UFUG 2601
C++ Programming

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Important Announcement

- Midterm Exam
  - October 26th (Sun.) 15:00 – 17:00
  - Lecture Hall A
  - Coverage: all topics up to and including Struct and Class
- Programming Assignment 1
  - DDL: Nov. 2nd (Sun.), 23:59
  - The detailed document and OJ contest has been released
  - Late Submission Policy: A penalty of 10% for the corresponding question will be deducted for each day submitted past the deadline.
  - Collaboration is strictly prohibited.

Recap

- Class & Object
- Default Member Functions
  - Default Constructor
  - Copy Constructor
  - Assignment Operator
  - Destructor
- Encapsulation

What’s the output

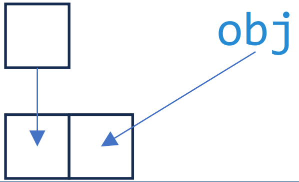
```
class MyClass {
public:
    MyClass() { std::cout << "A"; }
    MyClass(const MyClass& other) { std::cout << "B"; }
    MyClass& operator=(const MyClass& other) { std::cout << "C";
return *this; }
    ~MyClass() { std::cout << "D"; }
};
int main() {
    std::vector<MyClass> vec;
    vec.reserve(10); // preallocate memory for at least 10 elements
    vec.push_back(MyClass());
    MyClass obj;
    vec.push_back(obj);
    vec[0] = obj;
    return 0;
}
```

ABDABCDDD

What’s the output

```
class MyClass {
public:
    MyClass() { std::cout << "A"; }
    MyClass(const MyClass& other) { std::cout << "B"; }
    MyClass& operator=(const MyClass& other) { std::cout << "C";
return *this; }
    ~MyClass() { std::cout << "D"; }
};
int main() {
    std::vector<MyClass> vec;
    // vec.reserve(10); // preallocate memory for at least 10 elements
    vec.push_back(MyClass());
    MyClass obj;
    vec.push_back(obj);
    vec[0] = obj;
    return 0;
}
```

ABDABBDCDDD



Emplace_back

- Avoid creating a temporary object, which is then copied (or moved) to the destination

```
class Point{
public:
    Point(int x = 0, int y = 0) :
x(x), y(y) { cout << "Construct!\n"; }
    Point(const Point& o) : x(o.x),
y(o.y) { cout << "Copy!\n"; }
    Point(Point&& o) : x(o.x), y(o.y)
{ cout << "Move!\n"; }
private:
    int x;
    int y;
};
```

```
int main() {
    vector<Point> vec;
    vec.reserve(10);
    vec.push_back(1,2);              // error
    vec.push_back({1, 2});           // con + move
    vec.push_back(Point{1, 2});// con + move
    vec.push_back(Point(1, 2));// con + move
    vec.emplace_back(1, 2);          // con
    vec.push_back(1);                // con + move
    vec.emplace_back(1);             // con
}
```

Accessing Data Fields

- Only member functions of a class are allowed to access the private data fields of objects of that class

```
void raise_salary(Employee &e, double percent) {
    e.salary = e.salary * (1 + percent / 100);
    // error: 'double Employer::salary' is private within this context
}
```

- Private data fields must be accessed by accessor and mutator functions

```
void raise_salary(Employee &e, double percent) {
    double new_salary = e.get_salary() * (1 + percent / 100);
    e.set_salary(new_salary);
}
```

Accessing Data Fields

- Consider the previous nonmember function `raise_salary`

```
void raise_salary(Emplyee& e, double percent) {
    double new_salary = e.get_salary() * (1 + percent / 100);
    e.set_salary(new_salary);
}
```

It is kind of C-style

- versus the member function

```
void Employee::raise_salary(double percent) {
    salary = salary * (1 + percent / 100);
}
```

Without needing getters/setters
The function logically belongs inside the Employee class

Accessing Data Fields

- A nonmember function is called with two explicit parameters
  - `raise_salary(harry, 7);`
- A member function is called using the dot notation, with one explicit parameter
  - `harry.raise_salary(7);`
- A member function can invoke another member function on the implicit parameter without using the dot notation

```
void Employee::print() const {
    cout << "Name: " << get_name()
    << "Salary: " << get_salary();
}
```

We will see more difference between them after learning inheritance

Accessing Data Fields

- It is common to see a class has methods like `get_xxx` and `set_xxx`
- If we can get and set a member variable, why do we put them in private section rather than public section?

Accessing Data Fields

- Not every data member needs accessor functions (the `Product` class did not have a `get_score()` function)
- Not every get function needs a matching set (the `Time` class can `get_minutes()` but not `set_minutes()`)
- Remember that implementation is supposed to be hidden – just because a class has member functions named get or set does not necessarily explain how the class is designed

```
Time::Time(int hour, int min, int sec) {
    time_in_secs = 60 * 60 * hour + 60 * min + sec;
}
int Time::get_minutes() const {
    return (time_in_secs / 60) % 60;
}
```

Accessing Data Fields

```
class BankAccount {
private:
    double balance;
    bool frozen;
public:
    BankAccount() : balance(0.0), frozen(false) {}
    double get_balance() const {
        if (frozen) {
            std::cout << "Account is frozen. Balance unavailable.";
            return -1;
        }
        return balance;
    }
    void set_balance(double amount) {
        if (frozen) {
            std::cout << "Account is frozen. Balance unavailable.";
        }
        if (amount < 0) {
            std::cout << "Negative balance not allowed"
        }
        balance = amount;
    }
```

Static Member

- A static member belongs to the class, not an individual object

```
class Account {
public:
    Account(const string &name): name(name) {++account_cnt;}
    static int get_count() {return account_cnt;}
private:
    static int account_cnt;
    string name;
};
int Account::account_cnt = 0;
int main() {
    Account a("a"), b("b");
    cout << a.get_count() << b.get_count() << Account::get_count();
}
```

What is the output

```
class Account {
public:
    Account(const string &name):
name(name) {++account_cnt;}
    ~Account() {--account_cnt;}
    static int get_count()
{return account_cnt;}
private:
    static int account_cnt;
    string name;
};
int Account::account_cnt = 0;
```

```
int main() {
    Account a("a");
    cout << Account::get_count();
    {
        Account b("b");
        cout << Account::get_count();
    }
    cout << Account::get_count();
    return 0;
}
```

Static Variable

- Don’t confuse with static member variable
- Static variables are initialized only once and exist until the termination of the program
- It is kind of like global variable, but only visible in the scope
  - “Belongs to” this function

```cpp
void counter() {
  static int count = 0;  // initialized only once
  count++;
  std::cout << count << " ";
}
```

```cpp
int main() {
  counter(); // prints 1
  counter(); // prints 2
  counter(); // prints 3
}
```

Operator Overloading

- We have overloaded `=`

```
Point2D& operator=(const Point2D& p) {
    if (this == &rhs) return *this;
    x = p.x;
    y = p.y;
    return *this;
}
```

- We can overload many operators
  - +,-,*,/,%
  - +=,-=,*=,/=,%=
  - ++,--
  - <,>
  - ….

Operator Overloading

`return-type operator op (arg)`

- To overload unary operator
  - `arg` is empty
  - Exception: ++,--
    - `Point2D& operator++();      // Prefix increment operator`
    - `Point2D operator++(int);    // Postfix increment operator`
- To overload binary operator
  - The caller is the first operand
  - `arg` is the second operand (right-hand side, rhs)
  - e.g., `Point2D operator+(const Point2D& rhs);`

```
auto p3 = p1 + p2;
auto p4 = p1.operator+(p2); // equivalent
```

Operator Overloading

- What if the first operator is not the object of class we define?
  - e.g. we want to do something like `std::cout << p`
- Overloaded as non-member function

```
class Point2D {
public:      Do we have to expose data field to public?
    Point2D(double x, double y):x(x),y(y) {}
    double x;
    double y;
};
ostream& operator<<(ostream& os, const Point2D& p) {
        os << '(' << p.x << ", " << p.y << ')';
        return os;
}
```

Friend Function / Friend Declaration

- To allow a function access private data

```
class Point2D {
public:
    Point2D(double x, double y):x(x),y(y) {}
    friend ostream& operator<<(ostream& os, const Point2D& p);
    // It declares that this non-member function is a friend
private:
    double x;
    double y;
};
ostream& operator<<(ostream& os, const Point2D& p) {
        os << '(' << p.x << ", " << p.y << ')';
        return os;
}
```

operator<<

```
ostream& operator<<(ostream& os, const Point2D& p) {
        os << '(' << p.x << ", " << p.y << ')';
        return os;
}
```

- `std::cout` is an instance of `std::ostream`
- Why we use `ostream&` here?
  - This is for `std::cout << p1 << ", " << p2;`
  - `(((std::cout << p1) << ", " ) << p2);`

Move Semantics & rvalue

- Constructor
- Copy Constructor
- Assignment Operator
- Move Constructor
- Move Assignment Operator
- Destructor

Motivation for Move Semantics

- In some cases, we only need to move an object rather than copy

```
std::vector<int> vec1 = {2, 3, 5, 7};
std::vector<int> vec2 = vec1; // This is a costly copy
for (int v : vec1) std::cout << v << " "; // 2 3 5 7
for (int v : vec2) std::cout << v << " "; // 2 3 5 7
std::vector<int> vec3 = std::move(vec1); // what's this???
for (int v : vec1) std::cout << v << " "; // nothing
for (int v : vec3) std::cout << v << " "; // 2 3 5 7
```

Move semantics allow you to “steal” the data rather than copy each element one by one

Motivation for Move Semantics

```
class MyVector {
    int size;
    int* data;
public:
    Resource(int size) : size(size), data(new int[size]) {}
    ~Resource() { delete[] data; }
    Resource(const MyVector& other) : size(other.size), data(new int[size]) {
        for (int i = 0; i < size; ++i) data[i] = other.data[i];
    }
    Resource(MyVector&& other) : size(other.size), data(other.data) {
        other.data = nullptr;
    }
}
```

- `new int[size]` allocates size*4 bytes in memory and return its address
- `delete[] data` will release the allocated memory (RAII)
- Copy constructor allocates the memory and copy each elements
- Move constructor steal the pointer without allocating memory and copying elements

lvalue & rvalue

- An lvalue is an expression whose address can be taken
  - Is persistent in the memory, permanent
  - You can modify the value
  - e.g., `int x = 3; int y = x;`
- An rvalue is an expression if it results in a temporary object
  - Is about to disappear, temporary
  - e.g., `a + b, Point(3, 5)`
  - Using rvalue reference (&&) can extend its lifetime and keep it alive (so that it can be used later rather than destroyed now)

std::move()

- It's sort of a converter from lvalue to rvalue
- std::move() doesn't actually move anything
    - It just means that “Hey, you can steal anything from me! I don’t need them anymore. Feel free to do anything if that is more efficient”

```cpp
void f(const int& x) {
    std::cout << "const lvalue ref: " << x << std::endl;
}
void f(int&& x) {
    std::cout << "rvalue ref: " << x << std::endl;
}
int a = 2 * 3;
f(a);                   // const lvalue ref: 6
f(2 * 3);               // rvalue ref: 6
f(std::move(a));        // rvalue ref: 6
```

Move Constructor

```
class Product {
public:
    Product(string name, int
id):name(name),id(id){}
    Product(const Product&
other):name(other.name),id(other.id){}
    Product(Product&&
other):name(std::move(other.name)),
id(other.id){}
    void print() {cout << name << ", " <<
id << endl;}
private:
    std::string name;
    int id;
};
```

```
int main() {
    Product a("A", 1);
    Product b = a;
    a.print();  // A, 1
    b.print();  // A, 1
    Product c = std::move(b);
    b.print();  // , 1
    c.print();  // A, 1
}
```

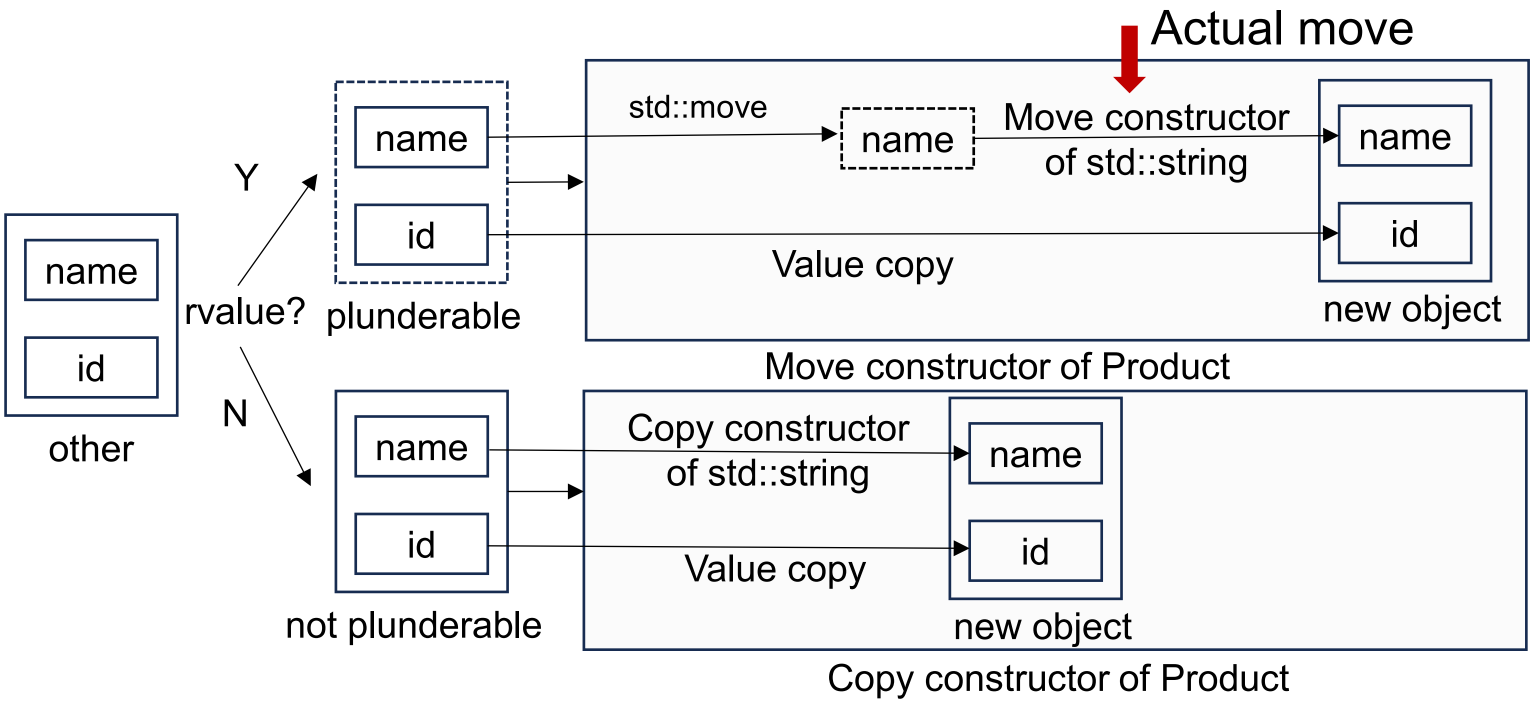
Move Constructor

```
class Product {
public:
    Product(string name, int
id):name(name),id(id){}
    Product(const Product&
other):name(other.name),id(other.id){}
    Product(Product&&
other):name(std::move(other.name)),
id(other.id){}
    void print() {cout << name << ", "
<< id << endl;}
private:
    std::string name;
    int id;
};
```

- std::move() doesn't actually move anything
- Move happens in the implementation of move constructor
- Try to "move" every member by first mark them as plunderable (std::move), and then call their move constructors

Move Constructor

```
Product(Product&& other):
name(std::move(other.name)), id(other.id){}
```



You can also do a simple copy if you find copy is also cheap in move constructor.

Move Constructor

- Pass the resources you hold

```
class MyVector {
    int size;
    int* data;
public:
    Resource(int size) : size(size), data(new int[size]) {}
    ~Resource() { delete[] data; }
    Resource(const MyVector& other) : size(other.size), data(new int[size]) {
        for (int i = 0; i < size; ++i) data[i] = other.data[i];
    }
    Resource(MyVector&& other) : size(other.size), data(other.data) {
        other.data = nullptr;
    }
}
```

- `new int[size]` allocates size*4 bytes in memory and return its address
- `delete[] data` will release the allocated memory (RAII)
- Copy constructor allocates the memory and copy each elements
- Move constructor steal the pointer without allocating memory and copying elements

What’s the output

```cpp
class MyClass {
public:
    MyClass() { std::cout << "A"; }
    MyClass(const MyClass& other) { std::cout << "B"; }
    MyClass& operator=(const MyClass& other) { std::cout << "C"; return *this; }
    MyClass(MyClass&& other) noexcept { std::cout << "E"; }
    MyClass& operator=(MyClass&& other) noexcept { std::cout << "F";return *this;
    ~MyClass() { std::cout << "D"; }
};
int main() {
    std::vector<MyClass> vec;
    vec.reserve(10); // prevent reallocation
    vec.push_back(MyClass());
    MyClass obj;
    vec.push_back(obj);
    vec[0] = obj;
    vec[1] = std::move(obj);
}
```

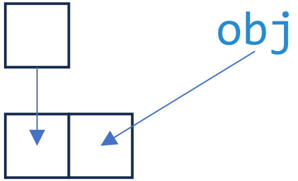
AEDABCFDDD

What's the output

If the move constructor or move assignment operator could throw an exception, `vector` cannot safely use them during reallocation and will fall back to copying elements instead, which may be less efficient.

```
class MyClass {
public:
    MyClass() { std::cout << "A"; }
    MyClass(const MyClass& other) { std::cout << "B"; }
    MyClass& operator=(const MyClass& other) { std::cout << "C"; return *this; }
    MyClass(MyClass&& other) noexcept { std::cout << "E"; }
    MyClass& operator=(MyClass&& other) noexcept { std::cout << "F";return *this;
    ~MyClass() { std::cout << "D"; }
};
int main() {
    std::vector<MyClass> vec;
    // vec.reserve(10); // prevent reallocation
    vec.push_back(MyClass());
    MyClass obj;
    vec.push_back(obj);
    vec[0] = obj;
    vec[1] = std::move(obj);
}
```

AEDABEDCFDDD



What's the output

```
class MyClass {
public:
    MyClass() { std::cout << "A"; }
    MyClass(const MyClass& other) { std::cout << "B"; }
    MyClass& operator=(const MyClass& other) { std::cout << "C"; return *this; }
    MyClass(MyClass&& other) noexcept { std::cout << "E"; }
    MyClass& operator=(MyClass&& other) noexcept { std::cout << "F";return *this;
    ~MyClass() { std::cout << "D"; }
};
int main() {
    std::vector<MyClass> vec(2);
    std::vector<MyClass> vec2 = vec;
    vec2 = vec;
    std::vector<MyClass> vec3(3);
    vec3 = vec;
    std::vector<MyClass> vec4(3);
    vec4 = std::move(vec);
    std::cout << " ";
}
```

AABBCCAAACCDAAADDD DDDDDD

The Compilation of a C++ program

- Preprocessing: deal with headers (#include) and macros (#define), generate “pure” C++ code   g++ -E file.cpp

```cpp
int main() {
    int a = 2, b = 3;
    int c = a + b;
    return 0;
}
```

```cpp
# 0 "test.cpp"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "test.cpp"
int main() {
    int a = 2, b = 3;
    int c = a + b;
    return 0;
}
```

The Compilation of a C++ program

- Preprocessing: deal with headers (#include) and macros (#define), generate “pure” C++ code

```
g++ -E file.cpp
```

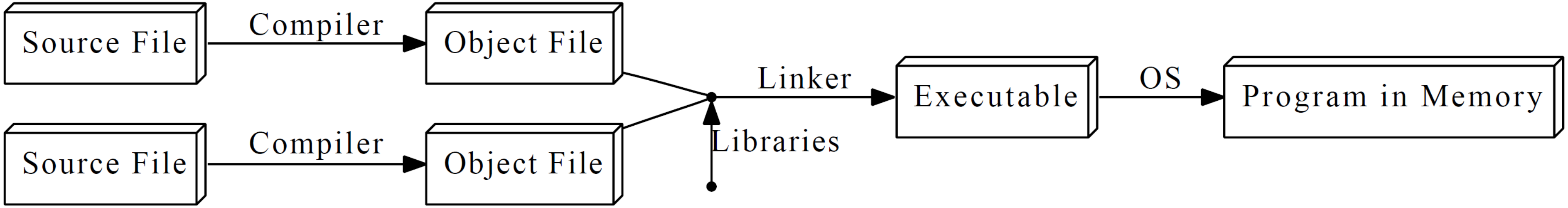
- Compilation: C++ code -> binary object file
  - Compiler: C++-code -> assembly code for a specific processor

```
g++ -S file.cpp
```

```
	.file	"test.cpp"
	.text
	.globl	main
	.type	main, @function
main:
.LFB0:
	.cfi_startproc
	endbr64
	pushq	%rbp
```

The Compilation of a C++ program

- Preprocessing: deal with headers (#include) and macros (#define), generate “pure” C++ code g++ -E file.cpp
- Compilation: C++ code -> binary object file
  - Compiler: C++-code -> assembly code for a specific processor g++ -S file.cpp
  - Assembler: assembly code -> machine code **g++ -c file.cpp**
- Linking: link multiple object files, replaces the references to undefined symbols with the correct addresses, and generate an executable



File Structures

- When your code gets large or you work in a team, you will want to split your code into separate source files
  - Saves time: instead of recompiling the entire program, only recompile files that have been changed.
  - Group work: separate programmers work on separate files
- The header file (e.g. product.h) contains
  - definitions of constants
  - definitions of classes
  - declarations of nonmember functions
  - declarations of global variables
- The source file (e.g. product.cpp) contains
  - definitions of member functions
  - definitions of nonmember functions
  - definitions of global variables

Multi-File Compilation

- Special code (include guards) for the compiler must be put in a header file to prevent the file from being compiled twice

```
#ifndef PRODUCT_H
#define PRODUCT_H
. . .
#endif
```

- Some programmers prefer the non-standard but widely supported `#pragma once`
- The source file includes its own header file `#include "Product.h"`
- The source file does not contain a main function because many programs may use this class.

Multi-File Compilation

```cpp
// Product.h
#include <string>
class Product {
public:
    Product(std::string name, int
id):name(name),id(id){}
    void print() const;
private:
    std::string name;
    int id;
};
```

```cpp
// Product.cpp
#include <iostream>
#include "Product.h"
void Product::print() const {
    std::cout << "Name: " << name <<
std::endl;
    std::cout << "Id: " << id <<
std::endl;
}
```

```cpp
// main.cpp
#include "Product.h"
int main() {
    Product a("A", 12);
    a.print();
}
```

Compilation Using Command Line

- Basically, you need to
  - Compile each source file into binary object file
  - Link multiple files into an executable

```
g++ -c Product.cpp // produce Product.o
g++ -c main.cpp    // produce main.o
g++ main.o Product.o -o main // produce main
```

We don’t want to type these commands by hand everyday

Makefile

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

```
all: main

main: main.o Product.o
    g++ main.o Product.o -o main

main.o: main.cpp
    g++ -c main.cpp

Product.o: Product.cpp
    g++ -c Product.cpp

clean:
    rm *.o
```

```
hkust@Vica-Office:~/cpp/multi$ make
g++ -c main.cpp
g++ -c Product.cpp
g++ main.o Product.o -o main
hkust@Vica-Office:~/cpp/multi$ make clean
rm *.o
rm main
```

CMake

- CMake is a cross-platform build system tool. It generates Makefile or other build system files

```
cmake_minimum_required(VERSION 3.27)
project(Demo)

set(CMAKE_CXX_STANDARD 17)
# Add include directory

include_directories(include)
add_definitions(-O3 -std=c++17)
add_executable(hello src/main.cpp)
```

Multi-file Compilation in VSCode

- Modify the .vscode/task.json to include multiple files

```
"args": [
    "-fdiagnostics-color=always",
    "-g",
    // "${file}",
    "${workspaceFolder}/*.cpp",
    "-o",
    "${fileDirname}\\${fileBasenameNoExtension}.exe
"
],
```

Order of Includes

- It is suggested to follow a specific order of include headers
  - related header (e.g., "product.h")
  - system headers (e.g., <windows.h>)
  - standard libraries (e.g., <string>)
  - others
- Example

```
#include "product.h"

#include <string>
#include <vector>

#include "util.h"
```

Solution

```
void Polynomial::print() const {
    for (int i = coef.size() - 1; i >= 0; --i) {
        if (coef.at(i) == 0) continue;
        if (i != coef.size() - 1 && coef.at(i) > 0) cout << '+';
        if ((coef.at(i) == -1 && i != 0)) cout << '-';
        else if (!(coef.at(i) == 1 && i != 0)) cout << coef.at(i);
        if (i >= 2) {
            cout << "x^";
            cout << i;
        } else if (i == 1) {
            cout << "x";
        }
    }
    cout << std::endl;
```

The logic is quite complicated here!

Do code refactoring!

1
-1
x
x+1
x-1
x^2
x^2+x
x^2+1
x^2+x+1

Nothing is printed for 0

This 0 should be printed if and only if the polynomial is 0

```
if (coef.size()==1 && coef.at(0)==0) {
    std::cout << '0';
} else {
    // blablabla
}
```

Exercise: polynomial_multiplication

https://onlinejudge.hkust-gz.edu.cn/problem/HW1-01

- The polynomial $x - 1$ is represented as degree 1 and the list of coefficients [1, -1].
- The polynomial $2x^2 + 1$ is represented as degree 2 and the list of coefficients [2, 0, 1].
- The polynomial $3x^4 - x + 1$ is represented as degree 4, and the list of coefficients [3, 0, 0, -1, 1].

Sample Input 1

```
2 2 0 1
4 3 0 0 0 1
```

Sample Output 1

```
6x^6+3x^4+2x^2+1
```

Sample Input 2

```
1 1 1
2 1 0 1
```

Sample Output 2

```
x^3+x^2+x+1
```

Sample Input 3

```
2 1 -2 0
3 1 0 1 1
```

Sample Output 3

```
x^5-2x^4+x^3-x^2-2x
```

Exercise

- Implement a class of `Polynomial` (with integer coefficients) so that it is closed under addition/multiplication

```
class Polynomial {
public:
    // what should I provide here?
private:
    // what should I store here?
};
```

Solution

```
class Polynomial {
public:
private:
    vector<int> coef;
};
```

$x^2 + 2x + 5$	[1,2,5]	[5,2,1]
$x + 2$	[1,2]	[2,1]
1	[1]	[1]
0	[0]	[0]

Anything need to be careful?
Are the following ones valid?

- [5,2,1,0] $\rightarrow$ [5,2,1]
- [0,0,1]
- [] $\rightarrow$ [0]

Ensure that the size of `coef` ALWAYS equals to the degree of the polynomial (+1)
And ensure that each polynomial has only one representation
This can be easily maintained using `class`

Anything other issues?
What if we have a $x^{1000000}$?

Sparse representation
$[(0,5),(1,2),(2,1)]$
$[(0,2),(1,1)]$
$[(0,1)]$
$[(0,0)]$

For simplicity we use dense representation here
But it is highly encouraged to implement a sparse one

Solution

```
class Polynomial {
public:
    Polynomial();
    Polynomial(const vector<int>& vec);
    void print() const;
    Polynomial operator+(const Polynomial& rhs) const;
    Polynomial operator-(const Polynomial& rhs) const;
    Polynomial operator*(const Polynomial& rhs) const;
    boolean operator==(const Polynomial& rhs) const;
    static void test();
private:
    vector<int> coef;
};
```

Solution

```
void Polynomial::test() {
    const Polynomial p0 {{0}}; // 0
    const Polynomial p1 {{1}}; // 1
    const Polynomial p2 {{-1}}; // -1
    const Polynomial p3 {{0,1}}; // x
    const Polynomial p4 {{1,1}}; // x+1
    const Polynomial p5 {{-1,1}}; // x-1
    const Polynomial p6 {{0,0,1}}; // x^2
    const Polynomial p7 {{0,1,1}}; // x^2+x
    const Polynomial p8 {{1,0,1}}; // x^2+1
    const Polynomial p9 {{1,1,1}}; // x^2+x+1
```

```
p0.print();
p1.print();
p2.print();
p3.print();
p4.print();
p5.print();
p6.print();
p7.print();
p8.print();
p9.print();
```

```
assert(p0 + p0 == p0);
assert(p0 + p1 == p1);
assert(p1 + p2 == p0);
assert(p1 + p5 == p3);
assert(p9 - p9 == p0);
assert(p9 - p8 == p3);
assert(p9 - p7 == p1);
assert(p0 - p1 == p2);
assert(p9 - p7 == p1);
assert(p0 * p0 == p0);
assert(p0 * p1 == p0);
assert(p0 * p3 == p0);
assert(p0 * p6 == p0);
assert(p1 * p1 == p1);
assert(p2 * p2 == p1);
assert(p3 * p3 == p6);
assert(p3 * p4 == p7);
```

Test-driven development

In fact, it is better to implement the following one for test

```
string Polynomial::to_string() const;
```

Solution

- Implement constructors first

```
Polynomial::Polynomial(): coef({0}) { }
```
What does it mean

```
Polynomial::Polynomial(const vector<int>& coef):
coef(coef) { }
```
Is this correct?

What happen if we initialize a polynomial using `Polynomial p{{1,2,0,0,0}}`

And after some calculation, the size of `coef` will change! $(x^2 + x) - (x^2 + 1)$

We need to ensure the correctness of the states of `coef`

Solution

```
void Polynomial::update() {
    while (coef.back() == 0) coef.pop_back();
}
```

Is this correct?

```
void Polynomial::update() {
    while (!coef.empty() && coef.back() == 0) coef.pop_back();
}
```

Is this correct?

```
void Polynomial::update() {
    while (!coef.empty() && coef.back() == 0) coef.pop_back();
    if (coef.empty()) coef.push_back(0);
}
```

- Should it be a public or private function?
- Users will not (and should not) directly call it, so it is private

Solution

```
void Polynomial::print() const {
    for (int i = coef.size() - 1; i >= 0; --i) {
        cout << coef.at(i);  //print coef
        if (i >= 2) {
            cout << "x^";
            cout << i;
        } else if (i == 1) {
            cout << "x";
        }   //print x
    }
    cout << std::endl;
}
```

0
1
-1
1x0
1x1
1x-1
1x^20x0
1x^21x0
1x^20x1
1x^21x1

Need + for a positive coefficient!

Print many unnecessary ‘1’

Solution

```
void Polynomial::print() const {
    for (int i = coef.size() - 1; i >= 0; --i) {
        if (i != coef.size() - 1 && coef.at(i) > 0) cout << '+';
        if ((coef.at(i) == -1 && i != 0)) cout << '-';
        else if (!(coef.at(i) == 1 && i != 0)) cout << coef.at(i);
        if (i >= 2) {
            cout << "x^";
            cout << i;
        } else if (i == 1) {
            cout << "x";
        }
    }
    cout << std::endl;
}
```

0
1
-1
x0
x+1
x-1
x^20x0
x^2+x0
x^20x+1
x^2+x+1

Need + for a positive coefficient!

Print many unnecessary ‘1’

Print some unnecessary ‘0’s

Can this code help?

```
if (coef.at(i) == 0) continue;
```

Solution

- The coefficient of first term
  - Omit ‘+’ if it is possible
- The coefficient of other terms
  - Should print ‘+’ if it is positive
  - 1 should be omitted if it is 1 or -1
- Constant term
  - 1 cannot be omitted
  - 0 should be omitted if the polynomial is not 0

Solution

```
if (coef.size() == 1) { std::cout << coef.at(0) << std::endl; return;} //constant
for (int i = coef.size() - 1; i >= 1; --i) {
    if (coef.at(i) == 0) continue; // skip zero terms
    if (i != coef.size() - 1 && coef.at(i) > 0) cout << '+';     //print + and -
    if (coef.at(i) < 0) cout << '-';
    if (coef.at(i) != 1 && coef.at(i) != -1) cout << abs(coef.at(i));//print coef
    if (i >= 2) {
        cout << "x^";
        cout << i;        // print x
    } else {
        cout << "x";
    }
}
if (coef.at(0) > 0) std::cout << '+' << coef.at(0);  // print constant
else if (coef.at(0) < 0) std::cout << coef.at(0);
cout << std::endl;
```

There might be more elegant implementations.

Solution

```
Polynomial Polynomial::operator+(const Polynomial& rhs) const {
    Polynomial ret = *this;
    for (int i = 0; i < rhs.coef.size(); ++i)
        ret.coef.at(i) += rhs.coef.at(i);
    return ret;
}
```

- Always be careful when you access a slot of vector!
- What happens if $(x + 1) + (x^2 + x + 1)$
    - `reserve` enough slots for calculation
- What happens if $(x + 1) + (-x + 1)$?
    - Recall that we have implement an `update()`

Solution

```
void Polynomial::reserve(int size) {
    while (coef.size() < size) coef.push_back(0);
}

Polynomial Polynomial::operator+(const Polynomial& rhs) const {
    Polynomial ret = *this;
    ret.reserve(rhs.coef.size());
    for (int i = 0; i < rhs.coef.size(); ++i)
        ret.coef.at(i) += rhs.coef.at(i);
    ret.update();
    return ret;
}
```

It is almost the same for `operator-`

Solution

- For multiplication, the degree will be at most the sum of two degrees

	$(x+5)$	(x^2+x+2)	x^3+6x^2+6x+5
Degree	1	2	3
`coef`	$[5,1]$	$[2,1,1]$	$[10,7,6,1]$
`coef.size()`	2	3	4

The size of resulting polynomial should be `lhs.coef.size + rhs.coef.size - 1`

How to calculate the results?

$$
\begin{array}{llll}
[10, & 2] & & \\
 & [5, & 1] & \\
 & & [5, & 1] \\
[10, & 7, & 6, & 1]
\end{array}
$$

Solution

```
Polynomial Polynomial::operator*(const Polynomial& rhs) const {
    Polynomial ret;
    ret.reserve(coef.size() + rhs.coef.size() - 1);
    for (int i = 0; i < coef.size(); ++i) {
        for (int j = 0; j < rhs.coef.size(); ++j) {
            ret.coef.at(i + j) += coef.at(i) * rhs.coef.at(j);
        }
    }
    ret.update();
    return ret;
}
```

Solution

- Since we have ensured that each polynomial has only one representation, it is easy to check equality

```
bool Polynomial::operator==(const Polynomial& rhs) const {
    if (coef.size() != rhs.coef.size()) return false;
    for (int i = 0; i < coef.size(); ++i) {
        if (coef.at(i) != rhs.coef.at(i)) return false;
    }
    return true;
}
```

Exercise

```cpp
using Term = std::pair<int, int>; // coef, degree
typedef std::pair<int, int> Term; // C-style
```

Type Alias

```cpp
class SparsePolynomial {
public:
    SparsePolynomial();
    SparsePolynomial(const vector<Term>&);
    SparsePolynomial(const Polynomial&);
    Polynomial to_dense() const;
    void print() const;
    SparsePolynomial operator+(const SparsePolynomial& rhs) const;
    SparsePolynomial operator-(const SparsePolynomial& rhs) const;
    SparsePolynomial operator*(const SparsePolynomial& rhs) const;
    bool operator==(const SparsePolynomial& rhs) const;
    static void test();
private:
    vector<Term> terms;
    void add_term(const Term& t);
};
```

Recap for Midterm

- Data Types & Variables
- Const and constant expression
- Control flow
- Function & Recursion
- Vector/Array and Reference/Pointer
- Struct & Class

Data Types

- Number types integer, floating number, and their calculation
- Character type & String type ASCII table, conversion between int and char
- Enum type Conversion between int and char
- Boolean type
- Void type
- Null pointer type

- **Type casting**

Exercise

- What are the values of `c,d,e,f`

```
int main() {
    int a = 2;
    float b = 3.5;
    int c = b - a;          // 1
    int d = a - b;          // -1
    float e = b - a;        // 1.5
    float f = 1 / a + b;    // 3.5
    return 0;
}
```

Exercise

```
int main() {
    int a = 1 << 31;
    cout << a << endl;      // -2147483648
    int b = a - 1;
    cout << b<< endl;       // 2147483647
    unsigned int x = -1;
    cout << x << endl;      // 4294967295
    return 0;
}
```

You don’t need to memorize the exact values :)

Exercise

```
int main() {
    char a = 256 + 'A';
    cout << a;
    return 0;
}
```

- What is the output of this code

A

```
hello.cpp: In function ‘int main()’:
hello.cpp:5:18: warning: overflow in conversion from ‘int’ to ‘char’ changes value from ‘321’ to ‘'A'’ [-Woverflow]
    5 |     char a = 256 + 'A';
      |              ~~~~^~~~~
```

Exercise

- What are the values of these variables

```
enum Weekday {
    MON=1, TUE, WED, THU, FRI, SAT, SUN,
};             2    3    4    5    6    7
int main() {
    Weekday day = TUE;         // 2
    Weekday day2 = 100;        // Error
    int x = TUE + 2;           // 4
    Weekday day3 = TUE + 2;// Error
    return 0;
}
```

Variables

- Name
  - Can only use letters, numbers, and _, cannot starts with a number
  - Avoid using _ as prefix
  - Cannot use reserved words
  - Variable names are case-sensitive
- Type
  - C++ is a strongly-typed language
  - Type need to be specified when declaring a new variable, and cannot be changed

const and constexpr

- `const` applies for variables, and prevents them from being modified in your code (i.e., read-only, run-time constants)
- `constexpr` tells the compiler that this expression results in a compile time constant value (compiler-time constants)

```
int x;
std::cin >> x;
const int y = 2 * x; // I will never change y later
constexpr int z = 2 * x; // Error, cannot be
decided in compiler time
```

Operator

- Arithmetic Operators `+`, `-`, `*`, `/`, `%`
  - `%` only works for integral type
  - Non-integer quotients are rounded towards zero
  - dividend = quotient * divisor + remainder
- Increment / Decrement
  - Prefix `++i` returns the new value, postfix `i++` returns the old value
- Relational Operators
- Logical Operators
- Bitwise Operators

Control Flow

- If-else
- While loop
- For loop
- Switch
  - It evaluates the expression and “jumps into” the case with the same value, and “jumps out” when it hits a break
  - The label should be integral const expression
  - Compare with “goto”?
- Jump
  - break, continue, goto

Function

- Return type, function name, parameter list
  - How to “return” multiple values?
- Default arguments
  - The arguments are passed based on their locations
  - Default arguments must appear at the end of the parameter list

```
hello.cpp:4:34: error: default argument missing for parameter 3 of ‘double foo(int, int, int)’
    4 | double foo(int a, int b = 3, int c) {
      |                              ~~~~^
```

- Function overloading
  - Same function name but different parameter list

Variable Scope

- Their visibility of variables is limited by their scope
- Global variable
  - visible to all blocks
  - preserved throughout the lifetime of the program
- Local variable
  - visible to the current block
  - destroyed after exiting the current block
- Static variable
  - visible to the current block
  - preserved throughout the lifetime of the program

Variable shadowing (Name Hiding)

- The nested variable "hides" the outer variable in areas where they are both in scope

```
int main() {
    int x = 10;
    cout << x;
    {
        int x = 20;
        cout << x;
    }
    cout << x;
    return 0;
}
```

```
namespace first_space {
   void func(){}
}
namespace second_space {
   void func(){}
}
using second_space::func;
int main () {
   first_space::func();
   second_space::func();
   func();
   return 0;
}
```

Vector

```
std::vector<int> v{2, 3, 5, 7}; // list-initialization
std::vector<int> v = {2, 3, 5, 7}; // list-initialization
std::vector<int> firstV(4); // four ints with default value 0
std::vector<int> secondV(4, 10); // four ints with 10
std::vector<int> thirdV(secondV); // copy of secondV
```

A Set of Useful Functions in Vector

- `size()` & capacity()
- `resize()` & reserve()
- `empty()`
- `back()`
- `push_back()`, `pop_back()`, and `emplace_back()`
- `reserve()`

Array

- The capacity is pre-defined can cannot change anymore

```
int scores[10] {1, 3, 4}; // list-initialization
int scores[10] = {1, 3, 4}; // same as the above
int scores[] = {1, 3, 4}; // the length will be 3
```

Pointer v.s. Reference

```cpp
int* ptr = &val;
*ptr = 5;
cout << *ptr << val; // 5 5
val = 8;
cout << *ptr << val; // 8 8
```

```cpp
int& ref = val;
ref = 5;
cout << ref << val; // 5 5
val = 8;
cout << ref << val; // 8 8
```

Pointer and Array

- An array can be viewed as a special pointer, however
  - An array cannot be changed, while a pointer can

```
int arr[10] = {2, 3, 5, 7};
int* ptr1 = arr;
```

```
++arr; // Error!
cout << *ptr1; // 2
++ptr1;
cout << *ptr1; // 3
```

  - The size of an array is the size of type * capacity

```
std::cout << sizeof(arr) << ", " << sizeof(ptr1); // 40, 8
```

40 is because `arr` has 10 `int`, and `sizeof(int)` is 4

8 is because my computer is 64-bit, i.e., 8 bytes

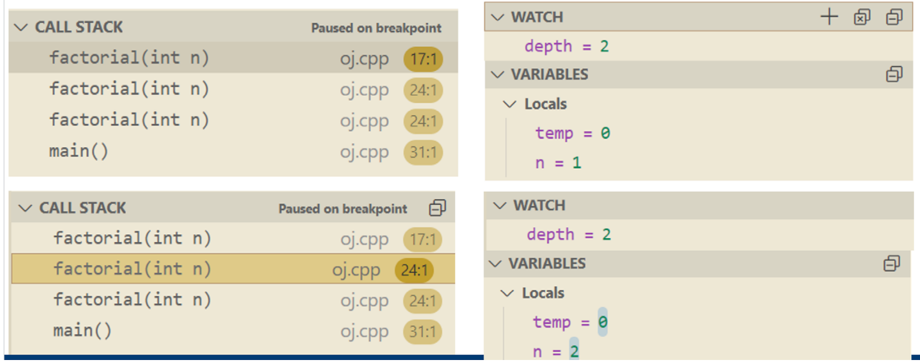
  - An arrays easily decays to pointers to their first element

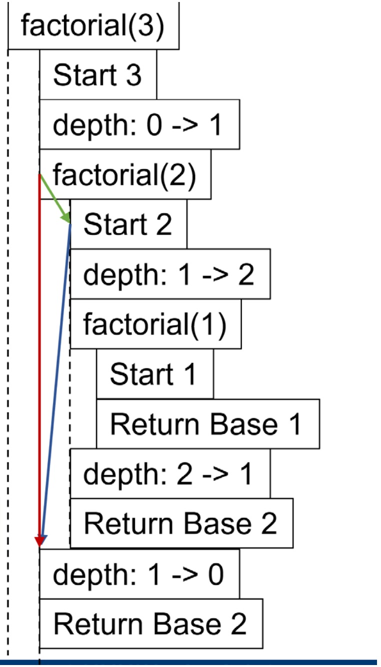
Recursion

- Understand how functions call functions

Use Debugger to Track the Flow

- Step into: go into a deeper level
- Step out: go out of current level
- Step over: go through current level





Recursive Exercise: Permutations

- A permutation is simply a rearrangement of the integers:
  - [1,2,4], [1,4,2], [2,1,4], [2,4,1], [4,1,2], [4,2,1]

```
void permute(vector<int>& v)
```

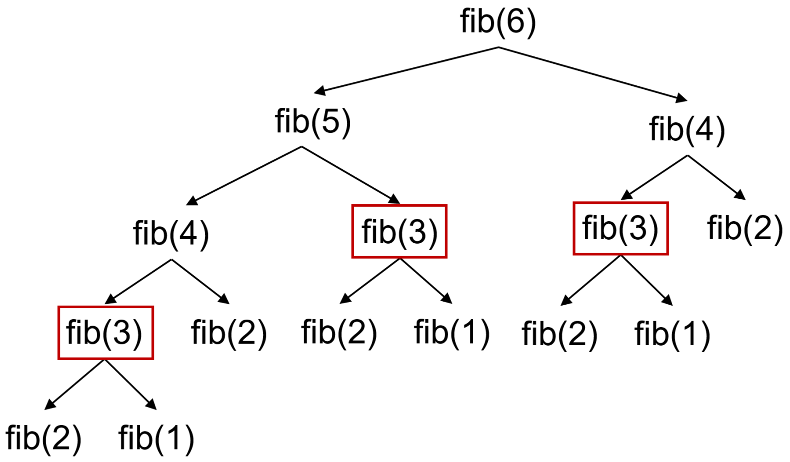
Recursive Example: Permutations

```cpp
void permute(vector<int>& v, int idx) {
    if (idx == v.size()) {
        for (auto val : v) cout << val << " ";
        cout << endl;
    }
    for (int j = idx; j < v.size(); ++j) {
        std::swap(v.at(idx), v.at(j));
        permute(v, idx + 1);
        std::swap(v.at(j), v.at(idx)); // backtrack
    }
}
void permute(vector<int>& v) {
    permute(v, 0);
}
```

What if there are duplicated elements?

The Efficiency of Recursion

- Recursive: solve the problem “top down”
    - Can use *memoization* to accelerate the result
- Iterative: solve the problem “bottom up”
    - It is also called *tabulation* in some context



Thanks