LAB 5 – INTRO TO C++

CONSTRUCTORS

- The place where we setup our class’s instance
- There can be 0 or more parameters
- Always exactly one constructor

DESTRUCTORS

- The place where we tear down our class’s instance
- Always exactly one destructor
- Can be virtual
- This is where we want to free anything that was allocated within the instance

PUBLIC, PRIVATE, PROTECTED, FRIEND

- public:
  - Everybody can see and use it
- protected:
  - Only classes who inherit from the class can access it (including itself)
- private:
  - Only itself can see and use it
- friend:
  - A specific field can be given access to a specific class
  - Field F in class C1 can be accessed from only C2, when given permission.
    - C2 does not have access to anything but F in C1 although given special permissions

PUBLIC, PROTECTED, PRIVATE EX

```
class PrivateProtectedPublic {
private:
    int privInteger; // Only Visible Within
    int privFunc(void); // Only Visible Within

protected:
    double protDouble; // Only Visible Within and Inherited
    double protFunc(void); // Only Visible Within and Inherited

public:
    char pubChar; // Visible by anybody
    char pubFunc(void); // Visible by anybody

};
```

FRIEND EX 1

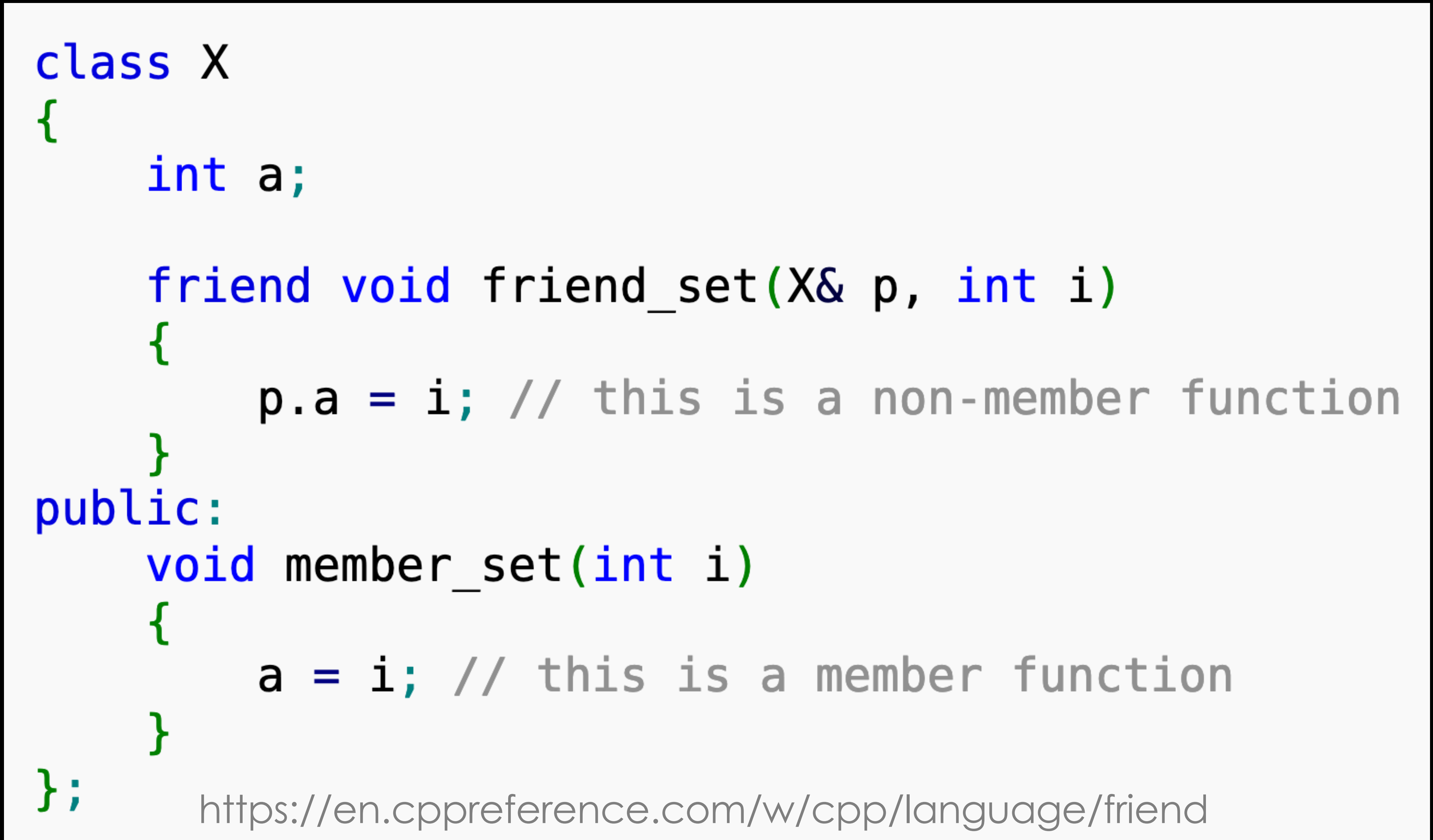
- Note: We do not like friends as they are used incorrectly a lot

```
class Y
{
    int data; // private member

    // the non-member function operator<< will have access to Y's private members
    friend std::ostream& operator<<(std::ostream& out, const Y& o);
    friend char* X::foo(int); // members of other classes can be friends too
    friend X::X(char), X::~X(); // constructors and destructors can be friends
};

// friend declaration does not declare a member function
// this operator<< still needs to be defined, as a non-member
std::ostream& operator<<(std::ostream& out, const Y& y)
{
    return out << y.data; // can access private member Y::data
}
```

https://en.cppreference.com/w/cpp/language/friend

FRIEND EX 2

- Note: We do not like friends as they are used incorrectly a lot



```cpp
class X
{
    int a;

    friend void friend_set(X& p, int i)
    {
        p.a = i; // this is a non-member function
    }
public:
    void member_set(int i)
    {
        a = i; // this is a member function
    }
};
```

https://en.cppreference.com/w/cpp/language/friend

METHOD VS FUNCTION

- A **related member** is one defined **in** a class
- A **non-related member** is one defined **outside** of a class
- A method is a related member
  - It is defined in the class
- A function is a non-related member
  - It is defined elsewhere of the class

OVERLOADED FUNCTIONS

- An overloaded function has multiple different parameter types with the same name

- int add(int, int)
- double add(double, double)
- string myConcat(string, string)
- char * myContact(char *, char *)

OVERLOADED OPERATOR

- Complex operator+ (const Complex &lhs, const Complex &rhs);
- In this example (1.a.iii), we have an operator which accepts a Complex class for the left and right side of the + operator. It then returns a Complex class.
- Suppose we implement an add function (1.a.ii) in Complex. What does our function look like for the prototype given above?
- Important note: The parameters of the operator are constant and passed by reference!

PASS-BY- VALUE VS REFERENCE

- Pass by value
  - Pass in the value directly, as we have in C
  - Same with pointers, we copy the value of the address to the callee

- Pass by reference
  - Pass in the value indirectly. Not as we use pointers
  - When we modify the value in the function, it acts as if we dereference a pointer
    - (By Reference) We need to: x = 100;
    - (With pointers) We need to: *x = 100;
    - Both of the above modify the source variable

CLASSES VS OBJECTS

- An **object** is an **instance** of a **class**

- Object
  - The actual thing in memory
- Instance
  - Relationship between object and class
- Class
  - The outline of the object that will be created

DYNAMIC MEMORY: GOODBYE MALLOC

```
size_t count = 100;


int * myNumber = new int;
int myNumbers[count]; // Note: We can now use non-constant values to set an
array size. You can count the number of elements you need before giving it a
size. Not resizable, so it is not dynamic memory, but is a feature of C++.
int * myNumbersDyn = new int[count];


delete myNumber; // if you use new TYPE
delete[] myNumbersDyn; // if you use new TYPE[COUNT]
```

USING STATEMENTS

```
#include <iostream>
#include <string>
#include <vector>
using std::cout;
using std::endl;
using std::string;
using std::vector;

// You can now use cout instead of std::cout
// You can now use endl instead of std::endl
// You can now use string instead of std::string
```

HPP
(THE PLAN/OUTLINE OF A CLASS)

```
class PrivateProtectedPublic {
private:
    int privInteger;
    int privFunc(void);

protected:
    double protDouble;
    double protFunc(void);

public:
    char pubChar;
    char pubFunc(void);

};
```

CPP
(THE IMPLEMENTATION OF PLANS)

```cpp
int PrivateProtectedPublic::privFunc() {
    return this->privInteger;
}
double PrivateProtectedPublic::protFunc() {
    return this->protDouble;
}
char PrivateProtectedPublic::pubFunc() {
    return this->pubChar;
}
```

RESOURCES

- https://en.cppreference.com/w/cpp/language/friend
- https://learn.microsoft.com/en-us/cpp/cpp/friend-cpp?view=msvc-170
- https://en.cppreference.com/w/cpp/language/class
- https://cplusplus.com/doc/tutorial/classes/
- https://cplusplus.com/doc/tutorial/dynamic/
- https://en.cppreference.com/w/cpp/memory
- https://cplusplus.com/doc/tutorial/classes/
- https://cplusplus.com/doc/tutorial/
- https://en.cppreference.com/w/cpp