CPSC 2620: C++ Classes

Contact:
Robert Benkoczi
C556, benkoczi@cs.uleth.ca

Outline

Goals:

- understand the concept of ADT. (abstract data types)

Objectives:

- students will correctly write C++ classes implementing ADT.

Resources:

- Chapter 7, Skansholm's text.
- Examples from `c9.io/roben777` } provides a development environment where you can write code until you get your lab account set-up.

Type in C++

Definition → tells compiler what to store under a variable

- identifies the "set of values" for a variable
- set of operations on these values.

Examples: char, int, float

Type in C++

Definition → tells compiler what to store under a variable

- identifies the "set of values" for a variable
- set of operations on these values.

Examples: char, int, float

What does the type tell us about the variables?

`int i;`

- values: $-2^{31} \ldots 2^{31}-1$
- operations: +, -, *, /, %

/ → quotient

`float f;`

- $-3.4 \cdot 10^{38} \ldots 3.4 \cdot 10^{38}$
- +, -, / (fractional division)

`string s;`

- sequence of characters
- concatenation (+), append characters substring, . . .

Exercise

Using CPSC 1620 knowledge, define a new type called rational that supports the following code: Project "rational1"

```
int main() {
  rational val = // initialize to 1/3
  // print val to cout
  return 0;
}
```

```
struct rational {
   int n, d;
};
```

Operations: plus, times, ... (+, * ...)

~~print~~ insert.

The type "rational"

- Values: $\{\frac{d}{n} : d, n \in \mathbb{Z}\}$. (the data structure, the values)
- Operations: insert into output stream (print), extract from input stream (read), add, multiply, etc.

Add the operations to the struct! (see project "rational2")

```
struct rational {
   int n,d;
   void insert (~~rational~~ v, std::ostream& str);
};
```

↑
not needed because
insert is part of
"rational"; we can work with n & d directly

The code for function (method, member function) insert:

scope access operator

```
void rational :: insert ( ostream & str ) {
        str << n << "/" << d ;
}
```

these struct fields belong to the struct variable from the function call.

In main function

a class

the object (an instance of the class)

```
int main () {
    rational val;        // initialize to 1/5 ...
    val.insert(std::cout);
}
```

notice the same struct access syntax, but with functions.

Thursday, Sept 15

-> we will finalize the rational ADT introduced Tuesday:

- students will declare & define constructors to initialize the ADT they are designing.

-> we will split the source code across multiple files

-> we will use access control keywords to hide the implementation details of the ADT from the users of the ADT.

initializing objects

- using constructors

Def: Constructor = special method that:

1) The name is the class name (eg: rational)
2) the return type from the prototype (header) is missing

Ex:

```
struct rational {
    int n,d;
    rational();          ← default constructor.
    void invert(rational &);
};
```

Constructors → may have any kind & number of arguments
↘ can be overloaded. (See project "rational 2")

Calling constructors : when objects are instantiated (variables declared)

Suppose the following constructors are defined

```
struct rational {
    int n,d;
    rational();
    rational(int num, int denum);
    .... // other methods ...
};
```

```
rational val;          ← default constructor
rational v1(1,2);      ← constructor 2, function call syntax
rational v2 = {1, 2};  ← constructor 2, struct (list) initialization syntax
rational v3{1, 2};     ←  — | | — , hybrid syntax
```

The type "rational"

Lecture from Sept. 15 ended here. You can attempt the homework from this and following page.

- Values: $\{\frac{d}{n} : d, n \in \mathbb{Z}\}$.
- Operations: insert into output stream (print), extract from input stream (read), add, multiply, etc.

Homework :

Add the operations to the struct! (see project "rational2")

Points to remember

- (Abstract) Data Type = data structure + operations.
- Add function prototypes of the operations (the methods) to the C++ *struct*.
- The methods can access the *struct* data fields implicitly.
- Constructor methods initialize variables of the type.

Homework

The homework is defined in `https://ide.c9.io/roben777/cpsc2620` in the *homework* folder.

- Project "complex1": define a data type for complex numbers in set $\mathbb{C}$ (see the description given in the source file).
- Project "set1": define a data type for a set of integers (description in the source file).

Object Oriented Programming

The struct + operations defining a type T = class.
A variable of the type T = object.

Object Oriented Programming: attempts to increase productivity...

Example

Use project “rational3” to split the code in separate .cc and .h files.

Example (c'ed)

...Then add access control keywords to hide data members and expose only the operations of the type.

Points to remember

- Write the type (ADT/class) definitions in separate files: declarations (prototypes) in .h files; implementation (code) in .cc files.
- Use *class* insead of *struct*.
- Use access control keywords (private, public) to hide data member definitions (use private) and publish the type operations (use public). This is called “data encapsulation”.

Homework

Revisit homework *complex1* and *set1* and introduce classes in separate files and access control keywords with the data members and methods of your classes.

Conclusion

You should be able now to correctly write simple C++ classes and reuse them in different projects.

Part II: more about methods

Make a copy of project *rational3* into *rational4*. Add a *plus* method that adds two rationals and returns a third one equal to the sum of the two. The two rational arguments should not be modified.

OOP for productivity: reduce programming errors

Question: how can we guarantee that *plus* does not modify its argments?

OOP for productivity: reduce programming errors

Question: how can we guarantee that *plus* does not modify its argments?

Point to remember

Be conservative: declare const all arguments that your method should not modify.

Homework

Add *plus* and *times* methods to the complex class from project *complex1*, similar to the *plus* from class *rational*.
Add *union* and *intersect* methods to the set class from project *set1*.
Introduce keyword *const* in projects *complex1* and *set1* everywhere it is appropriate.

Conclusion

- OOP: classes implement ADT.
- Class definitions and declarations in separate source files to facilitate code reuse.
- Hide data/implementation details; export methods. Use access control keywords *public* and *private*.
- Be conservative: use *const* keyword wherever appropriate.
- Practice and have fun.