Introduction

Introduction to C++

Software interacts with hardware via an operating system, which makes sure your trash code doesn't mess up the hardware. Software applications include Safari, VS Code, Notion.

Hardware includes Central Processing Unit (comprised of arithmetic logic unit), I/O devices, and main memory.

Structure of a C++ Program

There exist `<<` and `>>` operators. There is also the `using namespace std;` line, which is a container for the standard library.

Data Types

1. Integers,
2. Real numbers,
3. Characters,
4. Booleans, and
5. Strings

Functions

In the line `int fact = factorial(n);` we pass the variable `n` by value. This means we passed a copy of `n`, and if is changed within the function, it won't be changed in `main`. This is called **pass-by-value**.

In C++, there is **pass-by-reference** syntax that allows a function to access the original variable with no usage of pointers. A **reference** is an alias, an alternate name, to a variable.

```
swap(&a, &b);
```

Some notes about references:

1. References, once assigned, may not be reassigned.
2. Must be initialized at the declaration.
3. References do not have a separate memory allocation.

Multiple File Programs

To compile a C++ file, run the following command in terminal:

```
g++ main.cpp -o main
```

A multiple file program contains the following files:

1. A `main.cpp` file — the main program containing `main(void)` .
2. Header files like `print.h` or `input.h` which contain function or class declarations.
3. Source files like `print.cpp` or `input.cpp` which contain function or class implementations.

To compile a program with multiple files:

```
g++ main.cpp print.cpp input.cpp -o main
```

What happens if I include the same .h file multiple times?

```
#ifndef A_H
#define
struct A{
};
#endif
```

These are called **header guards**.

Classes

A class is an expansion to a struct in C. Using classes, we can bring together data and function/operations.

> 💡 **Definition** — A class is a user-defined data type. A variable of that class is called an **object**.

Declaring an object of a class is called instantiation.

```
class Student {
  private:
    int ID;
    string name;
  public:
    void setName(string n);
    string getName();
    void print();
};
```

- **Private members** can only be accessed within the class.
- **Public members** can be accessed within or outside the class.

Why Do We Use Classes?

- Hide information related to the class, for e.g. student ID is not available outside of the class — this is called **abstraction.**
- It brings related data and functions together. It organizes the code — this is called **encapsulation.**

Constructors

Constructors are called by default when an object is instantiated. If there is no user-implemented constructors, they are empty functions.

Constructors may not be called explicitly. They are called automatically when an object is instantiated.

It's name is the same as the class name, and it has no return or even return type.

```
Student::Student() {
  ID = 0;
```

```
  name = "";
}

int main(void) {
  Student x;
  Student y[10];
  Student *z; // no constructor called when we make a pointer
  return 0;
}
```

Some notes on constructors:

1. Constructors must be public.
2. Constructors have no return type.
3. Constructors must have the same name as the type.

Multiple Constructors

What if I want to initialize ID with a specific value? We can implement constructors.

When we use the same function name with different arguments, it is called **function overloading**. C++ compiler automatically decides which function to use depending on what arguments are provided.

> 💡 **Important!** If the default constructor `Student()` is not implemented but `Student(x)` is, then `Student x;` will raise an error because the default was not implemented.

Memory

Our computer's memory is broken into four parts:

1. The Heap — Contains dynamically allocated memory (program memory)
2. The Stack — Local variables inside functions
3. Data — Global variables
4. Code — The instructions given in the code

Memory on the stack gets freed when a function returns, all local variables in a function disappear when the function returns or when they go out of scope.

HOWEVER, memory allocated on the heap dynamically must be explicitly freed. If we don't free it, we will have what's called a memory leak.

Whereas in C we had `malloc` / `calloc` and `free`, in C++ we have `new` and `delete`.

Integer	Array
`int *pNum = new int;`	`int *arr = new int[10];`

Integer	Array
`delete pNum;` `pNum = nullptr;`	`delete []arr;`

Regular variables such as integers, bools, etc. get automatically freed when the `main` function returns, but what about class functions?

Destructors

A destructor is automatically called when an object is either destroyed, or goes out of scope. If not defined, it is empty. If we use dynamically allocated memory in our class, we MUST use a destructor to free up that space.

```
class Student{
  public:
    ~Student();
};

Student::~Student() {
  if (grades != nullptr) {
    delete []grades;
    grades = nullptr;
  }
}
```

Some notes on destructors:

1. The destructor is called when the object foes out of scope in a function.
2. The destructor is called when an object is destroyed on the heap.
3. The destructor can be called recursively if an object has a pointer to another dynamically allocated object.

Dynamic Memory Allocation

Some different ways if we want to have an array that stores data:

1. Fixed-size array — allocated int arr[4];
2. Variable-sized array —

```
int size;
cin >> size;
int arr[size];
```

3. Dynamically allocate memory for the array —

```
int size = 7;
int *arr = new int[size];
```

Input/Output

File Input/Output

Some ways to take input and produce output:

1. Standard input-output using `cin` and `cout`, which are objects of `iostream`.
2. File input-output using `ifstream` and `ofstream`, which are objects from `fstream`.
    a. If a file doesn't exist, it will be created. If it already exists, then its content will be erased and overwritten.

```
#include <fstream>
using namespace std;
int main(void) {
  ofstream outfile("myfile.txt");
  string name = "We are engineers!";
  outfile << name;
  outfile.close();

  return 0;
}
```

To append to a file, we can use:

```
ofstream outfile("myfile.txt", ios::app);
```

Input from a File

```
#include <fstream>
using namespace std;

int main(void) {
  ifstream inputFile;
  inputFile.open("myfile.txt") // OR ifstream inputFile("myfile.txt");

  int num1, num2, num3;
  inputFile >> num1 >> num2 >> num3;
  inputFile.close();
```

```
  return 0;
}
```

Where to Find the File?

- Absolute path — `inputFile.open("Users/arnavpatil/ece244-labs/lab/myfile.txt");`
- Relative path — `inputFile.open("lab/myfile.txt");`
- Relative path — `inputFile.open("./myfile.txt");`
- Current directory — `inputFile.open("myfile.txt");`

Buffering

Output is not immediately written to a file, it gets written in chunks. This is called buffering. Why buffering? Writing into a buffer is faster, and writing into the file (from the buffer) in chunks optimizes resources as well.

We can force output using `outputFile.flush();` or `outputFile << endl;` Input streams are also stored in a buffer before they get read by the program; reading happens until it encounters a delimiter character, usually a space, a tab, or a newline.

Take the following code:

```
int x, y;
cin >> x >> y; // we input 13.7
```

What happens? `cin` will read `13` into `x`, but then it will encounter `.` and the `cin.fail()` flag will be raised silently. The buffer and `y` will be unaffected but all other `cin`'s in the program will fail.

What should we be doing before taking input?

1. Detect: Check if the fail flag is raised — **ALWAYS!**
2. Handle: If it is raised, handle the error.

Detecting Errors — What Can Go Wrong?

1. File doesn't exist — `inputFile.fail()` set to `true`
2. Input type is incorrect — `cin.fail()` and `inputFile.fail()` set to `true`
3. Reached the end of the file — `cin.eof()` or `inputFile.eof()` set to `true`
    a. This does not set off the `.fail()` flag.

What is `cerr`? It's an output stream, similar to `cout`. It is unbuffered and the output will appear immediately on the console/terminal.

Handling Errors

- `cin.clear()` will clear the fail condition so `cin.fail()` and `cin.eof()` are back to `false`.
- `cin.ignore(int n, char c)` will discard `n` characters or up to the character `c`, whichever comes first.

💡 We must use the `.clear()` first! If we try to `.ignore()`, it will fail.

String Streams

String streams are most helpful when the inputs are line-oriented.

```
#include <sstream>
#include <string>

int main(void) {
  int ID;
  string name;
  string inputLine = "1001 Joe"
  stringstream ss(inputLine);
  ss >> ID;
  ss >> name;
  cout << "Name: " << name << endl << "ID: " << ID << endl;
  cout << ss.str(); // converts a stringstream into a string for printing
  return 0;
}
```

The `getline()` Function

```
string inputLine;
getline(cin, inputLine); // we store an entire line
stringstream myStream(inputLine);
myStream >> ID;
myStream >> name;
```

Overloading Operators

Consider this class:

```
class Complex {
  private:
    double real;
    double img;
  public:
    Complex() {real = 0.0; img = 0.0;}
    Complex(double r, double i) {real = r; img = i;}
};
```

If we had two complex numbers `x` and `y` that we wanted to add, we can't just do `z = x + y;` because the `+` operator is not defined for `Complex` objects. However, we can use **operator overloading** to do so. This is because `x + y` and `x.operator+(y)` are the same thing.

There are two operators that we will need to implement: `+` and `=`.

`operator+()`

```
Complex Complex::operator+(const Complex &rhs) {
  return Complex(real + rhs.real, img + rhs.img);
}
```

Three notes:

1. Passing by value would be memory inefficient as the program would have to make a copy of it. By instead **passing by reference**, we will not create a copy and conserve some memory.
2. We pass in the object as a constant with `const` to avoid making any accidental changes to the right hand side object's data values.

`operator=()`

There is a default `operator=()`, however, it wouldn't work for our Complex class because we need to allow for chain assignments, i.e., `z=x=y;`.

We use the this keyword. this is a pointer to the object on which the function is being invoked, for e.g., the `z` in `z=x`.

```
Complex &Complex::operator=(const Complex &rhs) {
  real = rhs.real;
```

```
  img = rhs.img;
  return *this;
}
```

Exercise: Implement `operator==()`

```
bool Complex::operator==(const Complex &rhs) {
return (real == rhs.real && img == rhs.img);
}
```

Overloading `operator<<()`

Reminder that `cout` is an object of `ofstream` class, meaning it cannot be a member function of `Complex`, since the LHS object is of type `ostream`. However, it also can't be a non-member function as it has to access private members of a function.

SOLUTION: We can make operator<< a friend function, meaning it can access private members of an object and LHS can be a non-Complex type object.

```
class Complex {
  public:
    friend ostream &operator<<(ostream &os, const Complex &x)
}

ostream &operator<<(ostream &oc, const Complex &x) {
  os << "(" << x.real << "," << x.img << ")";
}
```

`cout` and all other streams can't be passed or returned by value; only by reference, as their copy constructor is deleted. → explained later

Copy Constructor

Recall that `cout` and other streams can't be passed or returned by value, only by reference.

A **copy constructor** is a constructor used to create a copy of an existing object. It is called when:

1. `Student a(b);` — b is an object of Student
2. `Student a = b;` — create and initialize on some line
3. Passing an object by value to a function
4. Return an object by value from a function

By default, every class has a copy constructor that copies all data members.

```
class Student{
  private:
    string name; int ID;
  public:
    Student(const Student &other) {
      name = other.name;
      ID = other.ID;
    }
};
```

Must pass by reference or else we will hit a compile-time error. If passed by value, copy constructor will be called again — infinite recursion. Constructors have no returns!

Problem: What Happens When Data Members are Pointers?

Solution: We use a **deep copy**!

```
Mystring::Mystring(const Mystring &other) {
  buf = new char[other.len + 1];
  strcpy(buf, src buf);
  len = src.len;
}
```

The Rule of Three

> 💡 If a class requires one of the following, it almost certainly requires all three.

1. A user-defined destructor
2. A user-defined copy constructor
3. A user-defined assignment operator `operator=()`

Notes on Confusing Points

Similarities and Differences Between

Copy Constructors

- `Student X(Y);`
- `Student X = Y;`

`Operator=()`

- X = Y;
- X.operator=(Y);

- `Student *p = new Student(Y);`
- Pass by value
- Return by value
- Must pass by reference, if you don’t, hit a compile-time error
- Argument better passed as a `const` to prevent changes
- Default is given
- Doesn’t return anything.

- Better pass by reference to avoid calling the copy constructor
- Argument better passed as a `const` to prevent changes
- Default is given
- Return by reference `return *this;` to allow chained assignments

Data Structures

Linked Lists

To store and organize data of a similar type, we used arrays. An issue is that they are not flexible, if we need to increase the size, we need to allocate new memory space to copy data into. This is where we use linked lists! They're extendable and flexible.

A linked list is a collection of nodes that get linked together using pointers. In C, each node and the linked list was a `struct`. Now we can use classes to bundle together operations and data values into objects.

Definition of a linked list that we will be using for the entirety of this section.

```
class Node{
  private:
    int data;
    Node *next;
  public:
    Node() {data = 0; next = nullptr;}
    Node (int d) {data = d; next = nullptr;}
    Node (int d, Node *n) {data = d; next = n;}
    ~Node() {delete next;}
    int getData() {return data;}
    Node *getNext() {return next;}
    void setData(int d) {data = d;}
    void setNext(Node *n) {next = n;}
};
```

Stacks

Stacks have two operations: pop() and push(), which happen LIFO (last-in, first-out):

1. `pop()` — Remove the node that was most recently inserted
2. `push()` — Insert a node at the head

Class implementation of a stack:

```
class Stack{
  private:
    Node *head;
  public:
```

```
    Stack() {head = nullptr;}
    ~Stack() {delete = head;}
    void push(int d) {
      Node *p = new Node(p, head);
      head = p;
    }
    int pop() {
      if (head == nullptr) return -1;
      Node *p = head;
      int d = p->getData();
      head = p->getNext();
      p->setNext(nullptr);
      delete p;
      return d;
    }
};
```

Some special cases to think of:

1. Does it work when the stack is empty?
2. Does it work if the stack has just one node?

Queues

Queues have two operations: enqueue() and dequeue(), which happen FIFO (first-in, first-out).

- `dequeue()` removes the first node that was put into the list
- `enqueue()` puts a node at the end of the list

```
class Queue{
  private:
    Node *head;
    Node *tail;
  public:
    Queue() {
    head = nullptr;
    tail = nullptr;
    }
    ~Queue() {
      delete head;
    }
    void enqueue(int d) {
      Node *p = new Node(d, nullptr);
```

```
      if (tail != nullptr) tail->setNext(p);
      tail = p;
      if (head == nullptr) head = p;
    }
    int dequeue() {
      Node *p = head;
      head = p->getNext();
      if (head == nullptr) tail = nullptr;
      int d = p->getData();
      p->setNext(nullptr);
      delete p;
      return d;
    }
};
```

Ordered Linked Lists

Ordered linked lists have four basic operations:

1. Insert (in sorted order)
2. Search
3. Delete
4. Copy

```
class List{
  private:
    Node *head;
  public:
    List() {head = nullptr;}
    ~List() {delete head;}
    void insertData(int d);
    bool dataExists(int d);
    bool deleteData (int d);
    List (const List&); // copy constructor
    List &operator=(const List&);
};
```

Searching a Linked List

```
bool List::dataExists(int d) {
  Node *p = head;
  while (p != nullptr && p->getData() <= d) {
```

```
    if (p->getData() == d) return true;
    else p = p->getNext();
  }
  return false;
}
```

Note: we have added the `p→getData() ≤ d` expression because (if the list is sorted) then we know we won't find the value after that Node's value.

Inserting into a Linked List (General Case)

We want to search for the first node with data greater than the data we want to insert, and add our inserted data before that node.

```
void List::insertData(int d) {
  Node *n = new Node(d);
  Node *p = head, *prev = nullptr;
  if (p == nullptr) head = n; // if the list is empty
  while (p != nullptr && p->getData() < d) {
    prev = p;
    p = p->getNext();
  }
  // either p->getData() > d or p == nullptr
  n->setNext(p);
  if (prev == nullptr) head = n;
  else prev->setNext(n);
  return;
}
```

Some special cases to consider:

1. The list is empty
2. The list has only one node
3. We insert at the tail
4. We insert at the head

Deleting Data in a Linked List (General Case)

```
void List::deleteData(int d) {
  Node *p = head, *prev = nullptr;
  while (p != nullptr && p->getData() {
    if (p->getData() > d) return;
    prev = p;
    p = p->getNext();
```

```
  }
  // p is null or p->getData() == d
  // not found
  if (p == nullptr) return;
  // delete from the front
  if (prev == nullptr) head = p->getNext();
  // delete at the middle or the tail
  else prev->setNext(p->getNext());
  // delete p
  p->setNext(nullptr);
  delete p;
}
```

Destructor

```
List::~List() {
  delete head;
}
```

Copy Constructor

Creates an object from an existing one. The default copy constructor does a shallow copy, whereas we need to do a deep copy.

- Copy one node at a time.
- p to iterate original list, np to build new list.

```
List::List(const List &original) {
  Node *p = original.head;
  Node *np = nullptr;

  head = nullptr;
  while (p != nullptr) {
    Node *n = new Node(p->getData(), nullptr);
    if (np == nullptr) head = n;
    else np->setNext();
    p = p->getNext();
    np = n;
  }
}
```

`Operator=`

Idea is similar to copy constructor, except list may not be empty.

```
List &List::operator=(const List &original) {
  if (&original == this) return *this;
  if (head != nullptr) {delete head; head = nullptr;}

  Node *p = original.head;
  Node *np = nullptr;
  while (p != nullptr) {
    Node *n = new Node(p->getData(), nullptr);
    if (np == nullptr) head = n;
    else np->setNext();
    p = p->getNext();
    np = n;
  }
  return *this;
}
```

Recursion

Recursion is a programming technique that solves a problem by breaking it into a smaller problem repeatedly until it can be solved easily. Solutions of smaller problems can be combined to form the solution of the original bigger problem.

Example: write a function that gets the factorial of `n` recursively

1. Recursively get the factorial of `n - 1`
2. Until we reach the terminating case: `1! = 1`
3. Combine solutions to get solutions of the bigger problem.

```
int factorial (int n) {
  if (n == 1 || n == 0) return 1;
  else return n * factorial(n-1);
}
```

We need to communicate to the next recursive function call from where we should start adding and till where.

Backtracking

Exploring different routes and going back when they lead to a dead end.

Example: 2D maze

2D array called Maze, which has the map the maze. Spaces mean there is a path, and x's mean there is a wall. We want to leave a trail of stars `*` showing a path from 🦆 to 🍞.

Idea: If `maze[i][j]` is a valid step the 🍞, then one of `maze[i+1][j]` or `maze[i-1][j]` or `maze[i][j+1]` or `maze[i][j-1]` is also a valid step to 🍞.

Procedure:

1. We start with location of 🦆: `maze[i][j]`
2. Is 0≤i<4 and 0≤j<4?
3. If yes, check if `maze[i+1][j]` is a valid step to 🍞.
4. Or if `maze[i-1][j]` is a valid step
5. Or is `maze[i][j+1]` is a valid step
6. Or is `maze[i][j-1]` is a valid step

7. If yes, `maze[i][j] == '*'`
8. If no, nothing changes.

```
bool solveMaze(int row, int col) {
  if (row < 0 || row >= height || col < 0 || col >= width) return false; // outs
  if (maze[row][col] == 'E') return true; // reached exit
  if (maze[row][col] != ' ') return false; // reached a wall 'X' or path again '
  maze[row][col] = '*';

  if (solveMaze(row+1, col)) return true;
  if (solveMaze(row-1, col)) return true;
  if (solveMaze(row, col+1)) return true;
  if (solveMaze(row, col-1)) return true;
  // at this point none of the four directions were a valid step to :bread:
  maze[row][col] = ' ';
  return false;
}
```

Recall: we used linked lists to have a more dynamic and flexible data structure to share information. However, the way a data structure is organized can speed up or slow down the run time of some operations like searching, inserting, and deleting nodes. Trees is another data structure that shares for different purposes.

Properties of Trees

- Have nodes, and edges connect nodes,
- Have no cycles,
- Have 1 parent, multiple children,
- Have 1 root (empty trees have no root).

Binary Trees

- Root is at the top and leaves are at the bottom.
- Each node stores some data.
- Each node has at most 2 child nodes, and one parent (except root).
- An edge links a node to it's children.
- Nodes with no children are leaf nodes.

Binary Trees

Binary search trees (BSTs) are a variant of tree structures.

- All nodes in the left subtree have values < value of node, and
- All nodes in the right subtree have values > value of node.
- The minimum value is in the left most node, and
- The maximum value is in the right most node.

#of comparisons = height of BST = log(n)

Implement BST Node

```
class BSTNode {
  private:
    int value;
    BSTNode *left, *right;

  public:
    BSTNode(int v) {value = v; left = right = nullptr;}
    ~BSTNode() {delete left; delete right;}
    int getValue() {return value;}
    BSTNode *getRight() {return right;}
    BSTNode *getLeft() {return left;}
    void setRight(BSTNode *r) {right = r;}
    void setLeft(BSTNode *l) {left = l;}
}
```

Implement BSTree

```
class BSTree {
  private:
    BSTNode *root;
    bool searchNode (int v, BSTNode *n) {
      if (!n) return false;
      else if (n->getValue() == v) return true;
      else if (n->getValue() > v) return search(v, n->getLeft());
      else return search(v, n->getRight());
    }
```

```
  public:
    BSTree() {root = nullptr;}
    ~BSTree() {delete root;}
    BSTNode *getRoot() {return root;}
    bool search(int v) {return searchNode(v, root);}
}
```

Inserting into a BST

```
void BSTree::insertHelper(int v, BSTNode *n) {
  if (n->getValue() == v) return;
  else if (v < n->getValue()) {
    if (!n->getLeft()) n->setLeft(new BSTNode(v));
    else insertHelper(v, n->getLeft());
  } else {
    if (!n->getRight()) n->setRight(new BSTNode(v));
    else insertHelper(v, n->getRight());
  }
}

void BSTree::insert(int v) {
  if (!root) root = new BSTNode(v);
  else insertHelper(v, root);
}
```

Printing in Order

```
void BSTree::printInOrderHelper(BST Node *n) {
  if (!root) return;
  printInOrderHelper(n->getLeft());
  cout << n->getValue() << " ";
  printInOrderHelper(n->getRight());
}

void BSTree::printInOrder() {
  return printInOrderHelper(root);
}
```

Deleting a Node

To delete a node in a BST, we have to make sure we maintain its properties.

1. Find the node
2. If the node has no children, delete the node and update the parent node pointer.
3. If the node has only one subtree, make the parent node point to the parent of the subtree.
4. If the node has two subtrees, replace the node data with the minimum in the right subtree, then delete the node with the min value in the right subtree.

```
BSTNode *BSTree::deleteNode(int v, BSTNode *node) {
  if (!node) return nullptr;
  if (v < node->getValue()) node->setLeft(deleteNode(v, node->getLeft()));
  else if (v > node->getValue()) node->setRight(v, node->getRight()));
  else {
    if (!node->getLeft()) {
      BSTNode *temp = node->getRight();
      node->setRight(nullptr);
      delete node;
      return temp;
    } else if (!node->getRight()) {
        BSTNode *temp = node->getLeft();
        node->setLeft(nullptr);
        delete node;
        return temp;
    } else {
      BSTNode *temp = minValue(node->getRight());
      node->setValue(temp->getValue());
      node->setRight(deleteNode(temp->getValue(), node->getRight());
    }
  }
  return node;
}
```

Inheritance

One of the pillars of object-oriented programming is inheritance. Inheritance is a process in which a class acquires all properties and behaviour of the parent class. This allows programmers to extend/improve existing classes without modifying the code of these classes.

Example: in `person.h` file

```
class Person {
 private:
   string name;
   int age;
 public:
   Person() {name = ""; age = 0;}
   Person(string n, int a) {name = n; age = a;}
   void setName(string n) {name = n;}
   void print() {
     cout << "Name: " << name << endl;
     cout << "Age: " << age << endl;
   }
};
```

Let's say, after implementing the `Person` class, we want to implement a `Student` class, which is more specific. It has data members `name`, `age`, and `ID`. We want the `print()` member function to print all of the data members, and we also want `setName` to set the name of the `Student`.

To avoid writing the `Student` class from scratch or copy-pasting the `Person` class and modifying it, which requires we understand `Person` very well, we will inherit `Person` to reuse some code.

We call `Person` the base class, and `Student` the desired class.

```
#include "Person.h"

class Student: public Person {
  private:
    int ID; // name and age included but not accessible
  public:
    Student() {ID = 0;}
    void setNameID(string n, int d) {
      Person::setName(n);
      ID = d;
    }
```

```
    void print() {
      Person::print();
      cout << "ID: " << ID << endl;
    }
};
```


An overridden method like `print()` is not the same as an overloaded method. Overloaded functions have different argument and/or return types. `Student`, inherited from `Person`, has all data and function members except for overridden `print()` function. `Student` can't access private members of `Person`.

In `main.cpp` file,

```
#include "Person.h"
#include "Student.h"
using namespace std;

int main() {
  Person p("Joe", 23);
  Student s;
  return 0;
}
```

1. The type of `s` is `Student` and `Person`.
2. The type of `p` is `Person`.

What if I want to call a different Person constructor when I create a Student object?

```
class Student : public Person {
  private:
    int ID;
  public:
    Student(string n, int a, int d):Person(n, a) {
      ID = d;
    }
};
```

Data Protection

Protected data and function members are inherited and accessible to derived classes but not to all classes (somewhere between and private).

```
class Person {
  protected:
    int age; string name;
};

class Student : public Person {
  private:
    int ID;
  public:
    Student(string n, int a, int d) {
      Person::name = n;
      Person::age = a;
      ID = d;
    }
}
```

Important! We DO NOT inherit:

1. Constructors (you can call them)
2. Copy constructors (should make your own) ⬇️
3. Operator=
4. Destructors

Pointers to Dynamic Memory in Derived Classes

When a class has members that point to dynamically allocated memory, we should have our own operator=, copy constructor.

We have to explicitly call the copy constructor of Person, otherwise, e call the default constructor.

When the Student object then is destroyed, we call the destructor Student first, then of Person.

Dynamic vs. Static Binding

```
class Project {
  protected:
    int width, length;

  public:
    void set(int w, int h) {width = w; height = h;}
};

class Rectangle : public Polygon {
  public:
```

```
    int area() {return Polygon::width * Polygon::height;}
};

class Triangle : public Polygon {
  public:
    int area() {return Polygon::width * Polygon::height / 2;}
};
```

Virtual Functions

Problem: We can't access members of a derived object if the pointer pointing to it is of `Base*` type.

Answer: If a function is declared as a virtual function in base class, and redefined/overridden in the derived class, a call to that function through a `Base*` pointer will invoke the function depending on the type of object (not pointer).

```
class Polygon {
  protected:
    int width, height;
  public:
    void set(int w, int h) {width = w; height = h;}
    virtual int area() {return 0;}
};
```

Then, when we say `p1→area()`, it will invoke the area function of what p1 is pointing to, i.e., Rectangle = 12

If we were to remove virtual, `p1→area()` will return `0` .

Non-virtual functions are invoked depending on type of pointer — this is known at compile-time **(static binding)**.

Virtual functions are invoked depending on type the pointer points to — this is known at run-time **(dynamic binding)**.

If only one function should be virtual, it is the destructor. Because if it wasn't virtual, the destructor of polygon will be called, and the length won't be freed.

Problem: This necessitates that we implement the area() function in Polygon. We may need Polygon class to exist with some functions, but never implement them as they won't be needed.

```
class Polygon {
  protected:
    int width, height;

  public:
    void set(int w, int l) {...}
```

```
    virtual int area() = 0; // pure virtual function
};
```

A class with a pure virtual function is called an **abstract class**. You cannot instantiate an object of an abstract class.

```
Polygon p; // incorrect as p is of an abstract class

Rectangle r;
Polygon *pr = &r;
cout << pr->area() // invokes area() of Rectangle
```

Hash Tables

Hash tables provide average performance on search/insert/deletetion

- Very large arrays
- Each key maps to a unique index
- Hash function `h(k)` maps key to an array index

Use cases include databases and caching, where quick data retrieval is critical.

Problem: Collision happens when two keys may map to the same index

Solution 1: Hashing with Chaining

Each hash table entry contains a pointer to a linked list of keys that map to the same entry. BUT, we may have collisions always and 1 array entry has a linked list of all keys. This leads to n steps to search (n is the size of the list).

Usually, good hash functions can reduce # of collisions. Ideally, we want the length of each linked list to be 1 maximum.

> Good hash functions involve multiplying key with large prime.
>
> e.g. `h(k) = k * 31 % m;`

```
class Node {
  public:
    int key;
    Node * next;
    ~Node() {delete next;}
};

class List {
  private:
    Node *head;
  public:
    List();
    bool isEmpty();
    void insert(int v);
    ListNode *remove(int k);
    bool isFound(int k);
```

```
    ~List();
};
```

```
#define SIZE 7
class HashTable {
  private:
    List **table;
  public:
    HashTable() {
      table = new List*[SIZE];
      for (int i = 0l i < SIZE; i++) table[i] = nullptr;
    }
    bool search(int v) {
      int idx = v %  SIZE;
      if (table[idx]) return table[idx]->isFound(v);
      else return false;
    }
    void insert(int v) {
      if (search(v)) return;
      else {
        int idx = v % SIZE;
        if (!table[idx]) table[idx] new List;
        tble[idx]->insert(v);
      }
    }
    ~HashTable() {
      for (int = 0; i < SIZE; i++) delete table[i];
      delete []table;
    }
    void remove(int v) {
      int idx = v % SIZE;
      if ((table[idx]) && (table[idx]->isFound(v))) {
        delete table[idx]->remove(v);
      }
    }
};
```

Solution 2: Closed Hashing With Linear Probing

If hash functions lead to a collision, then insert values at the next available space in the table.

i.e., if it collides at h(k), then try (h(k) + 1) % size_of_table

```
class Element {
  public:
```

```
    int key;
    bool isDeleted;
};
int hash(int k);
#define SIZE 7
class HashTable{
  private:
    Element **table;
    int size;
  public:
    HashTable() {
      table = new Element *[SIZE];
      for (int i = 0; i < SIZE; i++) {
        table[i] = nullptr;
      }
      size = 0;
    }

    bool insert(int v) {
      int idx = hash(v);
      if (count == SIZE) return false;
      for (int i = 1; table[idx] != nullptr; i++) {
        if (table[idx]->key == v && !table[idx]->isDeleted()) return true;
        if (table[idx]->isDeleted()) {
          table[idx]->key = v;
          count++;
          table[idx]->isDeleted() = false;
          return true;
        }
        idx = (hash(v) + i) % SIZE;
      }
      table[idx] = new Element;
      table[idx]->key = v;
      table[idx]->isDeleted = false;
      count++;
      return true;
    }
};
```

Disadvantage: It would cause clustering of nodes in one area in the hash table. Solution: Try distributing nodes by probing faraway!

Solution 3: Quadratic Probing `(hash(v) + i*i) % SIZE`

Solution 4: Double Hashing `(hash1(v)+ihash2(v)) % SIZE`

`HashTable` class implementation continued

```
void remove(int v) {
  int idx = hash(v);
  for (int i = 1; (table[idx]) && (i <= SIZE); i++) {
    if (table[idx]->key == v && !table[idx]->isDeleted) {
      table[idx]->isDeleted() = true;
      count--;
    }
    idx = (hash(v) + i) % SIZE
  }
  // exit loop when table[idx] is NULL
}

~HashTable() {
  for (int i = 0; i < SIZE; i++) delete table[i];
  delete []table;
}
```

Complexity Analysis

Time Complexity

An **algorithm** is a set of steps to solve a problem.

1. **Linear search**: worst case scenario, the number we search for is the last number. We'll need to make n comparisons.
2. **Binary search**: we assume the array is ordered. We start looking in the middle. If the number we're searching for is smaller, look left. Otherwise, look right. With every comparison, we can toss out half of the remaining array. Worst case is when we have only one element remaining. $x = \log_2(n)$ comparisons

The time it takes to run a program depends on a computer's power, nature of the data, compiler, language, and input size. The choice of the algorithm also matters, however. We will use **time complexity analysis** to evaluate how fast an algorithm is with respect to it's input size.

$T(n)$ is the run-time estimate as a function of input size n.

- We don't care about constants or coefficients — a fast computer can make up the difference between $3n + 4$ and $2n + 2$.
- What we care about is the highest order term, as it dominates when n is very large.
  - We use big-O notation to represent $T(n)$.
- $T(n) = O(g(n))$ if $T(n)$'s highest order term is $g(n)$ — disregarding coefficients and constants.
  - $O(g(n))$ is the upper-bound on $T(n)$.