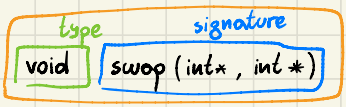
Binding -> associate properties with names

Different properties different binding times

1

static binding	dynamic binding
occurs before runtime	during execution
unchanged during execution	chan change during exe.



2

declaration -> introduces identifier, binds static type (as many as I want)
prototype+ specifiers

definition -> implements identifier (once)
declaration+ function body

inline -> tell compiler to replace function call with its code

constexpr -> evaluate function at compile time

noexcept -> function shouldn't throw exceptions

nodiscard -> return value should be used

3

stack -> stores local variables, function arguments, control information

heap -> stores objects with dynamic lifetime

static memory -> fixed size, contains read only code and static data

storage duration -> variable's property, defines rule of creation and destroying

- Automatic -> from start to end of scope
- static -> from start to end of program
- thread -> " " thread
- Dynamic -> from allocation request until deallocation request

lifetime -> period of time in execution in which variable can be referenced consistently
from when memory allocated to when released

4

data members -> attributes

member functions -> functions

initializers list -> more efficient bcz uses copy constructor instead of assignment operator

copy -> constructs and initializes memory

assignment -> assign to previously assigned object

5

ownership -> who has autority over the lifetime of an object
responsible for cleanup

shared pointer -> retains shared ownership, counts how many references, deletes object when last reference destroyed

unique pointer -> exclusive ownership copy constructor deleted
can return a unique_ptr bcz of copy elision

copy elision -> compiler optimisation, prevents additional copies

lvalue -> expression with accessible address

rvalue -> no address

xvalue -> has inaccessible address (usually object near end lifetime)

& -> lvalue reference

&& -> rvalue reference

std::move -> changes lvalue to xvalue, used to pass unique_ptr as parameter

---

6

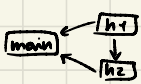
class -> by default private

struct -> by default public

class interface -> public data members and member functions
should construct an Abstract Data Type
hides implementation details

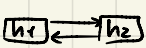
#include dangers:

breaking one definition rule



solved with include guards #ifndef CLASS
#define CLASS

cyclic includes



solved with forward declaration

const -> indicates member function doesn't modify object

mutable data member -> can be modified by const member function (logical const)

static data member -> part of the class but not of the objects (only one copy in mem)

static member function -> can be called without object of the class

Association

- **aggregation** -> X has Y
- **composition** -> X is part of Y

Generalization -> X is a Y

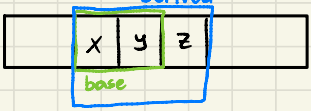
Hiding -> derived redefines fun. of base, then all f. of base with that name are hidden in derived

member lookup:

1) look in class scope of this
2) if not found, go up in chain
3) stop when found

memory layout

stack



Slicing:

Base b
Derived d=b
copy constructed from base B

construction -> from base to derived

destruction -> from derived to base

polymorphism -> pointer to base can point to derived (can only use stuff of Base, but uses override)

vtable -> for each virtual function in base, make an entry with address of function
Which function to execute at runtime
check vtable when have to execute virtual function.

Liskov substitution principle -> code that uses pointers to base must be able to use pointers to derived without knowing it.

type conformity -> derived classes should respect base class' pre/post conditions and invariants

principle of closed behaviour -> inherited functionality should respect derived pre-post conditions and invariants

8

Single responsability principle -> every class one responsability

Open-closed principle -> open for extension, closed for modification

Low coupling -> degree to which the different classes depend on each other

high cohesion -> degree to which elements of a class belong together

Creational patterns:

factory -> given variable construct objects with common interface

abstract factory -> create objects with multiple implementations. Factories have common methods do different stuff

Singleton -> have exactly one object of a class. Make constructor private.
Static member containing singleton. Static method creates when first called, after return already created

Structural patterns:

Adapter -> use similar classes with different interface, adapter that wraps class

Bridge -> want to specialize a class, adding stuff heavy
switch from inheritance to object composition

Facade -> classes form a complex subsystem, facade class manages the objects, calls are simpler

Behavioral patterns:

command -> problem different classes that perform same action on different targets
solution: implement class that represents action

observer -> classes that are interested in state of an object and need updates when it changes.
add common interface for observers, store them in a list (update it on change)

visitor -> complex data structure consisting of multiple interconnected classes.
add functionality that goes through the entire data structure
Add visitor class with visit method for every class, add accept method to these classes that take a visitor as parameter, then execute that visitor's visit method that corresponds to the class.

MVC:

model -> handles back-end

view -> handel visual front-end

controller -> handles user input

```
void print(){
    cout<<"hi";
}
int main(){
    void (*f_p)() = print;
    f_p()
}
```

pointer to function — name of function — type — no parameters

- complete assignment must be correct
- to start a thread: thread th { functionPtr, parameters } (lambdaFun)

Lambda function -> anonymus function

[capture](parameters) -> return type { definition }

capture: & all external by reference; = " " by value; or single variables

for_each -> applies function to every element of a container

std: all_of (v.begin, v.end, func/lambda) — uses this return value
any_of
none_of

- can store a lambda function as a function pointer (only if it doesn't catch anything)
- destructing thread while running -> runtime error

std::thread detach -> detach thread from variable, which can be destroied safely — dangling reference if captured by reference

std::thread join -> wait for thread to end

promise/future -> one thread owns one, another thread another

```
std:: promise<int> p;
std:: future<int> f=p.get_future();
std:: thread th(&calc, std::move(p));
th.join
int result=f.get();
```

Data race -> 2 instructions from different threads access the same memory without synchronization (at least one is a write)

Race condition -> semantic error, timing or order of events effects correctness.

atomic -> solves data race std::atomic<int> a{0};

mutex -> synchronization primitive, if thread locks mutex all others have to wait until unlock
a deadlock can occur

volatile -> its value may change spontaneously (from another thread)

thread_local -> each thread has its copy

generic programming -> abstracting algorithms from concrete ones

template functions (cannot be virtual)

```
template <Typename t>
T sum (T a, T b){
}  return a+b ;
```

compiler doesn't generate binary code with only definitions. Must be instanciated

implicit instantiation -> when called with called type

explicit instatiation -> forces generation of code

- type must be deducible at compile time, otherwise won't compile (can provide explicit type to bipass)
- can overload with specific implementation
- can use auto keyword for return and parameter type
- can also use templates for data members
- can have default type if not specified

universal reference (forwarding) -> `template <typename T> void f(T&& t);` rvalue reference to a template param. can be deduced as either rvalue or lvalue reference

std::forward -> preserves value category
copies lvalue, moves rvalue

parameter pack -> zero or more templates and parameters