封装

Encapsulation

- Why encapsulation
  - To organize
  - To protect
  - To simplify

    data and code

Private members

- Member variables and functions that can only be accessed by other member functions.

```
1  class Student
2  {
3  private:
4       int id;
5       int old_id;
6       void check_id(int id) {
7            return (id > 10000000 and id < 200000000)
8       }
9  public:
10      ...
11      void change_id(int id) {
12           if (this->check_id(id)) { // call another object function
13                this->old_id = this->id;
14                this->id = id;
15           }
16      }
17      void undo_change() {
18           this->id = this->old_id;
19      }
20 };
```

Private members

- Which ones should be private?
  - Variables and functions that the users do not have to know about
  - Variables and functions that should not be used directly by the users
    - to keep the internal correctness and consistency of the data in the object
  - Functions that are not recommended for directly use
    - e.g. due to being non-intuitive.

Private member

- Private members cannot be accessed from outside the class where they were defined.

```
22 int main() {
23     Student s;
24     // Compilation error: id is a private variable.
25     s.id = 10;
26     // Same error: check_id is a private funciton.
27     s.check_id(10);
28     s.change_id(12345678); // OK
29 }
```

Read/write only variable

```
1  class Student
2  {
3  private:
4      int id; // a read only member variable
5      double gpa; // a write only ...
6
7  public:
8      int getId() { // the getter function for id
9          return id; // this can be omitted
10     }
11     void setGpa(int gpa) { // the setter function for gpa
12         this->gpa = gpa; // here, we cannot omit this
13     }
14 };
```

Function matching in C++

- What is function matching
  - The compiler corresponds function calls to function definitions
- Function matching in C (gcc)
  - Using function names
  - So, every function should have a unique name
- Function matching in C++ (g++)
  - Using function signatures
  - Every function should have a unique signature
    - Name uniqueness is not required

Function matching in C++

- What is function signature?

```
1  // print(int,int)
2  void print(int a, int b) { ... }
3  // print(int)
4  void print(int a) { ... }
5  // print(int,double)
6  void print(int a, double b) { ... }
7  // print(int,int)
8  bool print(int x, int y) { ... }
9  class Student {
10 public:
11      // Student::Student()
12      Student() { ... }
13      // Student::print(int)
14      void print(int a) { ... }
15 }
```

Function matching in C++

- What is function signature?
  –

```
1  // print(int)
2  void print(int & a) { ... }
3  // print(int *)
4  void print(int * a) { ... }
5  // print(int *)
6  void print(int * const a) { ... }
7  // print(int *)
8  void print(int a[]) { ... }
9  // print(const int *)
10 void print(const int a[]) { ... }
```

what is the signature?

```cpp
#include <iostream>
using namespace std;

void fun(int x) {
    cout << 1;
}

void fun(int & x) {
    cout << 2;
}

int main() {
    int x = 1;
    fun(x); // ambiguous
}
```

```cpp
#include <iostream>
using namespace std;

void fun(int x) {
    cout << 1;
}

void fun(int & x) {
    cout << 2;
}

int main() {
    fun(1); // only match
            // the first
}
```

```cpp
#include <iostream>
using namespace std;

void fun(int x) {
    cout << 1;
}

void fun(const int & x) {
    cout << 2;
}

int main() {
    fun(1); // ambiguous
}
```

Constructors

- A constructor is the only way that member variables are initialized
- Can we initialize objects in a class in different ways?
  - Yes, we can provide more than one constructor in a class
  - Every constructors must have a unique signature
    - i.e. every constructor has a unique parameter list

Constructors

- Example

```
1  class Student
2  {
3  private:
4       char name[20];
5  public:
6       Student() { strcpy(this->name, "NO_NAME"); }
7       Student(const char name[]) { strcpy(this->name, name); }
8  };
9
10 int main() {
11      Student s1;
12      Student s1(); // Link error: function s1 is not defined
13      Student s2("Jack");
14      Student s2(12345678); // Error: constructor not defined
15 }
```

Two special constructors

- The default constructor
  - The one without parameters
  - The compiler will generate it if no constructor is provided
- The copy constructor
  - Is used to initialize by copying an existing object
  - Typical example ->
  - The compiler will generate it if it is not provided

```
1  class Student
2  {
3  private:
4       int id;
5       char name[20];
6  public:
7       Student(int id, char name[]) {
8            this->id = id;
9            strcpy(this->name, name);
10      }
11      Student(Student & s) {
12           this->id = s.id;
13           strcpy(this->name, s.name);
14      }
15 };
16
17 int main() {
18      Student s1(12345678, "Jack");
19      Student s2(s1);
20      Student s3 = s1;
21 }
```

- Example

```
4  class Student
5  {
6  public:
7      Student() {} // must provide,
8                   // o.w. no object can be constructed
9      Student(Student & s) {
10         cout << "copy constructor" << endl;
11     }
12 };
13 void test1(Student & s) { cout << "test1" << endl; }
14 void test2(Student s) { cout << "test2" << endl; }
15 Student test3(Student & s) {
16     cout << "test3" << endl;
17     return s; // call copy constructor
18 }
19
20 int main() {
21     Student s1;
22     Student s2(s1); // call copy constructor
23     test1(s1);
24     test2(s1); // call copy constructor
25     test3(s1);
26 }
```

The copy constructor

- Copy constructor is called when
  - Initialize an object using another one

```
1 Student s2(s1);
2 Student s2 = s1;
3 int i2 = 100;
4 int i2(100);
```

  - Initialize an object in the parameter list using the one in the argument list
  - In the return statement: initialize the object returned to the caller using the object returned from the callee

The copy constructor

- The parameter of a copy constructor must be a reference
  - Student(const Student & s)
  - Otherwise, there will be recursive call to itself
    - Student(Student s) ← call itself when trying to initialize s
- Copy constructor is usually unnecessary to write
  - It will be auto-composed if it is not provided
  - The auto-composed version does member-wise copy

```
1  class Student
2  {
3  private:
4      int id;
5      char name[20];
6      double age;
7  public:
8      // You do not need to write this
9      // copy constructor.
10     // It will be auto-composed exactly
11     // as follows.
12     /*
13     Student(const Student & s) {
14         this->id = s.id;
15         memcpy(this->name, s.name, 20);
16         this->age = s.age;
17     }
18     */
19 };
```

The copy constructor

- Situations where we need our copy constructor
  - If the class owns resources, such as dynamic memory, file, CPU resource, network connections, whcih we must define different ways to copy

```
 3 class Student
 4 {
 5 private:
 6     char * name;
 7 public:
 8     Student(const char name[]) {
 9         int len = strlen(name) + 1;
10         this->name = new char[len];
11         strcpy(this->name, name);
12     }
13     ~Student() { delete [] this->name; }
14     Student(Student & s) {
15         int len = strlen(s.name) + 1;
16         this->name = new char[len];
17         strcpy(this->name, s.name);
18     }
19     // The auto-composed copy constructor
20     /*
21     Student(Student & s) {
22         this->name = s.name;
23     }
24     */
25 };
```

Constructors as type conversion rules

- Conversion rules between fundamental data types are defined
- How can be convert between objects of different classes

```
1  class Student {};
2  class Teacher {
3  public:
4      // conversion rules are defined by
5      //   single-argument constructors
6      Teacher(Student & s) {}
7  };
8  void test1(Teacher t) {}
9  void test2(const Teacher & t) {} // const is necesary
10 int main() {
11     Student s;
12     Teacher t1(s); // initialize
13     Teacher t2 = s; // initialize
14     t1 = s; // convert and then assign
15     test1(s); // convert and then match
16     test2(s); // convert and then match
17 }
```