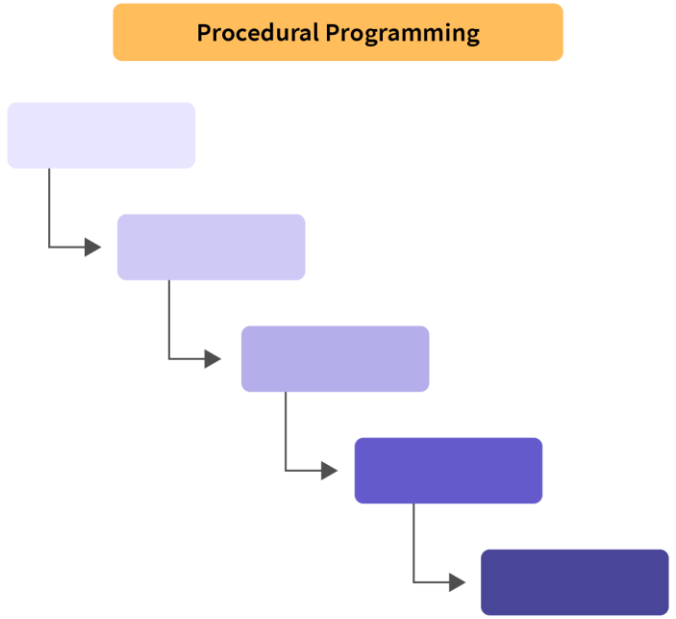
Programming Languages and Tools: Programming with C++ CS:3210:0003

Lecture/Lab #11

Procedural Programming



- Variables, constants, etc. are used to organize data
- Functions/procedures transform data
- Data is passed around between functions

Procedural vs Object-Oriented Programming

- To program the idea of a person eating an apple
  - Data: person, apple
  - Function: eat

- Procedural Programming:
```
eat(you, apple);
```

- OOP:
```
you.eat(apple);
```

Object-Oriented Programming

- Code is organized into classes which encapsulates both
  1. Properties through data
  2. Behaviors through functions
- Pillars of OOP:
  1. Abstraction: hide away implementation details
  2. Encapsulation: combine related data and functions together, restricting data access
  3. Inheritance: reuse data and properties
  4. Polymorphism: share behavior between types

C++ Types

- Fundamental Types:
  - `int`, `bool`, `double`, `float`, …
- Compound Types:
  - C-style arrays and strings, pointers, references, …
- User-Defined Types
  - enums, structs, classes, …
  - Custom names, conventionally starting with a capital letter

Enumerations

- User-defined type whose values are restricted to a set of named symbolic constants (called enumerators)
- Syntax:

```
enum EnumName {
   enumerator1,
   …
   enumeratorn
};
```

- Syntax to define variables of type EnumName:

```
EnumName x { enumerator1 };
EnumName y = { enumeratori };
```

Enumerations

- Enumerator scope:
  - In scoped enumerations, restricted to within the enumeration type
  - In unscoped enumerations, same scope as the enumeration itself
- By default, enumerators start at 0 and each next enumerator has a value 1 greater than the preceding one
- But we can supply explicit initializers to any or all of the enumerators
- Objects/constants of an unscoped enumeration type are implicitly converted to an integral type
- == and != will check for (un)equality of enumerators

structs

- Encapsulate related data in a single structure
- Benefits:
  - Conceptually useful to group data
  - Easier to create and pass all the data around
- Definition syntax:

```
struct StructName {
    type memberName1;
    …
    type memberNamen;
};
```

- Initialization syntax:

```
StructName s {}; //Variable s of type StructName
```

structs

- Dot (.) is the member selection operator
- To access `memberName1` of variable `s` of type `StructName`: `s.memberName1`
- Use aggregate list within { … } for member-wise initialization
- Set default values for members within `struct` definition

Pass by const reference

- Pass by value passes a copy of the value as an arg
- Pass by reference passes a reference
  - Any change made by function is reflected on the referend
- Cannot bind a reference to a constant (variable or literal)
- But we can bind a const reference to either:
  1. a non-constant variable
  2. a constant variable
  3. a constant literal
- But a const reference cannot be changed by the function
- So, prefer passing by const reference for functions that don’t need to modify arg

Passing `structs`

- Since structs often occupy more space pass them by reference
  - Pass by const reference when possible

- Define a struct `Fraction` for fractions and function `printFrac` to print fractions

Model Fractions as a `struct`

Activity

Define function `addFracs` that takes two Fractions (passed by const reference) and returns the Fraction representing their sum