Contents

1 Algorithms and Data Structures

Definition. *Algorithms*
An *algorithm* is a set of instructions/steps that solves a particular problem.

Each algorithm operates on input data, and produces an output result. Algorithms can be classified by how long they take to run on particular input, and by the quality of their results.

Definition. *Data Structures*
A *data structure* is the set of variable(s) that an algorithm uses to solve a problem.

To solve a problem, you have *o* design both the algorithms and the data structures together. You then need to provide a set of simple "interface" functions to let any programmer use them easily. We call this an ADT, or abstract data type.

1.1 Abstract Data Types

Definition. *ADT*
An *ADT* or *abstract data type* is a coordinated group of data structures, algorithms, and interface functions that is used to solve a particular problem.

In an ADT, the data structures and algorithms are kept secret, while the interface is public to enable the rest of the program to use the ADT. Typically, a program is built from a collection of ADTs, each of which solves a different sub-problem.

1.1.1 ADTs in C++

In C++, we use classes to define ADTs in our programs. Each C++ class can hold algorithms, data, and interface functions.

- Once the class has been defined, the rest of our program can treat the functions as a "black box", i.e. not knowing how exactly how the functions work.
- All our program needs to do is call the functions in our class' public interface.
- All of the underlying data structures and algorithms are hidden from the user.
- The rest of the program can ignore the details of how things work and just use its features.

Note. The benefit of this is that you can change the implementation of the class as much as you want, and as long as the interface functions do what they're supposed to, nothing will break outside of the class. This reduces complexity by breaking down large, complex problems into smaller, self-contained chunks.

1.2 Defining a Class in C++

We first write the outer shell of the class and give it a name, then give the class public and private variables/functions.

```
// Nerd.h
class Nerd
{
public:
  Nerd(int stink, int IQ) {
    myStinkiness = stink;
    myIQ = IQ;
  }
  void study (int hours) {
    myStinkiness += 3*hours;
    myIQ *= 1.01;
  }
  int getStinkyLevel() {
    int total_stink = myIQ * 10 + myStinkiness;
    return total_stink;
  }
private:
  int myStinkiness, myIQ;
};
```

> **Note.** Don't forget the semicolon at the end of the class!

1.3 Using a New Class

Once we define a new class, we can use it to define variables like any traditional data type.

- The class defines a new data type like int, float, or string.
- You typically define each class in its own `.h` file ("header file") and put the file in the same folder as your `.cpp` files. For example, in a file titled `ucla.cpp`, we could have

```
// ucla.cpp
#include "Nerd.h"
int main()
{
  int num_nerds = 1;
  Nerd david(30, 150);

  david.study(10);
}
```

- A header file is similar to a `.cpp` file except you typically only put class declarations and constants in it (you typically put the actual function bodies in your `.cpp` file).
- To import the contents of a header file, just put `#include "filename.h"` at the beginning of the file.
- Once an instance of the class is defined, you can call its member functions, i.e. `study`.

> **Note.** A class' primitive types start out with random values and not zero, so remember to initialise them with the constructor.

- You typically only use member variables to store permanent attributes of a class, i.e. `myStinkiness` and `myIQ`.

- All functions in the `public` section of your class can be seen/called by all parts of your program.
- All functions and data defined in the `private` section of your class are hidden from the rest of your program.

> **Definition.** *Encapsulation*
> Hiding the internal implementation details of a class (from the rest of your program) is called *encapsulation*.

2 Constructors and Destructors

Main Idea. A constructor is used to reset an object's member variables when the object is first created; otherwise they'd be random. An object will often reserve memory slots from the operating system while it runs, so we need a destructor function to free that memory when we delete the object.

2.1 Constructor Basics

We create a constructor just like any other function, but it doesn't have a return type (not even `void`).

```
// Gassy
class Gassy
{
public:
  // The constructor
  Gassy(int age, bool ateBeans)
  {
    m_age = age;
    m_ateBeans = ateBeans;
  }

  int getFartsPerHr()
  {
    if (m_ateBeans == true)
      return(100);
    return(3 * m_age);
  }
private:
  // Member variables
  int m_age;
  bool m_ateBeans;
};
```

```
int main()
{
  Gassy betty(18, true); // Requires two arguments!

  Gassy alan; // error!
}
```

A class can have many different constructors, as long as they have different parameters and/or types.

> **Definition.** *Constructor Overloading*
> When we have multiple constructors in a class, we call this *constructor overloading*.

Note. If you have a constructor that doesn't take any parameters, you may omit the parentheses when creating an instance of the class.

2.2 When Constructors are Called

- When you create a new variable of a class.
- A constructor is called N times when you create an array of size N (that holds the object type).
- A constructor is called when you use `new` to dynamically allocate a new variable.
- If a variable is declared in a loop, it is newly constructed during *every* iteration.
- A constructor is *not called* when you just define a pointer variable.

2.3 Destructors

Just as every class has a constructor, every class also has a destructor function (it may have *exactly one* destructor). It does not have any parameters. To define a constructor function, place a tilde n front of the name of the class, i.e.

```
class Gassy
{
public:
  ~Gassy()
  {
    // Destructor code goes here
  }
private:
  ...
}
```

2.3.1 Why Do We Need Destructors?

If we did not have a destructor, everytime we create a new instance of the class we would take up more and more memory, eventually running out of memory on our system. We need to use destructors to free up memory taken up by deleted objects.

Definition. *Memory leak*
A *memory leak* occurs when you delete an object without first clearing all of the reserved space for that object, effectively "leaking" memory.

2.4 When Must You Have a Destructor?

Any time a class allocates a system resource...	Your class must have a destructor that...
Reserves memory using the `new` command	Frees the allocated memory with the `delete` command
Opens a disk file	Closes the disk file
Connects to another computer over the network	Disconnects from the other computer

3 Class Composition

Main Idea. Classes inside classes. Need I say more?

3.1 Construction Order

- C++ always constructs member variables first, in the order that they're defined in the class.
- Then, C++ constructs the outer class after the member variables have been initialised/constructed.

> **Note.** This makes sense. Since the outer class constructor might need the member variables for construction, they must be constructed before the outer variable.

3.2 Destruction Order

- C++ always runs the main/outer object's destructor first. Its member variables are still valid when the outer destructor runs, so the outer destructor can run properly.
- After the outer class's destructor runs, then C++ runs the destructors of the member variables in the *reverse order* they were constructed.

> **Note (Auto-generated Constructors/Destructors).** If we leave out a constructor and/or destructor, C++ auto-generates an empty constructor and/or destructor for your class. Be careful, because the default constructor will not initialise your primitive member variables. If your class has any primitives you should define your own constructor and initialise them.

3.3 Initialiser Lists

Any time you have a member variable that requires *one or more* parameters for construction, you *must* add an initaliser list to *all* of your outer class's constructor. The syntax is as follows:

```
class HungryNerd
{
public:
  HungryNerd() :
    myBelly(10), myBrain(150), myAge(19) // Runs the corresponding constructors
                                         // with the given values
  {
    myBelly.eat();
  }
private:
  Belly myBelly;
  Brain myBrain;
  int myAge;
};
```

4 Miscellaneous Topics

4.1 Include Etiquette

(a) *Never* include a `.cpp` file in another `.cpp` or `.h` file.

(b) *Never* put a `using namespace` command in a header file. This is called "namespace pollution" and is bad because it forces `.cpp` files that include the header file to use its namespace. Instead, put all of the `using` commands into your C++ files.

(c) *Never* assume that a header file will include some other header file for you. If whoever manages the header file decides to change its implementation, then that will break your code.

4.2 Preprocessor Directives (Include Guards)

Problem. If our header file that we are including already includes another dependency that we are already including in our `.cpp` file, then we will have included that dependency twice! This causes issues like defining methods multiple times, etc.

(a) First, we use `#define` to define new constants, i.e. `#define FOOBAR_H`

(b) We then use the commands `#ifdef` and/or `#ifndef` to check if a variable has been defined (or not). To close the if statement, we use the `#endif` command.

Putting both of the above steps, we have something like this:

```
#ifndef FOOBAR_H // Only run the following if FOOBAR_H isn't defined
#define FOOBAR_H // Define FOOBAR_H
// Notice that this only runs if it has not been run before
class FooBar
{
public:
  some stuff...
private:
  some other stuff...
};
#endif // End the if statement
```

We call this an "include guard", because it "guards" the class and prevents it from being included multiple times.

4.3 When to Include .h Files

You must include the header file (containing the full definition of the class) when:

- You define a regular variable of that class' type.
- You use the variable in any way (call a method on it, return it, etc).

You must do this because C++ needs to know the class's details in order to actually do anything with instances of that class.

You must give a class declaration (just `class ClassName;`) when:

- You use the class to define a parameter to a function.
- You use the class as the return type for a function.
- You use the class to define a pointer or reference variable.

Since you aren't actually using any of the methods of the class nor creating any instances of the class, you don't need to include the full class definition.

Note. The reason why we don't always include the full class definition is because the header file might be large (i.e. thousands of lines) or two classes refer to each other (causing a loop).

4.4 Default Arguments

You can specify a default number of arguments for a function by defining it in the function header, i.e.

```
void doSomething(int varOne = 0, string varTwo = "some value")
{
  do something with the parameters...
}
```

You can leave out the default parameter if you want and C++ will automatically pass in the default value, i.e. `doSomething(5);` or `doSomething();`. However, you can only pass the first k elements as defaults, and may not skip defaults, i.e. `doSomething(,"some string")` is invalid.

One final quirk about default arguments is that if the jth parameter has a default value, then all of the subsequent parameters ($j+1$ to n) *must* also have default values.