OOP-345 FARDAD

* References → New names for already existing thing but the target is the same.

* Templates → If the logic applies to 50 different types, we don't write 50 different functions, we write one function using template and leave it to the compiler to figure it up.

* namespaces → a scope which we create to prevent conflict between names.

* Linkage:

Example of External Linkage:

Making a variable global by using the keyword 'extern'

```
//Module.cpp
const int MinPassGrade = 50; // global scope
const int MaxStdNoLength = 9; // file scope
```

```
//Module.h

extern const int MinPassGrade;
```

⇒ First you make the variable appear in file scope then make it global using extern in header file, this is known as external linkage.

⇒ Internal Linkage is like a file scope that is within the translation unit.

* Anything that you write starting with "#" tells the compiler what to do.

* ~~Static Variables~~

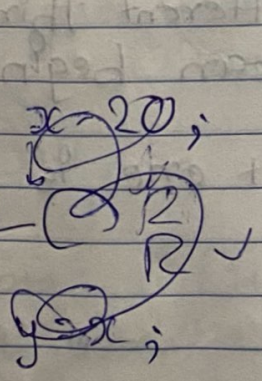
* Statically allocated Memory

```
int a;
int b[1000];
```
//gets loaded in the memory within your executable

* Dynamically

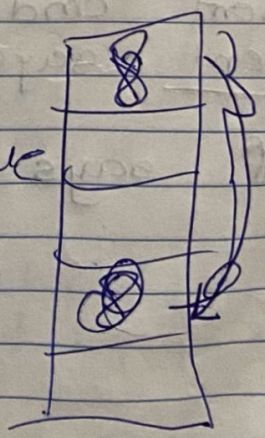
```
int* a;
a = new int[1000]
delete[] a;
```

* Static Variables are variables that have global life time but limited scope.



Copy & Lvalue

Move && Rvalue



* A non-static member function can call a static variable but a static member function cannot call a non-static variable.

* ~~bz 017~~
int b = 017;
int c = 0x17;

017 is not 17!
it is 15 in OCT base.

→ sizeof() is operator (within the operator structure in C++) and not a function.

While writing variables in a struct

→ Rule: Start with the largest one and sort it to the smallest one, by this way you are going to consume the least amount of memory.

→ Alignment and sizeof are different things.
alignment says where you can begin
sizeof says where the next one is going to start.

→ After compiling the code, if it does not return 0 to the OS then the code has some errors that are need to be fixed.

⇒ #define → It goes and searches the code and replace and then changes the values before it starts compiling.

∴ DONT USE #define except if you are defining macros

→

```
int *p;
int *q;
```

sizeof(p) and sizeof(q) could be the SAME as all addresses have the same size, it is like an envelope, if you send it to a big building or a small house the address size will be the same, it is the TARGET that is different.

BUT,

```
p++;  //it will be added by 4.
q++;  //it will be added by 8.
```

int b{5};
is same as
int b={5};

→ set them all to 0 → int c[400]={};

This is for INITIALISATION only!

* Range based FOR loop

```
int a[5]{10,20,30,40,50};
for(int& elm: a) {
    cout << elm << endl;
}
return 0;
```

Output:

10
20
30
40
50

* UNIONS when you are dealing with union, each member variable of a union is overlapping with the other one which means if you write a value for any variable among all member

variables, the other member variables will be overlapped.

→ x value → Temporary nameless object

B = Name ("XYZ"); // this is x value!
Name ("xvalue/glvalue") = "some value";

x value can be used both as lvalue and rvalue.

→ Templates

→ Overloaded functions always have priority towards templates.

To specialise a particular type to be called, we define template in the following ways

```
templat <>
const char* largest <const char*> (  ){

}
```

OR

```
template <>
Container largest <Container> (  ) {

}
```

→ While calling the function, you can resolve the function calls by writing

→ largest<int>(a,b); //tells the compiler to call the integer version

→ size_t is a C++ thing for anything that is more than 0.

References

→ Reference is an alternative name for the entity defined elsewhere. Any change to reference can also change the original object. The difference between reference and pointer is that references cannot be reassigned to refer to a different object once they are initialized, whereas pointers can be changed to point to different objects. Thus, references are a safer option.

* Two Types of References:

① l-value &
② r-value &&

* Standard Library

① std::ref() → returns an l value reference to its argument.

② std:: move() → returns an r value reference to its argument

NOTE: Their prototypes are declared in <utility> header.

* ONE-D ARRAYS

Type identifier[c]; // allocated on the stack
type* identifier = new Type[n]; // " " " heap

* A class without an identifier is called an anonymous class.

→ Defining an instance of anonymous type

```
struct
{
    char shortName[7];
    char fullName[41];
}name;
```

→ declaring a synonym type (Course)

```
typedef struct
{
    unsigned number;
    char desc[41];
} Course;
```

* MOVE

→ Prototype for a move constructor takes the form
class-name (class-name &&);

→ Prototype for a move-assignment operator takes the form
class-name& operator= (class-name &&);

* Deep Copy and Shallow Copy

→ Deep Copy - Is often used when you want to create a completely independent duplicate of an object.

→ Shallow copy - Is useful when you want to create a new object that shares some or all of its components with an existing object, or when you want to improve performance and efficiency. Changes made to these shared objects are visible in both the original and shallow copy.

* COPY Assignment and MOVE Assignment (with steps)

(A) Copy Assignment -

```
Array &operator = (const Array& src)
{
    //1. check for self-assignment
    if (this != &src)
    {
```

```
//2. clean-up the resource used by the current instance
    delete [] a;

//3. shallow copy
    n = src.n;
    dummy = src.dummy;

//4. deep copy
    a = new int[src.n];
    for (unsigned i=0u; i<src.n; ++i)
        a[i] = src.a[i];
}
    return *this;
}
```

(B) MOVE Assignment

```
Array& operator=(Array&& src)
{
    //1. check for self-assignment
    if (this != &src)
    {
        //2. clean-up the resource used by current instance
        delete [] a;

        //3. shallow copy;
        n = src.n;
        dummy = src.dummy;
```

```
    //4. move the resource parameter into current instance
    a=src.a; //copy address to current object
    src.a=nullptr; //the parameter doesn't have the resource anymore
  }
  return* this;
}
```

* ~~The~~ Static variables

→ The keyword static declares a variable in a class definition to be a class variable.

Ex:

```
class Horse{
    unsigned age; } //they are instance variables
    unsigned id;

public:
    static unsigned noHorses; //class variable
```

↓ They are shared among all instances of a class

→ ~~The~~

* Static functions

→ The keyword static declares a function in a class definition to be a class function.

→ They can only access static data members and static member functions of the class.

→ They are usually used to implement utility functions, factory methods and other class-level operations.

* Structs and Union

→ weakly encapsulated
→ public by default

* Enumerations:

→ User defined type that holds a discrete set of symbolic constants.

→ default type is `int`

→ Ex:

```
enum Color {

    RED,   //0
    GREEN, //1
    BLUE   //2
};
```

* Abstract Class:

It is an incomplete class in the hierarchy that can serve as an interface to an inheritance hierarchy. We cannot create

an instance of an abstract class and we complete its implementation by deriving a new class that adds the missing definitions, we call the completed class a concrete class.

* Polymorphism

→ A polymorphic object in the context of OOP is an object that can represent different classes within a class hierarchy.

Ex:

```
class Shape{
public:
    virtual void draw() const =0;
}

class Circle: public Shape {

    public:
    void draw() const {
                .
                .
                .
                }
}
class Square: public Shape {
    void draw() const {
    }
}
```

```
int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Square();

    shape1->draw(); // calls circle's draw method
    shape2->draw(); // calls square's draw method

    delete shape1;
    delete shape2;

    return 0;
}
```

→ shape1 and shape2 are polymorphic objects that can ~~derive~~ represent objects of any class derived from 'Shape'.

* Three categories of Polymorphism
① Ad-hoc
② Inclusion
③ Parametric

① Pen C++ performs adhoc by overloading a function name for different parameters types.
② Inclusion polymorphism by using the same function signature across different classes in an inheritance hierarchy.
③ Parametric by using the same name for classes or functions that share the same structure.

C++ implements the parametric (or generic) polymorphism through TEMPLATES.

→ This results in the reduction of code duplication.

* TEMPLATES

* Syntax

```
template <parameter>
return-type function-name (...)
{
    //body
}
```

* Parameters

① Type template parameter

template <typename identifier>

② Non-Type Template parameter

is a template parameter that is not a placeholder for a type. Its type is declared explicitly.

template <type identifier>

→ Here, identifier is a non-type parameter and it must be a compile time constant.

③ Template Template Parameters

is a template parameter that is a placeholder for a template.

template <template <parameter> typename identifier>

* Body

```
template <typename T>    //template header
void foo()
{
    T value;
    value = 1.5;          //body
    cout << value;

}

int main()
{
    foo <int> ();         //template instantiation
    foo <double> ();

}
```

Output:

```
1
1.5
```

OOP Continue

* Template Specialization

→ template specialization of a function defines an exception to a template definition of that function.

Example:

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

→ This definition can apply to all fundamental types but not to pointer to those types, e.g. const char* type. we need to create an exception for that.

```
template <> // denotes specialisation
const char* maximum <const char*> (const char* a, const
                                   char* b)
{
    return strcmp(a,b) > 0 ? a : b;
}
```

* Overloading vs Specialization

→ The compiler resolves overloading before instantiating any specialisation.

If we have an ~~function~~ overloaded function

```
const char* maximum(const char* a, const char* b)
{
        return strcmp(a,b) > 0 ? a : b;
}
```

then, the overloaded function will be called instead of specialized template.

* Class Templates

→ Template declaration for a family of classes follows the same rules as a template declaration for a family of functions.

Eg. Normal scenario

```
class Array {
    int a[50];
    unsigned n;
    int dummy;
public:
    . . .
};
```

```
//array.cpp
int main() {
    Array a, b;
    //introducing objects
}
```

with template &

```
template <typename T>
class Array
{
    T a[50];
    unsigned n;
    T dummy;
public:
    - - - -
};
```

```
//array.cpp
int main(){
Array <long> a, b;
}
```

* Non-Type Parameters

→ can receive the size of an array.

```
template <typename T, int SIZE>
class Array
{
    T a[SIZE];
    T dummy;
};
```

```
//array.cpp
int main(){
Array <int,50> a, b;
}
```

* Default Template Parameter Values

→ template declaration for a family of classes accepts default values.

```
template<typename T=int, int SIZE=50>
class Array
{
    T a[SIZE];
    unsigned n;
    T dummy;
public:
    . . .
};
```

```
//array.cpp
int main(){
Array<> a,b;
}
```

* Static data member declarations in a class template

In Notion

* Templates and inheritance

→ A class can be derived directly from a templated family of classes.

```
template<typename T>
class Base {
    T value;
public:
  void set(const T& v) { value = v; }
};
template<typename T>
class Derived: public Base<T> {
public:
    void set(const T& v) { Base<T>::set(v); }
};
```

* Compositions, Aggregations and Associations

→ The relationships between classes in oop aside from inheritance and parametric polymorphism, exhibit different degrees of ownership. These relationships include compositions, aggregations and Associations.

→ Composition is a strong relationship

↓

→ Aggregation is a weaker relationship

↓

→ An association is the weakest relationship

→ If a class is responsible for copying and destroying its resource, then that class is composition, otherwise it is either aggregation or association.

* Compositions

→ implements complete ownership

→ it is a better option than inheritance

→ if the composer object is destroyed, component object is also destroyed

* * * * Example in oop Notes * * * * *

* Aggregations

→ doesn't own the component object as in composition.

→ the component can exist independently of the aggregator.

* Associations

→ is a service relationship.
→ doesnot involve any ownership
→ each type is complete without the related type.

* Expressions

→ Any C++ expression is one of
① lvalue - a locator value that occupies a location in storage.
② xvalue - an expiring value that does occupy location in storage
③ prvalue - a value that doesnot occupy a location in ~~object~~ storage.

→ lvalue operands

The operands associated with the following three operators must be lvalues
① &
② ++
③ --

The left operands associated with the assignment operators must be lvalues.
① =
② +=
③ -=
④ *=
⑤ /=
⑥ %=

→ Expressions are divided into six operand related classifications

① primary

② postfix expressions
→ Subscripting operator : '[]'
→ Member Selection
  → Direct Selection '.'
  → In Direct Selection '→'
→ Postfix Increment '++'
→ Postfix Decrement '--'
→ Constrained cast operators
  → static_cast operator (safe type conversion)
  → reinterpret_cast operator (low-level, unsafe type conversions)
  → const_cast operator

③ prefix expressions

→ prefix increment
→ prefix decrement

④ unary expressions

→ sizeof()
→ logical negation (!)
→ bitwise operator (~)
→ Arithmetic Negation (-)
→ Arithmetic Plus (+)
→ Address-of operator (&)
→ Indirection operator (*)
NOTE: The result of *&x is x

→ alignof() operator
→ decltype() specifier
→ noexcept() operator
→ throw operator

(5) Binary Expressions &
→ Arithmetic
  → Multiplicative
  → Addition
  → Subtraction
→ Bit-Shifting ('<<', '>>')
→ Relational
  → less than (<, <=)
  → Greater than (>, >=)
  → Equality (==)
  → Inequality (!=)
→ Bit-wise and (&&)
→ Bit-wise or (||)
→ Bit-wise exclusive or (^)
→ Logical
  → true
  → false
→ Assignment expressions
  → simple assignment (=)
  → compound assignment (+=, -=, *=, /=, %=, >>=, <<=, ~=, &=, |=, ^=)
→ Sequential expressions (,)
→ Mixed-type binary expressions

(6) Ternary Expression &
→ is a conditional expression. (?) identifies the expression as a selection construct, (:) seperate the choices.

* Functions &

→ Well designed functions exhibit high cohesion and low coupling.

→ High cohesion refers to focus on a single task and low coupling refers to a minimal interface with other functions.

→ Nested function types can access variables within the scope of their host function. These types are known as closures.

→ C++ supports closures in the form of lambda expressions.

→ A function type can have either external or internal linkage.

Function with external linkage is visible outside its translating unit while function with internal linkage is invisible outside its translation unit.

→ The default linkage for a function type is external.
The keycoord extern is redundant.
To specify internal linkage, we preface the function declaration with the keycoord static.

→ Example in notion

→ Recursive Function is a function that calls itself from within its own body.

→ Function Pointers

A function pointer holds the address of a function type.

Array of Pointers to functions

If several functions share the same return types and the same ordered set of parameter types, we may store their addresses in an array of pointers to functions.

→ Function Objects

is an object-oriented representation of a function. It is also called a functor

Function wrapped in a class, so that it can be available like an object.

→ Lambda Expressions

A function object that is only used in a local area of an application. (within a function) can be represented by a lambda expression.

Does not require an identifier

Lambda expression takes the form:

```
[capture-list] (parameter-declaration-clause) -> optional return type
{
    // function body
}
```

Capture List:

is the mechanism for passing all non-local variables to the body of the lambda expression.

→ Empty List - []

denotes no capture

→ Capture by value - [=]

denotes capture by value

→ Capture by reference - [&]

denotes capture by reference