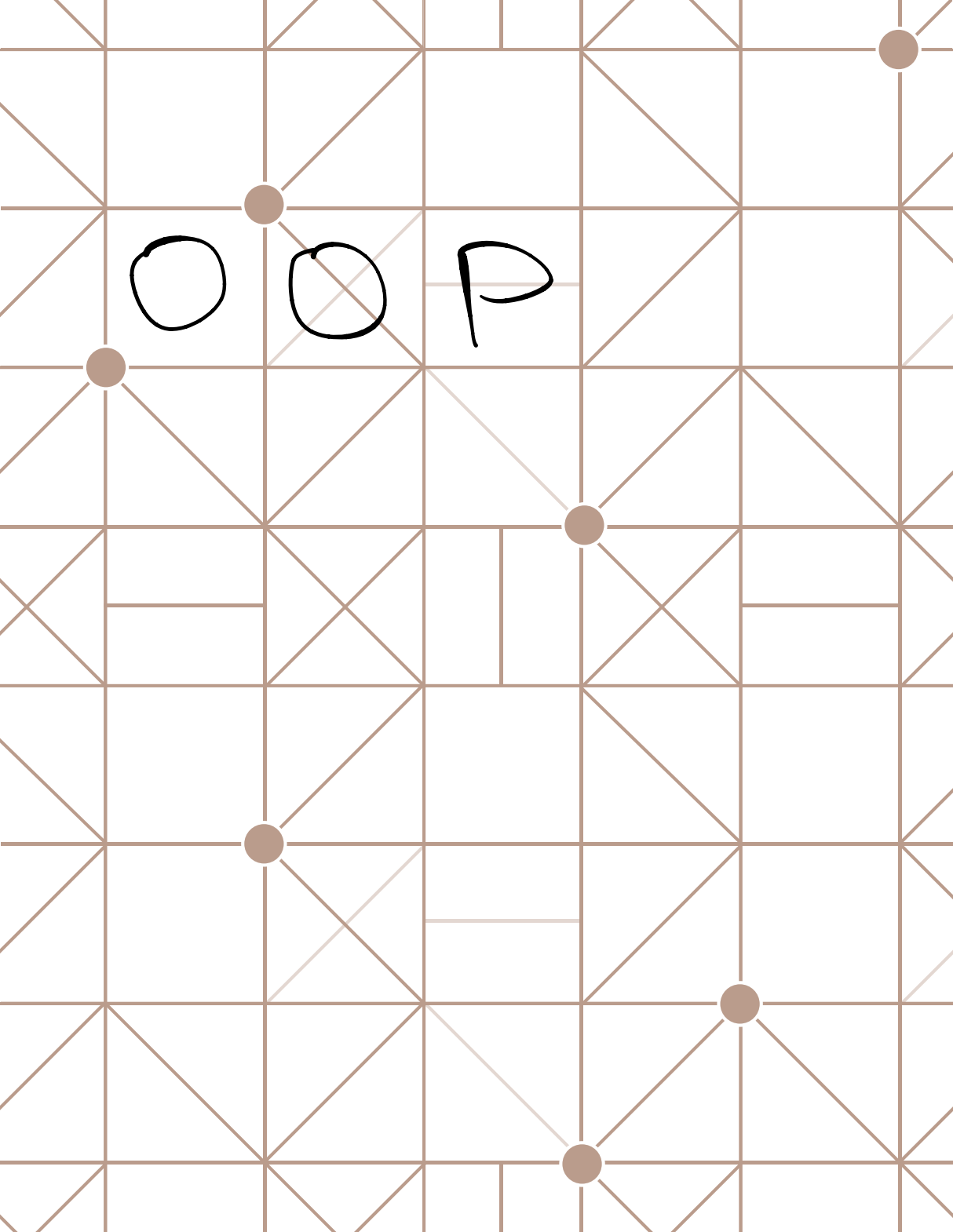

OOP

C++ Basics

- Bjarne Straustrup
- AT&T Bell Labs
- inspired by C and Simula

C Denis Ritchie

Python Van Rossum

Simplest Program

```
int main(){
    return 0;
}
```

- every program must have function main
- can take cmd arguments (but usually does not have arguments)
- must state datatype it return (main returns int, main returns 0 by default)
- comments are //comment written with hashtags

Hello World

```
#include <iostream>
using namespace std;
int main(){
    std::cout << "Hello CS2124!!\n";
}
```

- using namespace std; ← uses namespace in program
- << → output operator
- std → namespace
- cout → where to output

Variables

```
int n;  or  double d;  or  string s;
```

define variables before use

also includes function parameters & function return types

endl ≈ "\n"

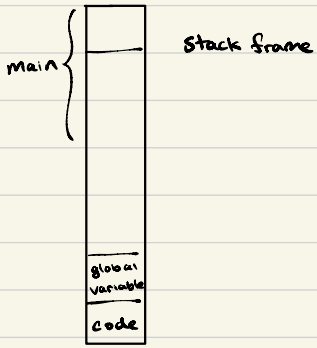
#include <string> → string must be imported

#include vs namespace?

```
int main() {
    int x;
    cout << "x: " << x << endl;
}
```

← undefined

Memory Diagram



Conditions

if (condition) {}

else if (condition) {}

else (condition) {}

and &&

or ||

not !

--x decriment ++x increment

Loop

for (int x=10 ; x> 0 ; --x) {

}

continue

break

do {

} while (condition);

Vectors

```
#include <vector>
vector<int> v;
v.size();
v.push_back();
v[i]
v.capacity();
v.back();
v.pop_back();
vector<int> v1(7,42);
vector<int> v2{1,1,2,3,5,8};
```

can only hold integers

↳ initialized using constructor

↳ true for all non-primitive types

clear size set to 0, capacity?

```
for (size_t i=0; i< v.size(); ++i){
    cout << v[i]<< " ";
}
```

size_t is unsigned

7 elements with values 42

initialize with certain values

Loop

```
for (int value: v) {
    cout << value << ' ';
}
cout << endl
```

copy next item into variable

shorter

don't need index

String

roughly vector of characters

uses double quotes ""

strings are mutable { v[0] += 1 ← ascii value

Files

```
#include <fstream>
ifstream jab("jabberwocky");   // variable; file type
```

remember to close file after opening `jab.close();`

```
jab.open("jabberwocky")
if (!jab {
    cerr << "failed";
    exit(1);
}
```

check if file was successfully openned

```
string something;
jab >> something;
cout << something << endl;
```

← empty string object, not undefined, only primitives are undefined

read file (no whitespace: white space delimited token)

Constructor with no arguments → default constructor

```
while (jab >> something) {
    cout << something << endl;
}
```

if reading 1 token successful

output line

Matrix

```
vector< vector<int>> mat;
```

vector of vectors kekw

Functions

output datatype — method name — input datatype — input name

```
void methodName ( int someInt) {
    ++ someInt;
}
```

← pass by value

ampersand

```
void addOne ( int& someInt) ← reference to input
    ++someInt;
}
```

make immutable — pass by reference (vector may be large)

```
void printVec (const vector<int>& someVec) {
    # prints vector
}
```

function prototypes

can move function down, add function header to top

Structure

```
struct Motorcycle {
    string model;
    int cc;
};
```

may be useful to add print function for structure

```
Motorcycle bike {"Softail", 1746};
```

remember to pass by constant reference for structs

Structures

functions can also be placed inside structs

↳ `void doSomething() {` ← no need for self

...

`}`

in struct, everything is public by default

Class

in classes, everything is private by default

usually, structs are only used to store a few fields

`class Example {`

`private:` ← everything below is private (generally private is after public)

...

`public:` ← everything below is public

...

return values also create copies by default, might be useful to pass by reference

`const string& getItem() const {`

↑ shows function will not modify classes & input

`}` ↑ return immutable reference

constructor → same name as class

`Constructor(const string& aName) {`

c++ constructor initization list ↑ } takes up extra work

↳ initialize attributes that are not primitive

`Constructor(const string& aName) : name(aName) {`

initialization list

↳ const must be in initialization list

`}`

this → variable ≈ this.variable or self.variable

compilation error ↑

no default constructor when another constructor defined

initialized by default constructor ↓

constructor that can take no arguments

Class

default constructor (no) → overload constructor

ostream&
<< → operator<< (ostream& os, const Class& Item) {

same

streams cannot be copied

return os;

}

class Example {

friend ~~~~ ; allows function to access private variables

function header

↳ can also write entire function

}

vector<thing> things; don't need to create object

things.emplace_back("a", "b", 3); and push_back

Address / Pointer

cout << &x << endl; ← address of operator in memory

int* p = &x pointer

cout << *p << endl; dereference operator

pointers are primitive

int* x = nullptr; pointer points to nothing

x = this; pointer to instance of class (self)

pointer->item; arrow operator

↑ equivilant to *(pointer).item

objects have constant size

↓

vector { size, capacity, pointer to array } fixed

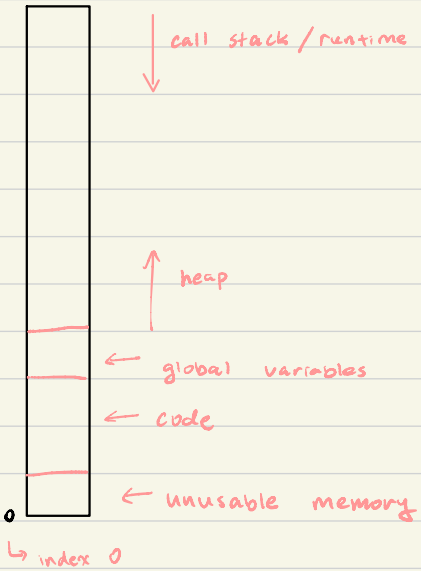
private refers to class as a whole, not instances of objects ☆

const int* p = &x x is immutable using pointer p

int* const q = &x pointer q cannot point to another address

may be useful to have vector of pointers

Memory



Pointer (cont.)

```
Person* p = new Person(name, age);
```

↳ allocates memory in heap & return address

```
delete p;
```
→ de-allocates memory in heap (pointer still exists)

might be important to clear vector after deleting

```
~Person() {delete spouse;}
```
destructor

```
Person(const Person& rhs) {
    p = new int(*rhs.p)
}
```
copy constructor

often appears together
↳ copy control

```
aPerson.operator=(anotherPerson);
```
assignment operator must be method

```
const Thing& operator=(const Thing& rhs) {
    if (this != &rhs) {          ① self assignment test
        delete p;                ② free up resources
        p = new int(*rhs.p);     ③ allocate & copy
    }
    return *this;                ④ return something
}
```

nonprimitive destructors are automatically called in destructor

Vector / Copy Control

dynamic array
- dynamic → allocated on heap
- array → continuous blocks of memory

`data[size] = val;`
- same as `*(data + size) = val;`
- point to address of first element, add based on index

`data = new int[2 * capacity]` → allocate memory on array

`delete[] data;` → delete array

array with capacity 0 will still use memory (metadata)

→ allows chaining

```
int operator[](size_t index) const {      // v[i]   getter
    return data[index];
}
```

→ must return reference

```
int& operator[](size_t index) {           // v[i] = 17   get reference?
    return data[index];
}
```

```
explicit Vector( ~ ~ ~){ //method must be used in this form
    ~ ~ ~                // can only be class
}
```

→ cannot change int pointed to

```
const int * begin() const {
    return data;
}
const int * end() const {
    return data + theSize;
}
```

} allows for ranged for

Const vector

```
int * begin(){
    return data;
}
int * end(){
    return data + theSize;
}
```

location [after] the last item

2 Way Pointer

```
class Froggy;          forward class declaration
void Princess::marries(Froggy& consort){
}
```

define method later

scope qualification operator

forward declaration: - allows for pointers & references
- does not allow for instance of objects

prototype: - include const
- include initialization list

```
friend class Person;
```

Seperate Compilation

Main	Princess	FP
test.cpp	Princess.h	FP.h
	Princess.cpp	FP.cpp

compiler

test.obj	Princess.obj	FP.obj

linker

testPrincess.exe

do not use namespace in header file → namespace pollution ☆

include guard:

```
#ifndef PRINCESS_H   (all caps)
#define PRINCESS_H
    : all code
#endif
```

explicit { constructor that can only take one argument
conversion operator

Conversion operator

```
operator bool() const {
    return ~~;
}
```

→ states conversion type

→ no output needed here

Things that cannot be done

→ cannot create new operator eg. **

→ cannot change arity of operators (unary to binary)

→ cannot change operator precidence

→ try not to overload "and" and "or" → disables short circuiting

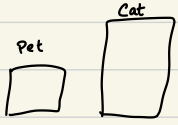
↳ check next argument if 2 operators does not suffice

```
class Pet {
};
class Cat : public Pet {};
```

inheritance → code reuse

→ Principle of Substitutability (Barbara Liskov)

can specify method by scope qualification `Pet::eat();`



← child classes usually takes up more memory

↳ slicing problem

polymorphism increases run time

↳ method call specific address → table of possible addresses

↳ need to turn on polymorphism

```
class Pet {
    virtual void eat() {
```

based = derived → ok, but slicing problem

derived = base → not allowed

methods are inherited, constructors are not inherited

Cat(const string& name) : Pet(name) {}

↳ call pet constructor

child constructors call default constructor of parent, before all others

virtual void draw() = 0; //abstract method, pure virtual method

triangle is still abstract?

void method() override {};

children can only call protected methods on itself

abstract/pure virtual classes are inherited until the abstract method is overloaded

methods with same names but different arguments in derived class can overload methods in parent

↳ using Base::foo; → also works for constructors

parent method

overload vs override → same name & argument related to inheritance

↳ same name & arguments

overload → compile time

override → runtime decision

destructor of a derived class will call destructor of base class at the end of its destructor

~Derived { ~~~ } ← here ~Base()

this → method() is equivilant to method();

however, in constructors, only method from class (not child) is called

↳ also includes copy constructors

multiple inheritance class One: public Two, public Three {};

Mixin: small lightweight class for multiple inheritance

diamond pattern + virtual inheritance

unary operators are methods by default

Iterator

① constructor

② equality (!=) operator

③ increment operator

④ dereference operator

⑤ value

- nested in class it is used for

need iterators for constant & nonconstant

Generics

template < typename T >

↳ all generics must have default constructors

sort(array, arraylen); → half open range [start, stop)

there are different kinds of iterators in the STL

↳ some are stronger than others, eg. support for direct access O(1)

↳ allows for sorting algorithms

convention is to return "end" (arr + len, list.end()) if algorithm fails

Error Catching

`v.at(17);` same as `v[17]`, but checks index

```
try {
    v.at(17) = 42;
}
catch (exception ex) {
    cerr << ex.what() << endl;
}
```

exception: catching error (anything can be thrown, but usually derived class of exception)

allows program to run after exception

```
catch (...) {
}
```

catch everything

can have multiple catch clauses

unwind call stack

destructors still called

```
const char* what() const noexcept { return "exception"; }
```

```
assert (n < 200);
```

condition you want to be true

for programming

```
#define NDEBUG
```

or `-DNDEBUG`

Questions
- why introduce DP?
- does multithreading increase efficiency
- locks → .join for threads
- what does race condition

Multithreading

thread myThread (function())

myThread.detach(); → allows thread to exist after main has terminated

myThread.join(); → will not execute code until thread finishes running

multiple threads can share the same variables

race condition

mutex myMutex;

myMutex.lock();

myMutex.unlock();

lock_guard<mutex> myLock(myMutex); → when lockguard goes out of scope
↳ unlock