

# Introduction to C, C++, and Unix/Linux

CS 60

*Today*

- C++ classes
- Reading [KR] Chapters 1-7
- Read [So] chapters 1, 3, 4 (Boolean), 9, 13, 14 & 18 & 10

```
SuperStarPlayer p1("Steve Nash");
StarPlayer p2("Dirk Nowitzki");
Player p3;
p3.Name() = "Shawn Bradley";

p1.AddGame(48, 5, 5);
p2.AddGame(25, 8, 3);
p3.AddGame(0, 0, 0);

p1.Print();
p2.Print();
p3.Print();
p1.IncreaseSalary(90);
p2.ShoeContract() = "Nike";
```

```
class Player {
private:
    std::string name;
    int games;
    int points;
    int rebounds;
    int assists;
public:
    Player();
    Player(std::string name);
    void Reset();
    void Print();
    bool AddGame(int, int,
        int);
```

```
    void SetName(std::string n);
    void SetGames(int g);
    void SetPoints(int p);
    void SetRebounds(int r);
    void SetAssists(int a);
    std::string Name();
    int Games();
    int Points();
    int Rebounds();
    int Assists();
};
```

*Methods to Get and Set values*

```
class Player {
private:
    std::string name;
    int games;
    int points;
    int rebounds;
    int assists;
public:
    Player();
    Player(std::string name);
    void Reset();
    void Print();
    bool AddGame(int, int,
        int);
```

```
    std::string& Name();
    int& Games();
    int& Points();
    int& Rebounds();
    int& Assists();
};

Player p1("Steve Nash");
p1.Games() = 82;
p1.Points() = 1512;
p1.Rebounds() = 281;
p1.Assists() = 1183;
```

*Get and Set values via returned  
reference variables*

```
class Player {
private:
    std::string name;
    int games;
    int points;
    int rebounds;
    int assists;
public:
    Player();
    Player(std::string name);
    void Reset();
    void Print();
    bool AddGame(int, int,
        int);
```

```
    std::string& Name();
    const int& Games();
    const int& Points();
    const int& Rebounds();
    const int& Assists();
};

    Player p1("Steve Nash");
p1.Games() = 82;
p1.Points() = 1512;
p1.Rebounds() = 281;
p1.Assists() = 1183;
```

## Data access

- `p1.games`, `p2.points`, `p3.assists` are all illegal accesses, because the class variables are **private**
- Inside a **Player** function definition, the variables **games**, **points**, and **assists** can be accessed and modified directly
- How about inside a **StarPlayer** function definition? A **SuperStarPlayer** function definition?

## Data access (cont.)

- The **protected** label indicates that the member variable is private to everyone except the functions of the class **and of any derived class**

```
class Player {  
protected:  
    std::string name;  
    int games;  
    int points;  
    int rebounds;  
    int assists;  
    ...  
}
```

**StarPlayer** and  
**SuperStarPlayer** can  
access these variables directly

# Data access summary: data and functions

- **public**

- Can be universally accessed: directly from the class object, in functions of the class and any derived class

- **protected**

- Can be accessed by functions of the class and any derived class

- **private**

- Can only be accessed by functions of the class

```
Player p1;
```

```
p1.games = 0;
```

```
int Player::f()
```

```
int StarPlayer::f()
```

```
int Player::f()
```

```
int StarPlayer::f()
```

```
int Player::f()
```

# The class interface

- Designing a class interface is an art, not a science
  - Should any data elements be public?
  - Should any data elements be modifiable via functions that “go both ways” (get/set)?
  - Should relevant computations (e.g., points per game, rebounds per game) be assigned when the raw data is entered, or when requested?
  - How many different ways should the class be initialized?

# Player class file structure

- **Player.h**
  - Class definition
- **Player.cpp**
  - Class functions defined
- File using the Player class includes the header:
  - **#include <Player.h>**

Except for simple classes,  
which can go in the file with  
the **main** function

## Structures vs. Classes

- A structure is just like a class with all the data fields and member functions declared public
  - It is legal to define a function within a structure, by the way
- Structures are still used in C++
  - For data only, when there are no associated functions that should be defined along with the data

# Class constructors

- Every class has at least one class **constructor**
  - **Player ()** is created automatically if a class constructor is not explicitly defined
  - Several constructors may be defined (function overloading)
- When a class object is created (*an instance of the class is created, the class is instantiated*), the class constructor is automatically called
  - Player p1;
  - Player p2(“Steve Nash”);
  - Player p3(“Steve Nash”, 4, 85, 15, 49);

*How many different constructors?*

```
Player::Player(str::string str="", int g=0,  
              int p=0, int r=0, int a=0)  
{  
    name = str;  
    games = g;  
    points = p;  
    rebounds = r;  
    assists = a;  
}
```

*Note no return type*

Could be accomplished with one constructor with default parameters, as here.

Or as several different constructors that allow different combinations of parameters to be set.

# Class destructor

- Every class has exactly one **destructor**
  - **~Player()** is created automatically if the class destructor is not explicitly defined
  - No function overloading for the destructor
- The destructor is called when the variable is destroyed
  - Goes out of scope
  - **delete** called on a pointer to the variable
- Don't call the destructor directly!

## Class destructor (cont.)

- Destructors are used as an opportunity to free up heap memory (call **delete** where appropriate), to warn of any strangeness, and to “tidy up”
- Example: an image class
  - When the image object is destroyed, should you free the image memory?
  - Not if another image is also pointing to that same memory!

# Copy constructor

- The copy constructor is invoked to make a copy of a class object
- Generally used in three ways:
  - **Player p1 (p2) ;** // p1 is a copy of existing p2
  - **Player p1 = p2 ;** // Same as above
  - **p3 = function(p1) ;** // p1 is copied onto the stack, and the result is copied from the stack into p3 (call by value)

*This assumes that a constructor is defined that takes a Player as the argument*

## Copy constructor (cont.)

```
Player p1 = p2;
```

The copy constructor is used here, NOT the = operator – the compiler understands that this is an initialization. So it's not treated as:

```
Player p1;
```

```
p1 = p2; // Here the assignment operator is used, not  
         the copy constructor!
```

## Copy constructor (cont.)

```
Player::Player(const Player& oldp)
{
    name = oldp.name;
    games = oldp.games;
    points = oldp.points;
    rebounds = oldp.rebounds;
    assists = oldp.assists;
}
```

The default copy constructor (provided by the compiler if one is not specified) does this anyway – copies the member elements  
Why might you need a copy constructor?

# The assignment operator

- It's often very useful to overload the assignment operator =

```
Player p1;
```

```
p1 = p2;      Used here
```

```
Player Player::operator= (const Player& oldp)
```

As with the copy constructor, the default = copies all the member elements, but pointers (heap memory) must be dealt with by overloading and handling it yourself (allocate and copy)

## Explicit constructor

If you really do want

```
Player p1 = p2;
```

to be the two-step initialization (creation + assignment), then you can tell the compiler to make the constructor *explicit*

```
explicit Player(Player& oldp);
```

*See p. 205*

## Highly recommended exercise

- Create a class with various constructors, a copy constructor, an assignment operator, and a destructor
  - In each of these functions, print out a short message (like `std::cout << "In constructor #1\n"`)
- Experiment with different ways of declaring a class object, assigning a value to it, assigning another object to it, deleting it, letting it go out of scope, etc.
  - See what gets printed out – and if you predicted it!

# Defining/overloading class operators

- Can be overloaded:

– unary operators

`+` `-` `*` `&` `~` `!` `++` `--` `->`

`new` `new[]` `delete` `delete[]`

– binary operators

`+` `-` `*` `/` `%` `^` `&` `|` `<<` `>>`

`+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<=` `>>=`

`<` `<=` `>` `>=` `==` `!=` `&&` `||`

`,` `[]` `()`

Try some!

## Example: Overloading ==

- The equivalence operator (==) has to be defined for a new class, if it is to be used
  - As do most of the operators on the previous slide
  - Some already exist (e.g., new, delete) but can be overloaded
- Let's define and overload == for **Player**

```
class Player {  
    ...  
public:  
    Player();  
    Player(std::string name);  
    bool operator== (const Player& p);  
    bool operator== (std::string name);  
    ...  
};
```

```
bool Player::operator==(const Player& p)
{
    if (name == p.name && games == p.games &&
        points == p.points && assists == p.assists &&
        rebounds == p.rebounds) return true;
    else return false;
}
```

```
bool Player::operator==(const str::string& n)
{
    if (name == n) return true;
    else return false;
}
```

*Now we can write this code:*

```
if (p1 == p1) ...
```

```
if (p1 == "Steve Nash") ...
```

```
if (p1 == 3) ... // ERROR
```

**Let's fix that**

```
class Player {  
    ...  
    bool Player::operator== (int n);  
    ...  
}
```

```
bool Player::operator== (int n)  
{  
    if (games == n) return true;  
    else return false;  
}
```

## Exercise

- Create a class called Base with the various constructors, etc., and try

```
Base a;           Base *g = &a;
Base b(4);       Base *h = new Base;
Base c=4;       Base *i = new Base(c);
Base d=c;       Base *j = new Base[3];
Base e(4, 3);   a = b;
Base f(b);     Func(a);
```

What gets printed out when we add print statements (as in the Class List)?

# Automatically generated class functions

```
class Base {
public:
    Base(); // add print "C1"
    Base(int num); // add print "C2"
    Base(const Base& b) // add print "CC"
    ~Base(); // add print "D"
    Base operator= (const Base& b); // + print "="
};
Base func(Base b) { // add print "func"
    Base a = b;
    return a;
}
```

- Assume that in each of the functions on the previous slide, we print out the string in red (C1, C2, CC, D, or =)
- Next slide:
  - Left column: Lines of code in a main() function
  - Right column: What will be printed out for each line
- Make sure you go through these carefully and understand!

Base a ;	C1
Base b (4) ;	C2
Base c = 4 ;	C2
Base d = c ;	CC
Base f (b) ;	CC
Base *g = &a ;	(nothing)
Base *h = new Base ;	C1
Base *i = new Base (c) ;	CC
Base *j = new Base [3] ;	C1 C1 C1
a = b ;	= CC D
func (a) ;	CC CC func D D
delete h ;	D
delete i ;	D
delete [] j ;	D D D

## Reminder

Of the four automatically generated class functions, only “operator = ” has a return type

```
class Base {  
public:  
    Base ();  
    Base (const Base& b)  
    ~Base ();  
    Base operator= (const Base& b);  
};
```

**Why?**

Also, if a different constructor is defined, the default one is *not* generated

## Constant classes

- A class object can be declared constant:

```
const Player p1;
```

- What functions can and cannot be called on a **const** class object?

- Functions must be declared **const** as such:

```
int Games() const;
```

```
const Player p1;
```

```
p1.Games(); // Okay
```

```
p1.SetName("Joe Smith"); // ERROR
```

## Constants in classes

- Classes can have constant member variables, but initializing the values is a bit awkward

```
class Base {
    const float pi = 3.14; // ERROR
    const float pi;
    Base();
    Base(float pival);
};
Base() : pi(3.14) { ... }
Base(float pival) : pi(pival) { ... }
```

## Static member variables

- Declaring a class member variable static makes it a global variable for that class

```
static int obj_count;
```

- No matter how many class objects are instantiated (even zero), there is only one instance of that particular variable
  - Think of it as belonging to the class, not to any particular class object
- Access:

```
Base::obj_count = 0;
```

```
Base::obj_count += 100;
```

Note: Accessed before any class object is declared

```
Base::Base()
{
    obj_count++;
}
Base::~Base()
{
    obj_count--;
}
Base::obj_count = 0;
Base b1;
Base b2;
Base *b3 = new Base;
delete b3;
std::cout <<
    Base::obj_count;
```

Static member variables are for keeping information about the class in general (not specific class objects)

P.S.

- You have to tell C++ which object file actually holds the (global) static class variable
- In one file, at global scope:

```
int Base::obj_count;
```

```
class Base {
public:
    static int obj_count;
    Base() { obj_count++; }
    ~Base() { obj_count--; }
};

int Base::obj_count = 0;

int main()
{
    Base a, b, c;
    cout << Base::obj_count << endl;
    return 0;
}
```

## Static member functions

- The preceding example assumed that the static variable was **public**, which probably isn't desirable. How to initialize and access if it's private?
- Static member functions
  - Can access static member variables (only)
  - Make **obj\_count** private
  - Define static functions **GetCount()**, **SetCount()**

```
class Base {
private:
    int obj_count;
public:
    static int GetCount()
    {
        return obj_count;
    };
    static void SetCount(int n)
    {
        obj_count = n;
    };
};
```

```
int Base::obj_count;
int main() {
    Base::SetCount(0);
    Base b1;
    Base b2;
    Base *b3 = new Base;
    delete b3;
    std::cout <<
        Base::GetCount(); }
```

# Meanings of **static**

Table 14-1

<b>Usage</b>	<b>Meaning</b>
Variable outside the body of any function	Global variable (scope: that file only)
Variable declared inside a function	Variable is permanent (keeps its value)
Function declaration	Scope of the function is limited to the file
Class member variable	One copy is created per class
Class member function	Function can only access static members