Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Copyright © ViSLAB, The University of Sydney

Object Oriented Design

Week 3-2

Copyright © ViSLAB, The University of Sydney

Encapsulation and Constructor

- Encapsulation : one of three important OO technique
- Constructor : being called when an object is instantiated
- Destructor : being called when an object is destroyed

Copyright © ViSLAB, The University of Sydney

Access Specifier - Encapsulation -

- applied to class's members
- determine their visibility/accessibility

```
class Monster {
    public:
        char name[64];     // the name of the monster
        int power;         // the current power
        int status;        // the current status such as live/dead/paused

        void DisplayData();
};

int main(void) {
    class Monster aMon;

    strcpy(aMon.name, "Pikachu");     //  <- OK.
    aMon.power = 100;                 //  <- OK.
    aMon.status = 0;                  //  <- OK.

    aMon.DisplayData();               //  <- OK.

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

Access Specifiers

- public: open to everything
- private: only within the class
- protected: bit more complicated...

- You can assign different access specifiers to each member.

Copyright © ViSLAB, The University of Sydney

private: Access Specifiers

- cannot access from the outside of an object of the class

```
class Monster {
    private:
        char name[64];      // the name of the monster
        int power;          // the current power
        int status;         // the current status such as live/dead/paused

    public:
        void DisplayData();
};

int main(void) {
    class Monster aMon;

    strcpy(aMon.name, "Pikachu");    //  <- compile error
    aMon.power = 100;                //  <- compile error
    aMon.status = 0;                 //  <- compile error

    aMon.DisplayData();              //  <- OK.

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

private: Access Specifiers

- If you omit an access specifier, a typical C++ compiler will use private: as default

```
class Monster {
        char name[64];    // the name of the monster
        int power;        // the current power
        int status;       // the current status such as live/dead/paused

    public:
        void DisplayData();
};

class Monster {
    private:
        char name[64];    // the name of the monster
        int power;        // the current power
        int status;       // the current status such as live/dead/paused

    public:
        void DisplayData();
};
```

Copyright © ViSLAB, The University of Sydney

protected: Access Specifiers

- behaves like public: or private: depending on how a class is used
- there are many possible access specifier combinations
- the combination occurs by inheritance.
  - you can specify how the new subclass inherit things from its superclass
  - the behaviour of access specifiers in the superclass differ

Copyright © ViSLAB, The University of Sydney

protected: Access Specifiers

- behaves like public: or private: depending on how a class is used

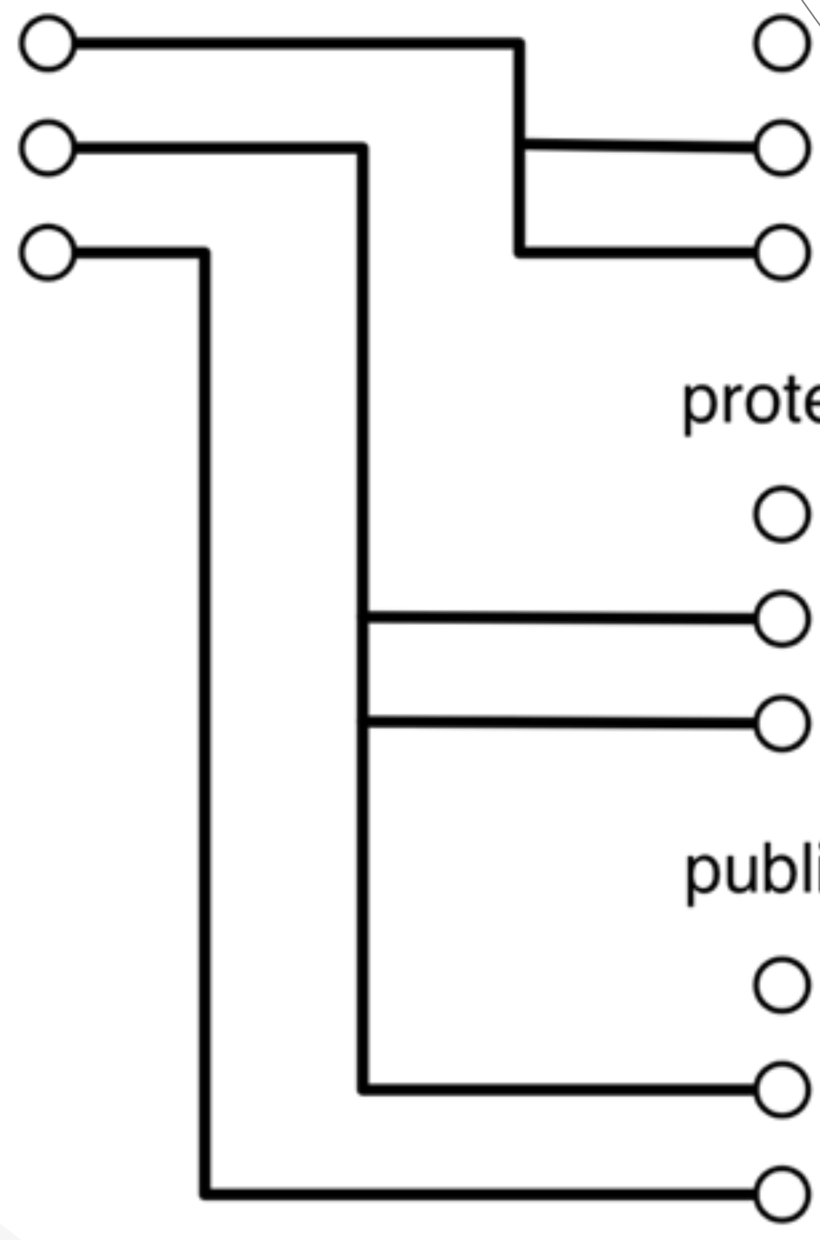
```
// public inheritance
class ASubClass1 : public APublicSuperClass {
        ...
};

// protected inheritance
class ASubClass2 : protected AProtectedBaseClass {
        ...
};

// private inheritance
class ASubClass3 : private APrivateBaseClass {
        ...
};

// the class inheritance without an access specifier is private inheritance as default
class ASubClass4 : APrivateBaseClass {
        ...
};

// the struct inheritance without an access specifier is public inheritance as default
struct ASubClass5 : APublicBaseClass {
        ...
};
```

Copyright © ViSLAB, The University of Sydney

combination of a.s.

SubClass

private
protected
public



private SuperClass

private
protected
public

protected SuperClass

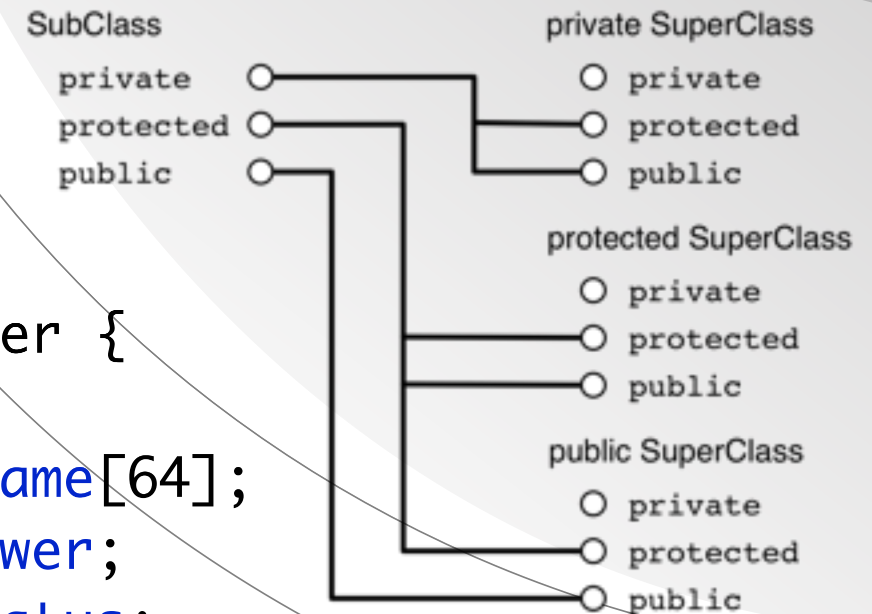
private
protected
public

public SuperClass

private
protected
public

Copyright © ViSLAB, The University of Sydney

combination of a.s.




```
class My Monster : private Monster {
    public:
        void DoSomething() {
            power *= 2; // double the power
        }
};

int main(void) {
    class Monster aMon;
    class MyMonster mMon;

    aMon.power = 100;
    mMon.DoSomething();

    return 0;
}
```

```
class Monster {
  protected:
      char name[64];
      int power;
      int status;

  public:
      void DisplayData();
};
```

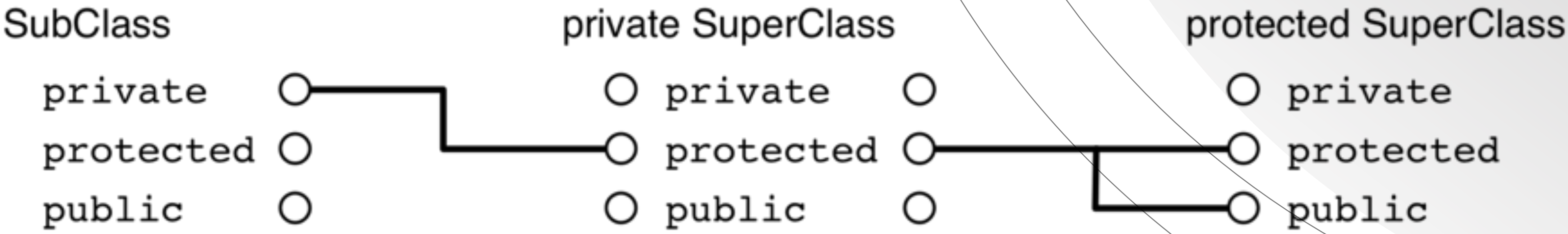
Copyright © ViSLAB, The University of Sydney

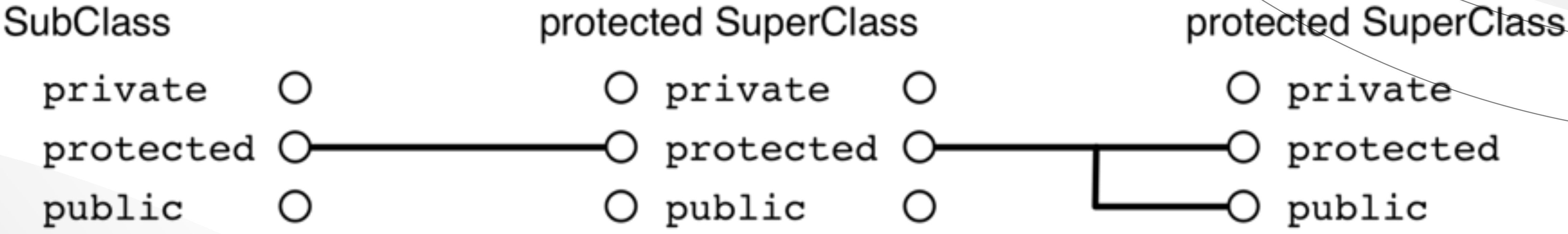
Access Specifier in UML

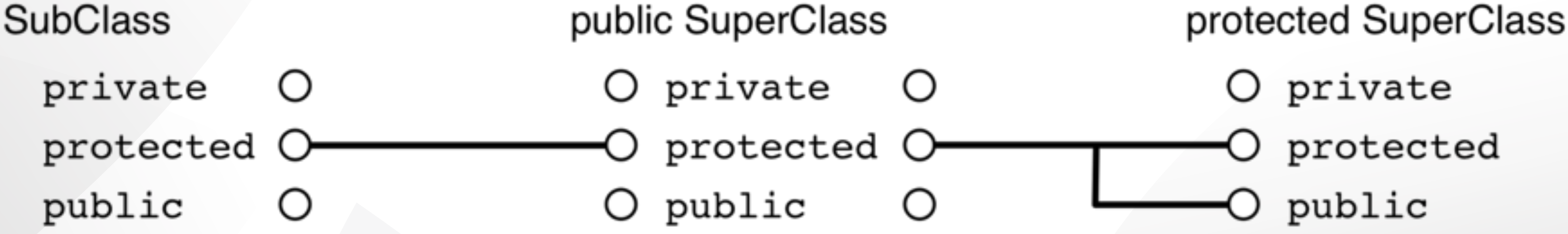
- use
  - - for private
  - + for public
  - # for protected

Copyright © ViSLAB, The University of Sydney

combination of a.s.







Copyright © ViSLAB, The University of Sydney

FYI: combination of a.s. in Java

Specifier	class	subclass	package	world
private	Y			
protected	Y	Y	Y	
public	Y	Y	Y	Y
(none)	Y		Y	

Copyright © ViSLAB, The University of Sydney

Purpose of Encapsulation

- Access specifiers
  - important mechanism for Encapsulation
  - make certain members inaccessible from others

- What will happen if Encapsulation is not available?

Copyright © ViSLAB, The University of Sydney

Purpose of Encapsulation

- Example 1:
  - assume you designed a class as an independent soft. component
  - many objects are instantiated and used in a much larger system.
  - it has a few hundred member variables/functions.
  - only a handful functions need to be called from outside

Copyright © ViSLAB, The University of Sydney

Purpose of Encapsulation

- Example 2:
  - your class has some member variables,
  - you might need to invoke a few internal function when the value of those variables are changed.

  - make those members "private", but
  - provide "public" accessor member functions

Copyright © ViSLAB, The University of Sydney

Purpose of Encapsulation

- Protecting an Object
- Treating an Object as a blackbox.

Copyright © ViSLAB, The University of Sydney

Constructor
Destructor

Copyright © ViSLAB, The University of Sydney

Constructor

- Special member function defined in a class
  - automatically invoked at instantiation (when an object is placed in the memory space.
- Its name is the same as its class name.
- Does not have return value (not even void).

- Compiler can find it and generate a code to invoke constructors.

Copyright © ViSLAB, The University of Sydney

Constructor

- How would you use it?
  - put object's initialisation code.
- How about access specifiers?....try it out.

Copyright © ViSLAB, The University of Sydney

Constructor

- is a member function
  - implementation can be inline or separate definition.

Copyright © ViSLAB, The University of Sydney

```
#include <iostream.h>

class Monster {
        char name[64];      // the name of the monster
        int power;          // the current power
        int status;         // the current status such as live/dead/paused

    public:
        Monster();
        void DisplayData();
};

void Monster::DisplayData() {
    cout << name << "\n";
    cout << power << "\n";
    cout << status << "\n";

    return;
}

Monster::Monster() {
    strcpy(name, "Unnamed");
    power = 100;
    status = 0;

    return;
}

int main(void) {
    Monster aMon;

    aMon.DisplayData();

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

- If the initialization is done in the constructor
  - don't have to explicitly initialize the newly created object.
- "Constructor" is a mechanism to improve the efficiency of OO programming.

Copyright © ViSLAB, The University of Sydney

Constructor with arguments

- Although it's special, it's a member function.
  - it can have arguments.
  - arguments are usually initial values of member variables.

Copyright © ViSLAB, The University of Sydney

```
    #include <iostream.h>

class Monster {
        char name[64];    // the name of the monster
        int power;        // the current power
        int status;       // the current status such as live/dead/paused

    public:
        Monster(char* name, int power, int status);
        void DisplayData();
};

void Monster::DisplayData() {
    cout << name << "\n";
    cout << power << "\n";
    cout << status << "\n";

    return;
}

Monster::Monster(char* name, int power, int status) {
    strcpy(this->name, name);
    this->power = power;
    this->status = status;
}

int main(void) {
    Monster aMon("Unnamed", 100, 0);

    aMon.DisplayData();

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

Overloading Constructor

- Since the constructor(s) is a method
  - you can overload. (same name but different implementation)

Copyright © ViSLAB, The University of Sydney

```
#include <iostream.h>

class Monster {
        char name[64];
        int power;
        int status;

    public:
        Monster();
        Monster(char* name, int power, int status);
        void DisplayData();
};

void Monster::DisplayData() {
    cout << name << "\n";
    cout << power << "\n";
    cout << status << "\n";

    return;
}

Monster::Monster() {
    strcpy(name, "Unnamed");
    power = 100;
    status = 0;

    return;
}

Monster::Monster(char* name, int power, int status) {
    strcpy(this->name, name);
    this->power = power;
    this->status = status;

    return;
}

int main(void) {
    Monster aMon;
    aMon.DisplayData();

    Monster pikachu("Pikachu", 1000, 0);
    pikachu.DisplayData();

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

- any ways to improve the efficiency of constructors?

Copyright © ViSLAB, The University of Sydney

```
#include <string>
#include <iostream>

using namespace std;

class Monster {
        string name;
        int power;
        int status;

    public:
        Monster(string s);
        void DisplayData();
};

void Monster::DisplayData() {
    cout << name << "\n";
    cout << power << "\n";
    cout << status << "\n";

    return;
}
```

```
Monster::Monster(string s) {
    name = s;
    power = 100;
    status = 0;

    return;
}

int main(void) {
    Monster aMon("Pikachu");
    aMon.DisplayData();

    return 0;
}
```

Copyright © ViSLAB, The University of Sydney

```
Monster::Monster(string s) {
    name = s;
    power = 100;
    status = 0;

    return;
}

int main(void) {
    Monster aMon("Pikachu");
    aMon.DisplayData();

    return 0;
}
```

- at instantiation
  - create “name” variable, then
  - substitute the value “s” to it.
- 2 step process....

Copyright © ViSLAB, The University of Sydney

```
Monster::Monster(string s):name(s) {
    power = 100;
    status = 0;

    return;
}

int main(void) {
    Monster aMon("Pikachu");
    aMon.DisplayData();

    return 0;
}
```

- Compiler generates a code
    - initialize “name” when it is created.

Copyright © ViSLAB, The University of Sydney

Initialization list

```
Monster::Monster(string s, int p, int st):name(s), power(p), status(st) {}

Monster::Monster(string s):name(s), power(100), status(0){}

int main(void) {
    Monster aMon("Pikachu");
    aMon.DisplayData();

    return 0;
}
```

- initialization via substitution
  - should use initialization list

Copyright © ViSLAB, The University of Sydney

Destructor

- created objects need to be destroyed....
- Special member function
  - automatically invoked when an object is destroyed (deallocated).
- starts with “~” (tilda)
- public:
- no argument .....no overloading....(please check it)

Copyright © ViSLAB, The University of Sydney

Destructor

- timing of invoking destructor:
  - when the program control goes out a block, which declares the object
  - when the program control goes out the main().

Copyright © ViSLAB, The University of Sydney

Destructor (M$)

- Microsoft introduced Finalizer
  - !ClassName
  - can be deterministically invoked
  - used to release resources no longer needed
    - call free-like member function.

Copyright © ViSLAB, The University of Sydney