ECE 220: Computer Systems & Programming

Lecture 20: Intro to C++: Objects, Constructors

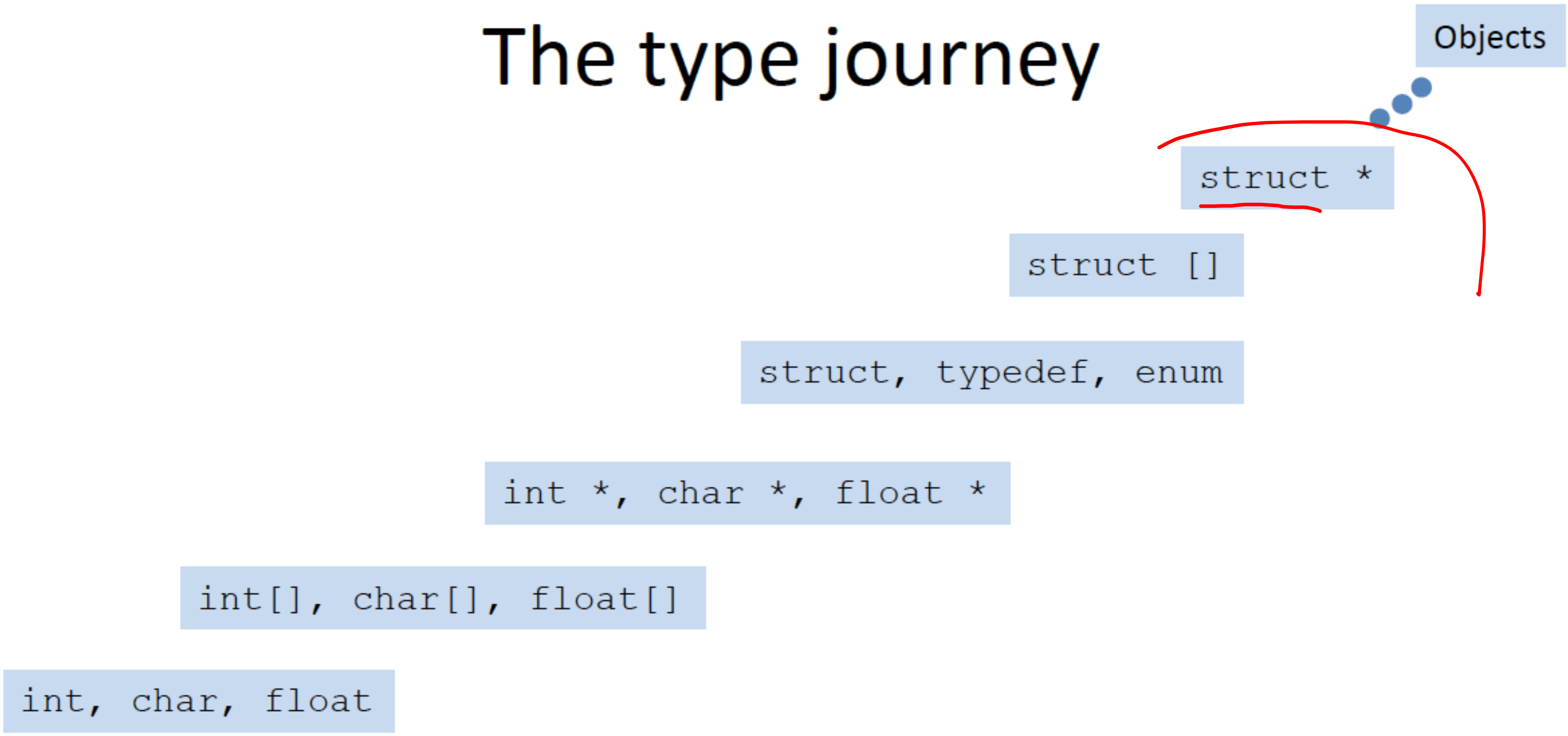
Thomas Moon

April 4, 2024

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

The type journey



Objects

struct *

struct []

struct, typedef, enum

int *, char *, float *

int[], char[], float[]

int, char, float

Motivation: Is Structure good enough?

```
#include <stdio.h>
typedef struct StructLaptop{
    int screenSize;
    int RAM;
    int power;
}laptop;

void powerON(laptop *p){
    p->power = 1;
}

void powerOFF(laptop *p){
    p->power = 0;
}

void printStatus(laptop *p){
    printf("Screen size: %d, RAM: %d,
Power: %d\n", p->screenSize, p->RAM,
p->power);
}
```

```
int main(){
    laptop mylaptop = {12, 8, 0};
    laptop *p = &mylaptop;

    powerON(p);
    printStatus(p);

    powerOFF(p);
    printStatus(p);

    mylaptop.power = 1;
    p->power = 1;
    p->power = 5;
}
```

The functions are not part of ”latptop” struct.

Members of “laptop” are accessed by anyone.

Not so happy about…

1. Struct cannot include functions
2. Member in struct can be accessed by anyone

C++: Class

```
#include <iostream>

class laptop{
private:
    int screenSize;
    int RAM;
    int power;
public:
    laptop(int _screenSize, int _RAM, int _power){
        screenSize = _screenSize;
        RAM = _RAM;
        power = _power;
    }
    void powerON(){ power = 1;}
    void powerOFF(){ power = 0;}
    void printStatus(){
        std::cout<<"Screen size: "<<screenSize<<", RAM: "<<RAM<<", 
Power: "<<power<<std::endl;
    }
};
```

Class can give different access

Class can have functions

C++: Class - continued

```cpp
int main(){
    laptop mylaptop(12, 8, 0);
    laptop *p = &mylaptop;

    mylaptop.powerON();
    mylaptop.printStatus();

    mylaptop.powerOFF();
    mylaptop.printStatus();

    mylaptop.power = 1; //compile error
    p->power = 2; //compile error
    mylaptop.printStatus();
}
```

👏 access to the private members not allowed outside the class

Encapsulation!

C++

- **Object Oriented Programming** (OOP)
  Programming style associated with **class** and **objects** and other concepts like
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

- Class – a blueprint for object (*laptop*).
  Similar to Struct in C except it defines
  - control "who" can access the data
  - provide functions specific for the class

Concepts Related to Class

- Object – an instance of the class (*mylaptop*)
  - shares the same function with other objects of the same class
  - but each object has its own copy of the data

- Member functions (methods) – functions that are part of a class

- Private vs. Public members
  - **private** members can only be accessed by member functions (default)
  - **public** members can be accessed by anyone

Access Modifiers: private and public

```
class AAA{
    private:
        int x;
    public:
        // member functions can access
        // private member x
        int getx(){return x;}
        void setx(int x_){x = x_;}
};
using namespace std;
int main()
{
    AAA a;
    // access private member directly outside the class
    // COMPILE ERROR!
    a.x = 1;
    cout<<a.x<<endl;
    // access private member through the piblic member functions
    a.setx(1);
    cout<<a.getx()<<endl;
```

Before dive in OOP,

Here are some (technical) updates from C to C++.

- File extension from .c to .cpp
- Compiler from gcc to g++
- I/O function
- Namespace
- Dynamic allocation (malloc to *new*, free to *delete*)
- Function overloading
- Operator overloading
- Default Arguments

...

Basic Input & Output

C

```
#include <stdio.h>

printf(”Hello World : %d\n", a);

scanf(”%d", &a);
```

C++

```
#include <iostream>

std::cout<<"Hello World : "<<a<<std::endl;

std::cin >> a;
```

- cin: standard input stream (use with >>)
- cout: standard output stream (use with <<)
- endl: standard end line '\n'

** You can still use the c-style I/O functions by including <cstdio>

Namespace

- A method for preventing name conflict.

```
// code from Alice
void sayHello(){
    std::cout<<"Hello from Alice";
}

// code from Bob
void sayHello(){
    std::cout<<"Hello from Bob";
}

int main()
{
    sayHello();
}
```

Compile Error!

```
namespace A{
// code from Alice
    void sayHello(){
        std::cout<<"Hello from Alice";
    }
}
namespace B{
// code from Bob
    void sayHello(){
        std::cout<<"Hello from Bob";
    }
}
int main(){
    A::sayHello();
    B::sayHello();
}
```

Use :: to resolve scope

Namespace with using

- We can use *using* keyword so that we don't have to use complete name all the time.

```
using namespace A;
int main()
{
    A::sayHello();
    B::sayHello();
    sayHello();
}
```

```
A::sayHello();
```

```
namespace std{
    cout ???
    cin ???
    endl ???
}
```

1

```
using namespace std;
          or
```

2

```
using std::cout;
using std::cin;
using std::endl;
```

```
cout<<"Hello from Alice"<<endl;
```

No more "std::" needed.

Dynamic Memory Allocation

- `new` – operator to allocate memory (similar to malloc in C)
- `delete` – operator to deallocate memory (similar to free in C)

C

```c
int *ptr;
ptr = (int*) malloc(sizeof(int));
free(ptr);
```

C++

```cpp
int *ptr;
ptr = new int;  ←
delete ptr;
```

- To allocate/deallocate an array of memory,

```cpp
int *ptr;
ptr = new int[10];
delete []ptr;
```

Function Overloading

- Two or more functions can have the same name but different parameters (type & number, not return type)

```
int f(void){
    cout<<"int f(void)"<<endl;
}
int f(int a){
    cout<<"int f(int a)"<<endl;
}
int f(int a, int b){
    cout<<"int f(int a, int b)"<<endl;
}
int f(char a, char b){
    cout<<"int f(char a, char b)"<<endl;
}
```

```
int main()
{
    f();
    f(10);
    f(10, 20);
    f('a', 'b');
}
```

```
double f(char a, char b){
    cout<<”double f(char a, char b)"<<endl;
}
```

<- Can we add this function?

Default Arguments

- If the caller function does not provide a value for the arguments, then it is automatically assigned by the compiler with a default value.

```
int volume(int length, int width = 1, int height = 1){
    return length * width * height;
}

int main(){
    cout << volume(4) << endl;
}
```

```
      == volume(4,1,1)
```

Default Arguments & Function Overloading

- Mixing default arguments with function overloading can cause *ambiguity*.

```
int volume(int length, int width = 1, int height = 1){
    return length * width * height;
}
int volume(int length){
    return length;
}


int main(){
    cout << volume(4, 2) << endl;
    cout << volume(4) << endl;
}
```

This is OK.

This causes compile error because it is ambiguous.

Initialize Objects

```
class Person{
private:
    char name[20];
    int age;
public:
    void ShowData();
};

int main(){
    Person p = {"Alice", -20};
}
```

Try to initialize just like structure.

Compile error

because the members (name and age) are *private*!

To solve,

1. Make the members *public* (not recommended)
2. Use "constructor"

Constructor

- A special method which is invoked automatically at the time of object creation.
- Used to initialize the data members.
- It has the same name as class.
- 2 types: default constructor & parameterized constructor
- Overloading and default arguments are possible.
- No return value

default constructor:

compiler implicitly declare <u>if no constructor provided by user</u>.

```
class Person{
    char name[20];
    int age;
public:
    void ShowData();
};
```

→

```
class Person{
    char name[20];
    int age;
public:
    Person(){};
    void ShowData();
};
```

Constructor

```
#include <cstring>

class Person{
    char name[20];
    int age;
public:
    Person(char const *_name, int _age);
    void ShowData();
};
Person::Person(char const *_name, int _age){
    strcpy(name, _name);
    age = _age;
}



int main(){
    Person p1 = {"Alice", 20};
    Person p2("Alice", 20);
    Person p3 = Person("Alice", 20);
}
```

They all call **`Person(char const *_name, int _age);`**

Default Constructor

```
class Person{
    char name[20];
    int age;
public:
    Person(char const *_name, int _age);
    void ShowData();
};
```

```
int main(){
    Person p1("Alice", 20);
    Person p2;
}
```

???

Looking for `Person()` construct.

But it's not declared.

```
class Person{
    char name[20];
    int age;
public:
    Person(char const *_name, int _age);
    Person(){};
    void ShowData();
};
```

← You need to explicitly declare the default constructor.

Destructor

- Destructor is a member function that destructs an object.
- It is called automatically when the object goes out of scope.
- It has the same name as class, but prefixed with ~.
- **No argument** (Overloading and default arguments are NOT possible).
- No return.

```
public:
    Person(){};
    Person(char const *_name, int _age);
    ~Person(){}; // destructor
```

Destructor

```
class Person{
    char *name;
    int age;
public:
    Person(){};
    Person(char const *_name, int _age);
    void ShowData();
    ~Person();
};
Person::Person(char const *_name, int _age){
    name = new char[strlen(_name) + 1];
    strcpy(name, _name);
    age = _age;
}
Person::~Person(){
    delete []name;
}
```

```
int main(){
    Person p1("Alice", 20);

}
```

→ Destructor is useful to deallocate memory

Operator Overloading

Not allowed overloading

```
.
::
?:
sizeof
```

- We can “redefine” the built-in operators (+, -, /, *, =,…).
- Overloaded operators are functions with special names: `operator` followed by the operator symbols.

```
int main(){
    Point p(1,2);



    p + 10;

      ↓

    p.operator+(10);


    p + 10.5;
```

```
class Point{
  private:
    int x,y;
  public:
    Point(int _x=0, int _y=0){x=_x; y=_y;}
    void ShowPosition();
    void operator+(int val){
      x = x + val;
      y = y + val;
    }
    void operator+(double val){
      x = x + val;
      y = y + val;
    }
```

Operator Overloading – A better way

```
int main(){
    Point p1(1,2);

    Point p2 = p1 + 10;
```

p1.operator+(10)

```
class Point{
    private:
        int x,y;
    public:
        Point(int _x=0, int _y=0){x=_x; y=_y;}
        void ShowPosition();
        Point operator+(int val){
            Point temp(x+val, y+val);
            return temp;
        }
```

Operator Overloading

- You can also define the operators between two objects.

```cpp
int main(){
    Point p1(1,2);
    Point p2(3,1);

    Point p3 = p1 + p2;
```

p1.operator+(p2);

```cpp
class Point{
    private:
        int x,y;
    public:
      . . .
        Point operator+(int val){
            Point temp(x+val, y+val);
            return temp;
        }
        Point operator+(Point p){
            Point temp(x+p.x, y+p.y);
            return temp;
        }
};
```

Note: `private` is to “Class”, not “Object”