer after deleting it or deleting a pointer more than once)
- Buffer overflows: where overflows the allocated memory and writes on adjacent memory
- Use `new []` with `delete`: only deletes the first element → memory leak

Ensure that objects are only created when needed and destroyed when not needed anymore.

Initialization

Use operator new to initialize newly created objects:

```
int *i = new int(2); → creates one element initialized to 2

int *i = new int; → creates one int element
*i = 2; → initializes element to 2

int *i = new int[2] → creates array of 2 uninitialized elements
int *i = new int[2]() → creates array of 2 elements = {0, 0}
int *i = new int[2]{3, 5} → creates array of 2 elements = {3, 5}
```

Array and Pointers Relationship

- ptr == &array[0] since array holds always a pointer to the first element of the array
- can't create an array and then try to assign it a different array (link) → acts like a constant pointer

Array & Pointer Notations

Array Notation (link)	Pointer Notation (link)

`int * getIntArray(int size) {`
`  int * ptr = new int[size];`
`  return ptr;`
`}`
	`int * getIntArray(int size) {`
	`  int * ptr = new int[size];`
	`  return ptr;`
	`}`

```
int main() {
    int *x = getIntArray(5);
    for(int i=0; i<5; i++){
        x[i] = i;
    }

    for(int i=0; i<5; i++){
        cout << x[i] << endl;
    }
    delete [] x;
}
```

```
int main() {
    int *x = getIntArray(5);
    for(int i=0; i<5; i++){
        *(x+i) = i;
    }

    for(int i=0; i<5; i++){
        cout << *(x+i) << endl;
    }
    delete [] x;
}
```

When to use array vs Pointers

Array	Pointer
Size is known	Size is not known
Can't be resized	When size can be changed
When the memory initialized can't change	Can be pointed to a different address during execution

Strings

Strings in C and C++

C++ String	C Char Array

string a = "hello";	sequence of chars terminated by null-char
	`char arr1[] = {'a', 'b'};`
	`cout << sizeof(arr1) << endl;`
	`=> 2`
	`char arr2[] = "ab";`
	`cout << sizeof(arr2) << end;`
	`=> 3`
	`char arr3[] = {'a', 'b', '\0'};`
	`cout << sizeof(arr3) << endl;`
	`=> 3`
	`char *arr4 = "ab";` → pointer notation

#include <string> → easier to use, a lot more functionality, cleaner code	#include <cstring> for string manipulation

C++ strings

Using pointer notation:

```
string* str(new string[3]{"Hello","World", string(2, '+')});
       object     array of 3 elements
for (string* p=str; p != str+3; p++){
    cout << *p << " ";
}
delete [] str;
```

```
// same as:
string* str = new string[3]{"Hello","World", string(2, '+')};
```

C++ String Streams

- #include <sstream>
- manipulate strings like IO file streams, used to convert from-to typed objects & char streams
- can be both input and output streams, also using << and >>

```
stringstream ssVar;
ssVar << "Write this to this ssVar";
cout << ssVar.str() << endl;

string strWord;
while(ssVar >> strWord)
    cout << strWord << " ";
```

[L6] Classes in C++

Review of OO

- to design modular and reusable system
- object: data + methods (logic): groups multiple data elements and attach intelligence to manipulate them → encapsulation

C Structs

- first step to objects
- compound data type: before we could have only one type in arrays
- data and functions are still separated

C++ Classes

- user defined type
- has data and methods (function inside a class) → are members of that class
- members have different access types: public, private, protected → access specifiers/modifiers
    - if defined with class default is private
    - if defined with struct default is public
- public members are the interface of class
- objects: instances of class
- this keyword: special pointer to the current object, `->` operator can be used to access that pointer (`this->age = 13;`)
- multiple classes per file are allowed

```
class Person {
  public:
     void setAge(int age){
     this->age = age;
     }
     int getAge(){
     return this->age;
     }
```

```
  private:
      int age;
};

int main(){
  Person Mike;
  Mike.setAge(24);
// OR
  Person *Tyler = new Person;
  Tyler->setAge(14);

  cout << Mike.getAge() << endl;
  cout << Tyler->getAge() << endl;

  delete Tyler;
}
```

Access modifiers

Public	Private	Protected
Accessible from anywhere	Accessible only from within class → compilation error if called from outside; friend functions can access	Accessible from within class and also children classes can also call (inheritance); friend functions can access
Needs to be explicitly declared	By default	Needs to be explicitly declared

- Private/Protected is used to keep data private and then provide public interface (getters and setters) to access data → data hiding

Member Methods

Definition

Inside class	Outside class
considered inline (will be copy pasted into the place where it's called by compiler) → faster but executable occupies more memory because code is bigger	cleaner interface, but takes more cpu to do work because has to be found in function table, get to the function, do work etc. Can also be declared to be inline with the modifier

`class Person {`
`  public:`
`  void setAge(int age){`
`  this->age = age;`
`  }`
`};`
	`class Person {`
	`  public:`
	`  void setAge(int age);`
	`};`
	
	`void Person::setAge(int age){`
	`  this->age = age;`
	`}`

Constructors

- used when instantiating a class → if not provided, compiler provides one that builds types for you (unless you have dynamic types)
- initializes the data members of class
- must have same name as class, must be declared public (except a few situations)
- has no return type (doesn't return values)
- default constructor: constructor without params → used to initialize data members to default values (static vars normally get initialized to 0)
- as many constructors as needed, personalized using params
-

```
class Person {
  public:
     Person();
     Person(int age);
  private:
      int age;
};

Person::Person(){
  this->age = 0;
}
Person::Person(int age){
  this->age = age;
}

int main(){
  Person Mike; // age will be 0
  Person Maria(87); // age will be 87
}
```

Initialization list

- initialization list is listed outside body of constructor
- much more efficient: normally not noticed for built-in data types, but for user-defined ones can make a big difference
    - in body
    - with initialization list

Default values in body	Default values in initialization list
will build the data type and then have to do an assignment (copying the data into var) = 2 operations per variable	injects value directly into variable without creating a new one

`class Person {`
`  public:`
`  Person(int age, char sex);`
`};`
 	
`Person::Person(int age, char sex){`
`  this->age = age;`
	`class Person {`
	`  public:`
	`  Person(int age, char sex);`
	`};`
	
	`Person::Person(int age, char sex):age(age), sex(sex){`

```
  this->sex = sex;
}
```
```
}
```

Destructors

- Java has no user-facing destructors: it has garbage collection
- C++: destructor is implicitly called when object is out of scope, cleans resources allocated to an object
- 1 destructor per class
- Default provided if not declared: deletes data members, but not dynamic memory
- Has ~ and the same name of the class ie. `~Person()`
- Has no return value
- Has to be public (unless for design patterns)

Works in different ways, depending on how the object was created: if dynamically, only called when `delete` is called.

Object created directly:

```
{
  // constructor:
  Person Mike("Mike", 24);
  // do something
  cout << Mike.getAge() << endl;
} // destructor gets called
```

Pointer to object created (object in dyn. mem.)

```
GPS * gps;
// constructor called only on next line
gps = new GPS(14, 12, 10);
// destructor on next line
delete gps;
```

[L7] Classes & Inheritance

Example: http://cpp.sh/4qrsp

Classes behind the scenes - default behavior

Compiler provides 4 methods by default for an empty class. If any of these are declared, compiler won't provide the default ones:

- default constructor
- default destructor
- copy constructor: receives an object of same type as param
- copy assignment operator: assign an object to another object with =. By default, shallow copy only: primitive types are copied, but anything with dynamic memory isn't

If want to prevent users from using copy constructor and/or copy assignment operator has to provide them both, with empty body and make them private.

```
class Person {};

int main(){
  Person p1; // default constructor
  Person p2 = p2; // copy assignment operator
  Person p3(p2); // copy constructor
} // default destructor
```

Copy constructor:

To override it, this is the signature:

```
Person(const Person& obj) { … }
```

In Java can be obtained by using "Cloneable".

Copy assignment operator:

Usually override if wants to provide deep copy of an object
To override it, this is the signature:

```
Person& operator=(const Person& obj) { … }
```

Friendship

May break OOP concepts: data hiding and encapsulation because private and protected members/methods are available to friend functions/classes.

```
class Person {
public:
  friend int getAge(Person& p);
};

int getAge(Person& p) {
  return p->age;
}
```

Inheritance

- Works similar to Java, but can have multiple inheritance.
- Allows to inherit (extend) members of another class
- Reuse methods and data from a base class
- Children (derived) can access all public and protected members of parent (base), not private
- Children can add their own members
- Parent can't access children's members directly

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

  void setCapacity(int i) { capacity = i; }
  void fly() { cout << "Aircraft flying at " << capacity << endl; }
protected:
  int capacity;
};

class Boeing: public Aircraft {
public:
  Boeing() { cout << "Boeing constructor" << endl; }
  ~Boeing() { cout << "Boeing destructor" << endl; }
```

```
}

int main() {
  Aircraft a;
    => Aircraft constructor
  a.setCapacity(50);
    => 50
  a.fly();
    => Aircraft flying at 50
  Boeing b;
    => Aircraft constructor
    => Boeing constructor
  b.setCapacity(100);
    => 100
  b.fly()
    => Aircraft flying at 100
}
    => Boeing destructor
    => Aircraft destructor
    => Aircraft destructor
```

Construction/destruction order

Constructors order:

When child is instantiated:

1. calls parent constructor (unless explicitly defined to call one with params, calls the default one)
2. calls child constructor

When parent's constructor is not called explicitly:

```
class Aircraft {
public:
  Aircraft() { cout << "Aircraft default constructor" << endl; }
  Aircraft(int i) {
    capacity = i;
    cout << "Aircraft with arg constructor" << endl;
  }
  ~Aircraft() { cout << "Aircraft destructor" << endl; }

  void setCapacity(int i) { capacity = i; }
  void fly() { cout << "Aircraft flying at " << capacity << endl; }
protected:
  int capacity;
};

class Boeing: public Aircraft {
public:
  Boeing() { cout << "Boeing default constructor" << endl; }
  Boeing(int i) {
    capacity = i;
    cout << "Boeing with arg constructor" << endl;
  }
```

```
  ~Boeing() { cout << "Boeing destructor" << endl; }
}

int main() {
  Boeing b(300);
    => Aircraft default constructor
    => Boeing with arg constructor
    => capacity = 300
  b.fly()
    => Aircraft flying at 300
}
    => Boeing destructor
    => Aircraft destructor
```

When parent's constructor is called explicitly:

```
class Aircraft {
public:
  Aircraft() { cout << "Aircraft default constructor" << endl; }
  Aircraft(int i) {
    capacity = i;
    cout << "Aircraft with arg constructor" << endl;
  }
  ~Aircraft() { cout << "Aircraft destructor" << endl; }

  void setCapacity(int i) { capacity = i; }
  void fly() { cout << "Aircraft flying at " << capacity << endl; }
protected:
  int capacity;
};

class Boeing: public Aircraft {
public:
  Boeing() { cout << "Boeing default constructor" << endl; }
  Boeing(int i):Aircraft(i) {
    capacity = i;
    cout << "Boeing with arg constructor" << endl;
  }
  ~Boeing() { cout << "Boeing destructor" << endl; }
}

int main() {
  Boeing b(300);
    => Aircraft with arg constructor
    => Boeing with arg constructor
    => capacity = 300
  b.fly()
    => Aircraft flying at 300
}
    => Boeing destructor
    => Aircraft destructor
```

Destructor order:

When out of scope or deleted:

1. calls child destructor
2. calls parent destructor

Types of inheritance

If access type isn't declared, then **private** by default.

.

Public:	Access doesn't change: - Public members in parent stay public in child - Protected members in parent stays protected in child - Private members from parent aren't accessible from child
Protected	Access partially changes: (only used in design patterns) - Public members in parent become protected in child - Protected members in parent stays protected in child - Private members from parent aren't accessible from child
Private (default)	Access changes: (can be called from child, but not from outside, ie. main) - Public members in parent become private in child - Protected members in parent become private in child - Private members from parent aren't accessible from child

Is-A vs Has-A

- A is B: inherit. Poodle is a Dog
- A has B: have it as a pointer. Engine has an Engine.

Polymorphism

```
class Aircraft {
public:
  Aircraft() { cout << "Aircraft constructor" << endl; };
  Aircraft(int i) { capacity = i; cout << "Aircraft const with arg" << endl; };
  ~Aircraft() { cout << "Aircraft destructor" << endl; };

  void setCapacity(int i) { capacity = i; }
  void fly() { cout << "Aircraft flying at " << capacity << endl; }
protected:
  int capacity;
};

class Boeing: public Aircraft {
public:
  Boeing() { cout << "Boeing constructor" << endl; };
  Boeing(int i) { capacity = i; cout << "Boeing const with arg" << endl; };
  ~Boeing() { cout << "Boeing destructor" << endl; };
  void fly() { cout << "Boeing flying at " << capacity << endl; }
};
```

```
int main(){
  Aircraft* a; // pointer of Aircraft type
    => Aircraft constructor
  a = new Boeing(300); // assigned memory of Boeing type size, but gets treated
                       // as Aircraft
    => Boeing const with arg
    => capacity = 300
  a->fly();
    => Aircraft flying at 300 - did not call child method
  delete a;
    => Aircraft destructor - memory leak, did not destroy Boeing!
}
```

How to solve these 2 problems? With virtual methods

Virtual methods

Flags to compiler that a method is polymorphic - only needed in parent class (child is also going to be virtual if parent is).

- Can also use `override` since C++11
- Constructors can never be virtual: needs to know the exact type (but virtual can work on partially constructed objects)
- Destructors of classes to be inherited must be set to `virtual`
- Methods don't need to be marked as virtual in children, but should for readability
- Used to avoid diamond problem when multiple inheritance (last slide)

```
class Aircraft {
public:
  Aircraft() { cout << "Aircraft constructor" << endl; };
  Aircraft(int i) { capacity = i; cout << "Aircraft const with arg" << endl; };
  virtual ~Aircraft() { cout << "Aircraft destructor" << endl; };

  void setCapacity(int i) { capacity = i; }
  virtual void fly() { cout << "Aircraft flying at " << capacity << endl; }
protected:
  int capacity;
};

class Boeing: public Aircraft {
public:
  Boeing() { cout << "Boeing constructor" << endl; };
  Boeing(int i) { capacity = i; cout << "Boeing const with arg" << endl; };
  ~Boeing() { cout << "Boeing destructor" << endl; };
  void fly() { cout << "Boeing flying at " << capacity << endl; }
};

int main(){
  Aircraft* a; // pointer of Aircraft type
    => Aircraft constructor
  a = new Boeing(300); // assigned memory of Boeing type size, but gets treated
                       // as Aircraft
```

```
    => Boeing const with arg
    => capacity = 300
  a->fly();
    => Boeing flying at 300 - called child method
  delete a;
    => Boeing destructor - no memory leak
    => Aircraft destructor
}
```

Abstract classes (virtual vs pure virtual methods)

Classes that have at least one pure virtual method.
Abstract classes cannot be instantiated (only derived classes can) → compiler error!

Similar to Java's interface class.

Virtual	Pure virtual
has implementation in base class can be overridden by derived class	has no implementation in base class has to be implemented in derived class

[L8 L9] Operator Overloading

Function overloading

Decide at compile time which method to use: has different signatures.

Class polymorphism

Decide at runtime which class to use: different object accessed through the same interface (cf. example)

Operators overloading

Possible to overload these operators:`+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- , ->* -> ( ) [ ]` ref
Non overloadable: :: . ?: .*

Why overload:

- use with user defined types
- make it more readable

Example: can't use cout << or +-.. in a user defined class.

Signature: `<classname> operator<op> (const <classname>& varname) { }`
const is needed here because it's not supposed to modify the incoming object.

There are two ways to implement this:

Member method:

```
class Vector {
public:
```

```
  Vector(): x(0), y(0), z(0) {}
  Vector(int a, int b, int c): x(a), y(b), z(c) {}
  Vector operator+ (const Vector& vec) {
      Vector result;
      result.x = this->x + vec.x;
      result.y = this->y + vec.y;
      result.z = this->z + vec.z;
      return result;
  }

  void display(){
      cout << x << ", " << y << ", " << z << endl;
  }
private:
  int x, y, z;
};

int main(){
  Vector v1(2, 4, 6);
  Vector v2(3, 5, 7);
  Vector v3 = v1 + v2;
  v3.display();
}
```

Friend function:

Usually makes more sense because it is a utility and not necessarily a behavior of the class itself.
With << (to be used with cout):

```
class Vector {
public:
  Vector(): x(0), y(0), z(0) {}
  Vector(int a, int b, int c): x(a), y(b), z(c) {}
  ostream& operator<< (ostream& os){
      return os << this->x << ", " << this->y << ", " << this->z << endl;
  }

  friend ostream& operator<< (ostream& os, const Vector& vec);
private:
  int x, y, z;
};

ostream& operator<< (ostream& os, const Vector& vec){
  os << vec.x << ", " << vec.y << ", " << vec.z;
  return os;
}

int main(){
  Vector v1(2, 4, 6);
  // with member function
  v1 << cout << endl; // same as v3.(<< cout), where ostream is the class, cout
                      // the obj.
  // with friend function
  cout << v1 << endl;
```

```
}
```

[L10] Exception Handling

What are exceptions

- Errors (bugs in logic) != exceptions (unexpected behavior/errors not dealt with)
- Exceptions = Exceptional circumstances
- Raised when no logical way to continue execution
- OO way of implementing error codes (in C)
- C++ has no `finally`: if code must run after an exception, it must be placed in destructor

Examples: no more memory, no file found, 3rd party api not accessible, etc.

Try - catch

Exceptions must be caught by putting code inside `try {}`. Anything being placed outside of that won't be monitored.

```
try {
  // code
} catch (SomeException& e) {
  cout << e.what() << endl;
} catch (exception& e) {
  cout << e.what() << endl;
} catch (...) {
  cout << "Program terminated due to an error." << endl;
}
```

Signaling exceptions (and propagating to outer code level):

```
throw invalid_argument("Element cannot be an array.");
```

The type in throw should be the same as the one expected as argument in catch.

```
throw "Warning! This is a char * ";
// in catch:
catch (const char* message) { }

// with string:
string msg = "Warning this is a string!";
throw msg;
//in catch
catch (string& msg){ }
```

C++ Standard Exceptions

- To use #include <exception>
- List of exceptions that can be used
- Use exception class as base (parent) class for user defined exceptions
- what() is virtual method that can be reimplemented to personalize a user message

- const throw in declaration: const won't alter state of class, throw: this method can't throw exceptions
- code that is after a throw never gets executed: execution control is transferred to catch block and never returns to the block that threw the exception
- if exception is thrown and not caught, program will abort. Problem: security issue, memory leak, etc. Even if program aborts, all destructors of in-scope objects will be called (automatic objects will be destroyed, but for dynamic memory your destructors have to delete them).

```
class ZeroException: public exception {
  virtual const char* what() const throw(){ // or const noexcept
      return "warning: division by zero.";
  }
};

//initialize object
ZeroException dividedByZeroException;

double getRatio(double a, double b){
  if(b == 0) {
      throw dividedByZeroException;
  }
  return a/b;
}
int main(){
  try {
      double ratio1 = getRatio(5, 0);
  } catch (exception& e){ // or could use ZeroException& e to be more precise
      cout << e.what() << endl;
  }
}
```

Layers of exceptions

- trickles down catches until one can catch it
- has to order catches from most precise to most general (ie. child classes to parent classes)
- try catch blocks can be nested: whatever doesn't get caught in the inner catches, will propagate out to the outer ones (but never back into inner ones)

[L11] Generic Programming

Generic Programming

Close to Generic classes in Java; metaprogramming, templates → code that generates other code
C++ is typed language, Python/Ruby is ducktyped so it can infer type.

- Programming paradigm: types are abstracted, common code is written only once
- Code is processed and transformed by compiler: only necessary code gets created by compiler
- C++ does generic programming via templates

C++ Templates

- declaration and definition (implementation) must be on same file (unless .hpp file, used for templates)
- only needed code gets generated by compiler
- declare new template: `template <typename T>` or `template <class T>`, same thing

With single type:

```
template <typename T>
T getMax(T a, T b){
  return (a > b) ? a : b;
}

int main(){
  float myF = getMax<float>(12.2, 12.3); => 12.3
  int myI = getMax<int>(15, 7); => 15
}
```

With multiple types:

Can support multiple data types: `template <typename T, typename U>` where T and U may or may not have the same type

```
template <typename T, typename U>
void printValues(T a, U b){
  cout << "val1: " << a << ", val2: " << b << endl;
}

int main(){
  printValues<int, float>(9, 12.11); // val1: 9, val2: 12.11
}
```

With a default type

If no type input is supplied, it will use the default one

```
template <typename T, typename U = double>
void printValues(T a, U b){
  cout << "val1: " << a << ", val2: " << b << endl;
}

int main(){
  printValues<int>(9, 12.11); // val1: 9, val2: 12.11
}
```

With specialized method for a type

When a general form doesn't apply to a given type.
The template<> is needed and then also declaring explicitly the types of T in the specialized versions.

```
template <typename T>
T subtract(T a, T b){
  return a - b;
}

template<>
string subtract(string a, string b){
  size_t pos = a.find(b);
  return a.substr(0, pos);
}

int main(){
  cout << subtract<int>(9, 12.11) << endl; // 12.11 gets casted to int

  string a = "Hello World";
  string b = "world";
  cout << subtract<string>(a, b) << endl;
}
```

Function Templates

All examples above are applied to functions.

Class Templates

Data, methods declared defined inside and outside class:

```
template <typename T>
class Coordinates {
  // for variables
  T x, y, z;
public:
  Coordinates(T a, T b, T c){
      x = a;
      y = b;
      z = c;
  }
  // for methods defined within class
  T getX(){
    return x;
  }

  // for methods defined outside class
  T getY();
};

template <typename T>
T Coordinates<T>::getY() {
  return y;
}
```

```
int main(){
  Coordinates<int> pointI(2, 4, -3);
  Coordinates<float> pointF(2.1, 4.9, -3.7);
  cout << pointF.getX() << endl; // 2.1 returns a float
  cout << pointF.getY() << endl; // 4.9 returns a float
}
```

Non-type Parameters:

Templates accept non-typed params (similar to functions)

```
template <typename T, int scale = 1>
class Coordinates {
  T x, y, z;
public:
  Coordinates(T a, T b, T c){
      x = a;
      y = b;
      z = c;
  }

  T getX();
};

template <typename T, int scale>
T Coordinates<T, scale>::getX() {
  return X;
}

int main(){
  Coordinates<int> pointI(2, 4, -3); // scale is 1 by default
  cout << pointI.getX() << endl; // returns 2 from 2 * 1
  Coordinates<float, 2> pointF(2.1, 4.9, -3.7); // scale is given = 2
  cout << pointF.getY() << endl; // returns 4.2 from 2.1 * 2
}
```

Templates Pros & Cons

Pros	Cons (not true since C++17)
- avoid code duplication: faster to write, easier to update and maintain - better than C-macros because has type safety - better performance than inheriting: compile time polymorphism is better than runtime polymorphism	- code bloating: huge compiled file uses more memory, slow to compile - no code hiding: all in same file, so can't hide the implementation - hard to debug: cryptic compiler errors, line of code might not exist in cpp (only in compiled),

Standard Template Library (STL) overview

- Was not part of standard library, but is now
- set of class templates implementing basic data structures

- main components: containers, iterators, algorithms, functions

Containers:

Store objects and data

- sequential containers: list, vector, array, …
- associative containers: map, set, …

```
list<int> intList;
intList.push_front(12);
```

Iterators:

Allow for iteration over containers and accessing their values: abstraction to access different types of containers

```
list<int> intList;
list<int>::iterator it;

intList.push_front(12);
intList.push_front(13);
intList.push_front(14);

for (it = intList.begin(); it != intList.end(); it++)
  cout << *it; // use pointer to get value itself
```

Algorithms:

Collection of algorithms to be applied to containers: sorting, searching, comparing, etc.

Functions:

Classes that overload the function call operator() → also called Functors

- Functor is a function object.
- Functions don't keep state, by using functors can imitate the behavior of a function but keep state.

Validation of code:

defined (code) vs declared (signature only)

1. Is it using C++ things: cout/endl, cin, getline, new/delete
2. Functions:
    a. Have all functions/vars been declared before they were called?
    b. Do all methods have: right structure? Have a name, return type (void if none) and argument types (names are optional), does it return something when required?
    c. Check all types: if function expects int but receives float, it will cast to int
    d. Check if functions have been correctly overloaded
    e. have values been passed by reference?
3. Check the type, scope and lifetime of all variables
    a. if pointer variable, does the type of the pointer match the type of the variable?
    b. if multiple vars being declared in same line including pointer, check what types they will be
    c. was a pointer to a variable being used outside the scope of that variable? -> no guarantee of what the value will be

4. Pointers:
    a. a pointer hasn't been declared but unitialized and then assigned a value (mem corruption)
    b. if pointer initialized to NULL, was there a check for null pointer before trying to access it?
    c. is a pointer function returning an address to a local var that will be destroyed when leaving the function? (L3)
    d. was a pointer defined as const and then tried to re-assign its value to something else?
5. Dynamic memory:
    a. was every mem created also deleted? → memory leak
    b. new with delete and new[] with delete[] → memory leak (if delete on new[])
    c. was a pointer dereferenced after being deleted? → undefined behavior
    d. was a mem deleted twice? → undefined behavior
6. Classes:
    a. check scope of methods+vars → was it created with struct (public) or class (private)?
    b. are private/protected things trying to get called from outside class/scope?
    c. methods defined outside class were properly declared and scoped with void Person::setAge(int age)?
    d. Constructors:
        i. was a constructor provided when dealing with dynamic data types?
        ii. does constructor have same name as class?
        iii. is declared public? → in some cases can be protected/private
        iv. has no return type? → declared: Person(), defined: Person::Person()
        v. was properly initialized? → Person::Person(int age, char sex):age(age), sex(sex) {}
    e. Destructors:
        i. is there only one for a class?
        ii. does it destroy allocated memory?
    f. is a const method trying to change the value of data members of an object?
7. Inheritance classes:
    a. Does it have the right access level for the methods that are being called?
    b. Did it have virtual in the correct places to ensure destructors are working
    c. Is abstract class? Is it being initialized somewhere? → wrong!
8. Input/Output:
    a. if cin → did it want only one word?
9. Exceptions:
    a. Does type thrown match type of arg in catch?
    b. Does the order of catching correct (from inner to outer)?
10. Templates:
    a. is declared and implemented in same file?

Quiz questions to look back:

- L2 rewrite factorial function iteratively
- L4: check double pointers and try to understand!
- const pointers (check what's not possible! - no error), null pointers
    - int errNumb = 0;
    - int *const curErr = &errNumb; // curErr will always point to errNumb (can't point to something else)
    - const double pi = 3.14159;
    - const double *const pip = π // pip is a const pointer to a const object (can't change values of obj)
    -