**STAGES OF COMPILATION**

Comprehensive compilation consists of three independent but sequential stages (as shown in the figure below):

1. Preprocessor - interprets all directives creating a single translation unit for the compiler - (inserts the contents of all **#include** header files), (substitutes all**#define** macros)
2. Compiler - compiles each translation unit separately and creates a corresponding binary version
3. Linker - assembles the various binary units along with the system binaries to create one complete executable binary

**Forward Declaration Solution** Inserting the prototype into **main.h** along with a forward declaration of **Transaction** informs the compiler that this identifier in the prototype is a valid type.

|  |
| --- |
| **// main.h**  **#define NO\_TRANSACTIONS 3**  **struct Transaction; // forward declaration**  **void add(double\*, const Transaction\*);** |

This design provides the compiler with just enough information to accept the identifer, without exposing the type details.

**Function Signature**

A function's *signature* identifies an overloaded function uniquely.  Its signature consists of

* the function identifier
* the parameter types (ignoring **const** qualifiers or address of operators as described in references below)
* the order of the parameter types

***type* *identifier* ( *type* *identifier* [, ... , *type* *identifier*] )**

The square brackets enclose optional information.  The return type and the parameter identifiers are not part of a function's signature.

C++ compilers preserve identifier uniqueness by renaming each overloaded function using a combination of its identifier, its parameter types and the order of its parameter types.  We refer to this renaming as name *mangling*.

**MEMORY ISSUES**

Issues regarding dynamic memory allocation and deallocation include:

1. memory leaks
2. insufficient memory

**Memory Leak**

Memory leaks are one of the most important bugs in object-oriented programming.  A memory leak occurs if an application loses the address of dynamically allocated memory before that memory has been deallocated.  This may occur if

* the pointer to dynamic memory goes out of scope before the application deallocates that memory
* the pointer to dynamic memory changes its value before the application deallocates the memory starting at the address stored in that pointer

Memory leaks are difficult to find because they often do not halt execution immediately.  We might only become aware of their existence indirectly through subsequently incorrect results or progressively slower execution.

**Insufficient Memory**

On small platforms where memory is severely limited, a realistic possibility exists that the operating system might not be able to provide the amount of dynamic memory requested.  If the operating system cannot dynamically allocate the requested memory, the application may throw an exception and stop executing.  The topic of exception handling is beyond the scope of these notes.  One method of trapping a failure to allocate memory is described in the chapter entitled The ISO/IEC Standard.

We classify member functions into three mutually exclusive categories:

* *queries* - also called accessor methods - report the state of the object
* *modifiers* - also called mutator methods - change the state of the object
* *special* - also called manager methods - create, assign and destroy an object

**Construction**

The compiler assembles an object in the following order

1. allocates memory for each instance variable in the order listed in the class definition
2. executes the logic, if any, within the constructor's definition

An object's destructor

* is called automatically
* cannot be overloaded
* cannot be called explicitly

**this** holds the address of the region of memory that contains all of the data stored in the instance variables of current object.  **\*this** refers to the current object itself;

**Overloading a Member Operator**

**Signature**

The signature of an overloaded member operator consists of:

* the **operator** keyword
* the operation symbol
* the type of its right operand, if any
* the **const** status of the operation

The compiler binds an expression to the member function with the signature that matches the operator symbol, the operand type and the **const** status.

**Promotion or Narrowing of Arguments**

If the compiler cannot find an exact match to an operation's signature, the compiler will attempt a rather complicated selection process to find an optimal fit, promoting or narrowing the operand value into a related type if necessary.

**Type of the Evaluated Expression**

The return type of the member function declaration identifies the type of the evaluated expression.

**COPY CONSTRUCTOR** The copy constructor contains the logic for copying from a source object to a *newly created* object of the same type.  The compiler calls this constructor when the client code. ***Type*(const *Type*&);**

1. creates an object by initializing it to an existing object
2. copies an object by value in a function call
3. returns an object by value from a function

**Definition**

The definition of a copy constructor contains logic to

1. perform a shallow copy on all non-resource instance variables
2. allocate memory for each new resource
3. copy data from the source resource to the newly created resource

**COPY ASSIGNMENT OPERATOR:** The copy assignment operator contains the logic for copying data from an existing object to an *existing* object.  The compiler calls this member operator whenever for client code expressions of the form**: *identifier* = *identifier*. *identifier*** refers to the name of an object.

**Declaration: *Type*& operator=(const *Type*&);** the left ***Type*** is the return type and the right ***Type*** is the type of the source operand.

Reference vs Pointers:

Reference is alias to an object. Reference must be initialize to point to an object and cannot be point to another object. We can't take address of reference like we can with pointer. There's no "reference arithmetics"

3 stages of compilation: preprocessing: deal with instruction starts with # (compiler will go and check #include, replace the thing in # with the content => the result file will get to go compilation), compiling (the result file will compile every modular files into a binary object file) and linking (objects intermediare files and combine them into one executable file)

//ArrayList.h

#ifndef COLLECTION\_ARRAYLIST\_H

#define COLLECTION\_ARRAYLIST\_H

namespace collection{

class ArrayList {

short\* m\_pItems;

int m\_size; //the number of items in m\_pItems

public:

ArrayList(); //constructor and destructor are mutator

ArrayList(short item);

ArrayList(const ArrayList&); //copy contructor, would call itself if there is no pass by reference

~ArrayList();

void display() const; //query

void add (short item); //not a query

ArrayList7 operator= (const ArrayList&); //copy assignment, almost identical to copy constructor

ArrayList& operator+= (short item);

ArrayList& operator+= (const ArrayList& aList);

ArrayList operator+(short item) const; //no & sign because it is returned a completely new object

//const because the changes doesn't apply to current object, it creates a new object

operator bool() const; //we dont put return type because we know what it will return (true or false)

};

std::ostream& operator <<(std::ostream&, const ArrayList&); //don't use namespace in header, that's why std:;

}

#endif

//ArratList.cpp

#include <iostream>

#include "ArratList.h"

namespace collection {

//we need to initialize every single data members in class regardless of empty state signal

ArrayList::ArrayList() {

this->m\_pItems = nullptr;

this->m\_size = 0;

}

ArrayList::ArrayList(short item) {

//notes that we cannot call the default constructor on existing object

//we need a temporary object to copy

this\* = ArrayList(); // don't understand

//Notes\*\*\* only use constructor when memory for object has not yet created

this->add(item);

}

ArrayList::ArrayList (const ArrayList& other){

this->m\_size = other.m\_size; //copy for non resouce object first

this->m\_pItems = new short[m\_size];

for (int i = 0; i < m\_size; i++) {

this->m\_pItems[i] = other.m-pItems[i];

}

}

ArrayList::~ArrayList() {

delete [] this->m\_pItems;

this->m\_pItems = nullptr; //useless when put in destructor but a good practice when used anywhere during codes

}

void ArrayList::display() const {

//no need for null check because m\_size already do the job

for (int i = 0; i < this->m\_size; i++) {

cout << " " << this->m\_pItems[i];

}

cout << endl;

}

void ArrayList::add (short item) {

//allocate memory to store the new array

short \* tmp = new short [this->m\_size +1];

//copy from the old array into the new one

for (int i = 0; i < this-.m\_size; i++) {

tmp[i] = this-.m\_pItems[i];

}

//deallocate memory for old array

delete [] this->m\_pItems;

this->m\_pItems = nullptr; //not neccessary again with good design

//store the new element into the new array

tmp[this->m\_size] = item;

//copy the address into the attribute

this->m\_pItems = tmp;

this->m\_size++; //should it be the other way around??

}

ArrayList& ArrayList::operator=(const ArrayList& other){

//we need to prevent self-assignment or else we will delete the object before it is being copied

if (this != &other){

delete [] this->m\_pItems; //for copy constructor, we need delete because we don't know when it is called, unlike the copy constructor

this->m\_size = other.m\_size; //copy for non resouce object first

this->m\_pItems = new short[m\_size]; //allocate new memory

for (int i = 0; i < m\_size; i++) {

this->m\_pItems[i] = other.m-pItems[i];

}

}

return \*this;

}

ArrayList& ArrayList::operator+=(short item) {

this->add(item); //resuse code

return \*this; //return reference of the object

}

ArrayList& ArrayList::operator+= (const ArrayList& alist) {

for (int i = 0; i < alist.m\_size; i++) {

this->add(aList.m\_pItems[i]);

}

return \*this;

}

ArrayList::operator bool() const {

/\* if (this->m\_size > 0)

return true;

else

return false;

\*/

return this->m\_size > 0; //alternative way to do

}

ostream& operator <<(std::ostream& out, const ArrayList& anArray){

lst.display();

return out; //return sth so we can express multiple expression on a continuous string, if don't we can only return a simple stream

}

}

//main.cpp

#include <iostream>

#include "ArrayList.h"

int main () {

Arraylist al;

al.add(10);

al.add(11);

al.add(20);

al.display();

al += 25; //same as add function but using operator instead

(al += 25) += 30; //return specify from function so we can do multiple operations at same time

ArrayList al2;

(al2 += 100) += 200;

al2.display();

al += al2;

al.display();

al2 = al + 32;

if (al) {

cout << "There are element in the list!" << endl;

}

else

{

coout << "List empty" << endl;

}

ArrayList al4 = 50; //only works because the al4 is not existed at this very line, if the object already exist, compiler will

//not use constructor but it will go look for an operator, remember that constructor can only called once when object is created

ArrayList al4(50); //same syntax as assignment, distinction haopens deoends on status of the object

al4 = 60; //on this line, compiler will look for an assignment operator, it cannot call the constructor because al4 is existing

//special function such as dedault constructor, destructor and copy constructor and copy assignment operator, these are automatically

//added by the compiler.

//basically it takes number 60 turns it into an arraylist and then use the default copy assignmnet to turn the temp object (60) and copy it into al4

al4.display();

return 0;

}

// TODO: header file guard

#ifndef SICT\_FRACTION\_H

#define SICT\_FRACTION\_H

// TODO: create namespace

namespace sict {

// TODO: define the Fraction class

class Fraction {

int m\_numerator;

int m\_denominator;

int gcd() const; // returns the greatest common divisor of num and denom

int max() const; // returns the maximum of num and denom

int min() const; // returns the minimum of num and denom

void reduce(); // simplifies a Fraction by dividing the

// numerator and denominator to their greatest common divisor

public:

// TODO: declare the member functions

Fraction(); //default constructor

Fraction(int nn, int nd); //2 parameters constructor

bool isEmpty() const; //to check if the object is empty

void display() const; //query

// TODO: declare the + operator overload

Fraction operator+ (const Fraction&) const; //operator to add current object and another object, return a new object

Fraction operator\* (const Fraction&) const; //operator to multiply current object ane one from parameter, return a new object

bool operator== (const Fraction&) const; //return true if two fractions are equal

bool operator!= (const Fraction&) const; //retrun true if two fractions are not equal

Fraction& operator+= (const Fraction&); // increment existing fraction by the fraction

};

}

#endif

// TODO: insert header files

#include<iostream>

#include"Fraction.h"

using namespace std;

// TODO: continue the namespace

namespace sict {

// TODO: implement the default constructor

Fraction::Fraction() {

m\_numerator = -1;

m\_denominator = -1;

}

// TODO: implement the two-argument constructor

Fraction::Fraction(int numerator, int denominator) {

//data valid if numerator is non-negative and denomeator is positive

bool valid = numerator >= 0 && denominator > 0;

if (valid == 1) {

//copy data to object when valid

m\_numerator = numerator;

m\_denominator = denominator;

}

if (valid == 0) {

\*this = Fraction();

//set object to safety state if data are invalid

}

}

// TODO: implement the max query

// max returns the maximum of the numerator and denominator

int Fraction::max() const {

int max = m\_numerator;

if (m\_numerator < m\_denominator) {

max = m\_denominator;

}

return max;

}

// TODO: implement the min query

// min returns the minimum of the numerator and denominator

int Fraction::min() const {

int min = m\_numerator;

if (m\_numerator > m\_denominator) {

min = m\_denominator;

}

return min;

}

// gcd returns the greatest common divisor of num and denom

int Fraction::gcd() const {

// min of numerator and denominator

int mn = min();

// max of numerator and denominator

int mx = max();

int g\_c\_d = 1;

bool found = false;

for (int x = mn; !found && x > 0; --x) {

// from mn decrement until divisor found

if (mx % x == 0 && mn % x == 0) {

found = true;

g\_c\_d = x;

}

}

return g\_c\_d;

}

// TODO: implement the reduce modifier

// reduce simplifies the fraction by dividing the numerator and denominator by the greatest common divisor

void Fraction::reduce() {

int g\_c\_d = gcd();

m\_numerator /= g\_c\_d;

m\_denominator /= g\_c\_d;

}

// TODO: implement the display query

void Fraction::display() const {

Fraction temp = \*this;

temp.reduce();

bool empty = isEmpty();

//print out when object is not empty

if (empty == false) {

if (temp.m\_denominator != 1) {

cout << temp.m\_numerator << "/" << temp.m\_denominator;

}

if (temp.m\_denominator == 1) {

cout << temp.m\_numerator;

}

}

//print out error message if object is empty

if (empty == true) {

cout << "no fraction stored";

}

}

// TODO: implement the isEmpty query

// isEmpty returns true if Fraction object is in safe empty state

bool Fraction::isEmpty() const {

bool emptyCheck = false;

//check for safety state

if (m\_numerator == -1 && m\_denominator == -1) {

emptyCheck = true;

}

return emptyCheck;

}

// TODO: implement the operator+

Fraction Fraction::operator+ (const Fraction& rhs) const {

bool empty = isEmpty();

Fraction temp;

if (empty == false) {

temp.m\_numerator = (m\_numerator \* rhs.m\_denominator) + (rhs.m\_numerator \* m\_denominator);

temp.m\_denominator = m\_denominator \* rhs.m\_denominator;

}

return temp;

}

// TODO: implement the operator\*

Fraction Fraction::operator\* (const Fraction& rsh) const {

bool empty = isEmpty();

Fraction temp;

//if object is not empty, perform calculation

if (empty == false) {

temp.m\_numerator = m\_numerator \* rsh.m\_numerator;

temp.m\_denominator = m\_denominator \* rsh.m\_denominator;

}

return temp;

}

// TODO: implement the operator== to compare two object

bool Fraction::operator== (const Fraction& rsh) const {

bool isEqual = false;

bool empty = isEmpty();

if (empty == false) {

if (m\_numerator == rsh.m\_numerator && m\_denominator == rsh.m\_denominator) {

isEqual = true;

}

}

return isEqual;

}

// TODO: implement the operator!= to compare two object

bool Fraction::operator!= (const Fraction& rsh) const {

bool isNotEqual = false;

bool empty = isEmpty();

if (empty == false) {

if (m\_numerator != rsh.m\_numerator && m\_denominator != rsh.m\_denominator) {

isNotEqual = true;

}

}

return isNotEqual;

}

// TODO: implement the operator+=

Fraction& Fraction::operator+= (const Fraction& rsh) {

bool empty = isEmpty();

Fraction temp = \*this;

if (empty == false) {

temp = operator+(rsh);

//add temp object with the Fraction object passed by parameter

temp.reduce(); //reduce object temp

}

\*this = temp;

return \*this;

}

}

Size 6 size 7 Size 8 Size 9 1985