Lecture 2020.10.07

Polymorphism (1)

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

Short Notice

- Will take the midterm
  - 10/21 during the class time
  - Convention hall A, E1 building
- HW2 is released
  - Due: 10/23, 23:59
  - But, do it before the midterm. It will be helpful! ☺
- The guideline for the project proposal is released
  - Due: 10/28, 23:59
    but submit as soon as possible to get started soon
  - You can get early feedback
  - Find teammates using Piazza & email me until 10/19

Today’s Topic

- Class Instance Copy
- Extra for member functions: inline – static – const – friend
- Polymorphism: Overloading
  - Focus on “Operator Overloading”

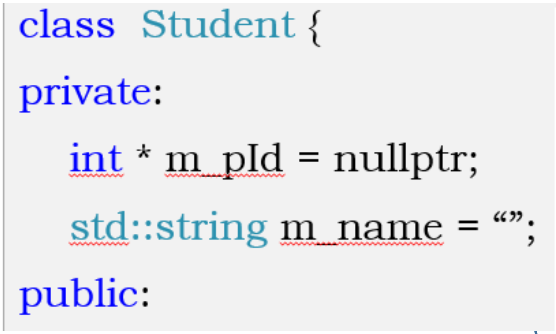
Issue on COPY

```
class  Student {
private:
    int * m_pId = nullptr;
    std::string m_name = “”;
public:
    Student(int, string );
    ~Student();
};
Student::Student (int id, string name) : m_pId{new
int{id}}, m_name{name} { }
Student::~Student () {
    delete m_pId;
}
```

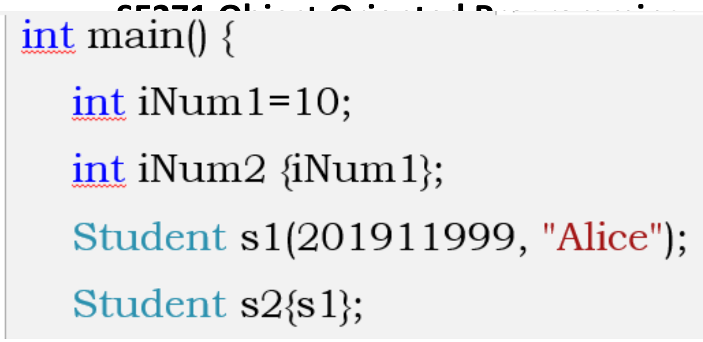
```
int main() {
    int iNum1=10;
    int iNum2 {iNum1};
    Student s1(201911999, "Alice");
    Student s2{s1};
    return 0;
}
```

Issue on COPY(2)

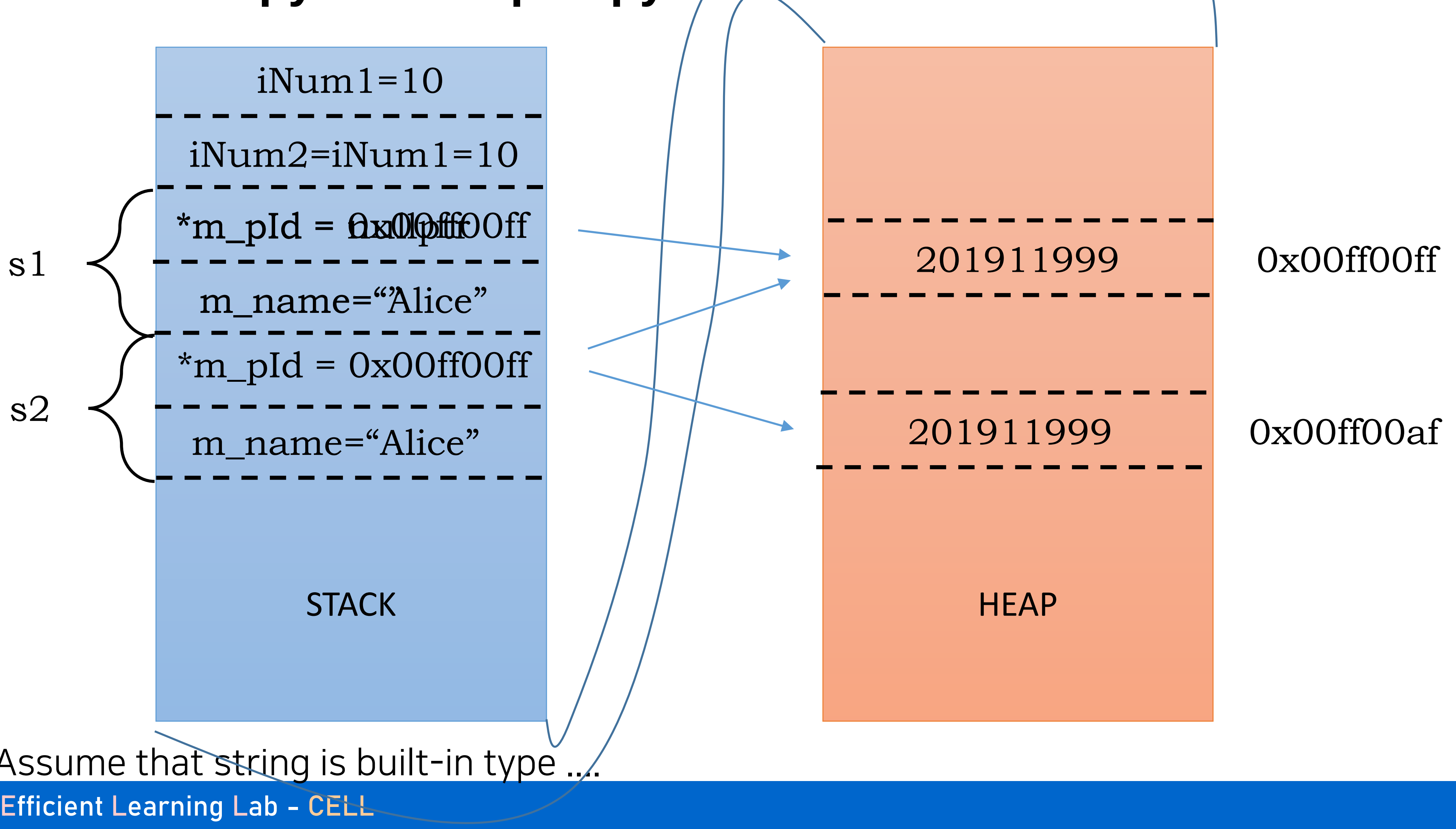
- **Shallow copy VS. Deep copy**



```
class  Student {
private:
    int * m_pId = nullptr;
    std::string m_name = “”;
public:
```



```
int main() {
    int iNum1=10;
    int iNum2 {iNum1};
    Student s1(201911999, "Alice");
    Student s2{s1};
```



iNum1=10

iNum2=iNum1=10

s1

*m_pId = 0x00ff00ff

m_name=“Alice”

s2

*m_pId = 0x00ff00ff

m_name=“Alice”

STACK

201911999

0x00ff00ff

201911999

0x00ff00af

HEAP

* Assume that string is built-in type ….

Copy Constructor

- Initialization with another instance

```
Student s2( s1 );
```

- Default copy constructor

```
Student(const Student& rhs) {
    this->m_pId = rhs.m_pId;
    this->m_name = rhs.m_name;
}
```

- User-specified copy constructor

```
Student(const Student& rhs) {
    this->m_pId = new int(*rhs.m_pId);
    this->m_name = rhs.m_name;
}
```

Extra: inline – static - const

```
class  Student {
private:
static int m_count;
    std::string m_name;
public:
    int GetCount() const { return m_count; }
    std::string GetName() const { return m_name; }

    void GetVariables(int& count, std::string& name)  const  {
        count = GetCount();
        name = GetName();
    }

inline void SetVariables(int cnt, std::string name) {
        m_count = cnt;
        m_name = name;
    }
};
int Student::m_count = 100;
```

Class Member Access - friend

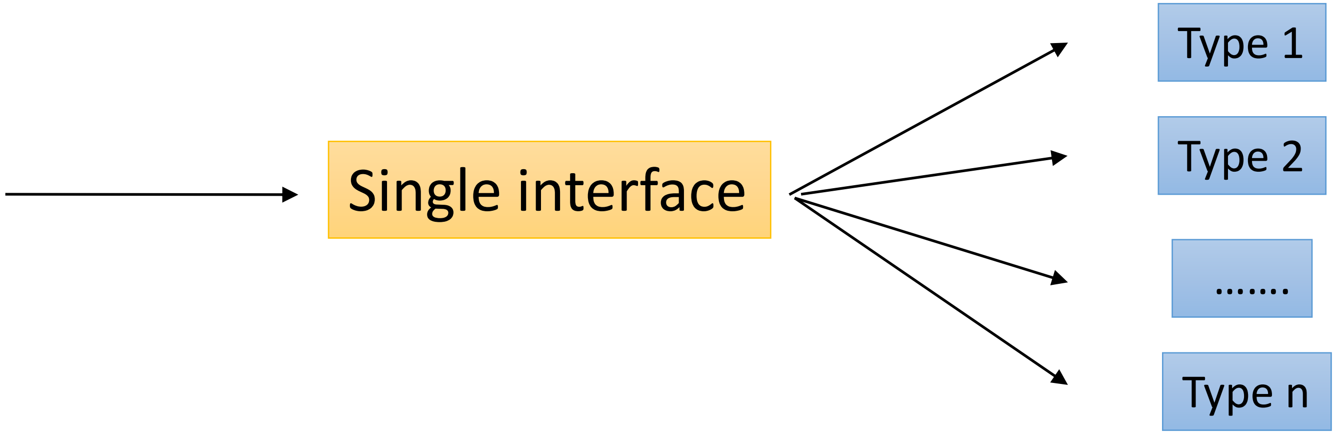
- Private Member
  - – Can be accessed by other class or function by keyword **friend**

```
class Student;
class SE217 {
    public:
    void CallStudent(Student*, int);
};
class  Student {
private:
    int m_Id;
    std::string m_name;
public:
  Student() { }
  Student(int i, string n) : m_Id{ i }, m_name{ n } { }
  friend void SE217::CallStudent(Student*, int); };
```

```
void SE217::CallStudent(Student* st_list, int size)
{
    for (int i = 0; i < size; i++) {
        cout << st_list[i].m_name << endl;
    }
}
int main(){
    Student st_list[30];
     ……
    SE217 cl;
     cl.CallStudent(st_list, 30);
}
```

Polymorphism

- In programming languages and type theory, **polymorphism** is
  - the provision of a single interface to entities of different types[1]
  - the use of a single symbol to represent multiple different types.[2]



1.Bjarne Stroustrup (February 19, 2007). "Bjarne Stroustrup's C++ Glossary". polymorphism – providing a single interface to entities of different types.

2.^ Jump up to: a b c *Cardelli, Luca; Wegner, Peter (December 1985). "On understanding types, data abstraction, and polymorphism" (PDF). ACM Computing Surveys.* ***17*** *(4): 471–523. CiteSeerX 10.1.1.117.695. doi:10.1145/6041.6042. ISSN 0360-0300.*: "Polymorphic types are types whose operations are applicable to values of more than one type."

Example: constructor overloading

- Complex type (23 + 7j)
  - Real part + imaginary part (j)
  - Using int* allocated in heap

```
class Complex {
    int* m_r=nullptr;  // real part
    int* m_i=nullptr;  // imaginary part
public:
    // Constructors
    Complex();
    Complex(int, int);
    Complex(int);
    ~Complex();
    Complex(const Complex & rhs);
};
```

```
Complex::Complex(int r, int i) {
    m_r = new int(r);
    m_i = new int(i);
}
Complex::~Complex() {
    if (!m_r) delete m_r;
    if (!m_i) delete m_i;
    m_r=m_i=nullptr;
}
Complex::Complex() : Complex(0, 0) {}
Complex::Complex(int r)
: m_r{ new int(r) }, m_i{ new int(0) } {}
Complex ::Complex(const Complex & rhs) :
Complex(*rhs.m_r, *rhs.m_i) {}
```

```
void Complex::print()
const {
    cout << *m_r
<< (*m_i < 0 ? "" : "+")
<< *m_i << "j" << endl;
}
```

Operator Overloading

- Operators in C++ are defined as functions – overloading
  - num1 + num2;
    - return_type **operator+** (a, b);
    - return_type a.**operator+**(b);

- Operators (can be overloaded)
  - +, -, *, /, %
  - =, +=, -=, *=, /=, %=
  - ==, !=, >, >=, <, <=
  - ++, -- (postfix, prefix)
  - ^, &, |, ~, <<, >>
  - &&, ||, !
  - ->*, ->, (), [], new, new[], delete[], delete
  - …. (except . , .*, ? : , ::)

Operator Overloading

- Syntax
  - Assume that operator @ is overloaded
  - Binary operators: a @ b
    - a.**operator@(**b)
    - **operator@**(a, b)
  - Postfix unary operators: a@
    - a.**operator@**(int)
    - **operator@**(a, int)
  - Prefix unary operators: @a
    - a.**operator@**()
    - **operator@**(a)

Example: operator overloading +

- Complex type (23 + 7j)
  - **Complex + Complex**

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    Complex operator+(Complex c2);
    Complex operator+(Complex& c2);
    Complex operator+(const Complex& c2);
};
```

```
Complex Complex::operator+(const Complex& c2)
{
    Complex result;
    *(result.m_r) = *(m_r) + *(c2.m_r);
    *(result.m_i) = *(m_i) + *(c2.m_i);
    return result;
}
```

Example: operator overloading +

- Complex type (23 + 7j)
  - Complex + Complex
  - **Complex + int**
  - **Complex + double**

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    // Constructors (omitted)
    Complex operator+(const Complex& c2);
    Complex operator+(int r);
    Complex operator+(double r); };
```

```
Complex Complex::operator+(int r) {
    Complex result;
    *(result.m_r) = *(m_r) + r;
    *(result.m_i) = *(m_i);
    return result;
}
Complex Complex::operator+(double r) {
    return Complex((static_cast<int> (r)) +
*(m_r), *(m_i));
}
```

Example: operator overloading +

- Complex type (23 + 7j)
  - Complex + Complex
  - Complex + int
  - Complex + double
  - **double + Complex**

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    // Constructors (omitted)
    Complex operator+(double r);
    friend Complex operator+(double r, const
Complex& c);
};
```

```
Complex Complex::operator+(double r) {
    return Complex((static_cast<int> (r)) +
*(m_r), *(m_i));
}
```

NOTE: There is no “Complex::”

```
Complex operator+(double r, const Complex& c)
{
    Complex result((static_cast<int> (r)) +
*(c.m_r), *(c.m_i));
    return result;
}


3.1 + C;
```

Example: operator overloading ==, +=

- Complex type (23 + 7j)

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    // Constructors (omitted)
    // + operator overloading (omitted)

    bool operator==(const Complex &);
    void operator+=(const Complex &);
};
```

```
bool Complex::operator==(const Complex &
rhs) {
    if ((*m_r == *rhs.m_r) && (*m_i ==
*rhs.m_i))
        return true;
    else
        return false;
}

void Complex::operator+=(const Complex &
rhs) {
    *m_r += *rhs.m_r;
    *m_i += *rhs.m_i;
}
```

Example: operator overloading ++

- Complex type (23 + 7j)

```
class Complex {
    int* m_r;  // real part
    int* m_i;  // imaginary part
public:
    // Constructors (omitted)
   // + operator overloading (omitted)
   data_type operator++(); // prefix
   data_type operator++(int dummy); // postfix
};
```

```
Complex& Complex::operator++() {
    (*m_r)++;
    return *this;
}

Complex Complex::operator ++(int dummy) {
    Complex ret(*this);
    (*m_r)++;
    return ret;
}
```

First,
copy the value

Return the
copied value

References

- Learn c++
  - https://www.learncpp.com/
  - Chapter 8, 9.1-7, 11

ANY QUESTIONS?