Week 2

NWEN 241 Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

- Admin stuff
- Derived and user-defined types
- C++ classes

People (1)

Course Coordinator

Alvin Valera
alvin.valera@ecs.vuw.ac.nz
AM401

NWEN 241 Office Hours: Mondays, 13:00-16:00

Lecturer

Sue Chard
sue.chard@ecs.vuw.ac.nz
CO254

Assistant Lecturer

Kirita-Rose Escott
kirita-rose.escott@ecs.vuw.ac.nz
CO254

People (2)

Tutors

Jakob Pfender
Deepak Singh
Carrie Yang
Guiying Yang
James Del Puerto
Keiran Batten
Mathias Ronimus
Xinyu Zhu
Alan Eir
Murugaraj Odiathevar

Class Representative(s)

Janel Huang
Felix Woodruffe
Sanjoo Manocha
Gene Culling

Programming Assignment #1

- Will be released today
- E-mail will be sent out once available in course wiki

Derived & User-Defined Types

Background

- Basic C/C++ data types
  - ✓ char : character
  - ✓ short, int, long, long long : integer
  - ✓ float, double, long double : floating point number
- Basic C++ only data types
  - ✓ wchar_t : wide character
  - ✓ bool : boolean
- Derived data types
  - ✓ Arrays
  - ✓ C strings
  - – Structures, unions, and C++ classes
- User defined data types
  - – New "types" including *enumeration* types

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to integral constants
- It is defined using the keyword `enum` and the syntax is:

```
enum enum_tag {
      identifier_0, identifier_1, ..., identifier_n
} variable_list;
```

  - ***enum_tag*** specifies the name of the enumeration type
  - ***enum_tag*** and ***variable_list*** are optional
  - If ***enum_tag*** is not specified, ***variable_list*** should be specified; otherwise, there is no way to declare variables using the unnamed enumeration type

Enumeration

- Enumeration is a user-defined data type that is used to assign identifiers to integral constants
- It is defined using the keyword `enum` and the syntax is:

```
enum enum_tag {
      identifier_0, identifier_1, ..., identifier_n
} variable_list;
```

  - The identifiers in the braces are symbolic constants that take on integer values from `0` through n
    - → `identifier_0` has value `0`,
    - → `identifier_1` has value `1`, and so on

Enumeration

- As an example, the statement:

```
enum colors { red, yellow, green };
```

creates three constants. `red` is assigned the value 0, `yellow` is assigned 1 and `green` is assigned 2

- It is possible to override the integer assignment, e.g.

```
enum colors { red = 3, yellow = 2, green = 1};
```

Enumeration

- If an identifier is assigned a value and subsequent identifiers are not assigned, the subsequent identifiers continue the progression from the assigned value

```
enum colors { red, yellow = 3, green, blue };
```

- `red` is assigned the value 0, `yellow` is assigned 3, `green` is assigned 4, and `blue` is assigned 5.

Enum Example 1 (Pure C)

```c
#include <stdio.h>

/* Declaration defines integer constants */
enum colors { red, yellow = 3, green, blue };

int main(void)
{
    /* Declaration defines variables of type enum colors */
    /* Can take values of red, yellow, green or blue */
    enum colors fgcolor = blue, bgcolor = yellow;

    printf ("%d %d\n", fgcolor, bgcolor);
    /* Will print 5 3 */

    return 0;
}
```

Enum Example 1 (C++)

```
#include <iostream>

/* Declaration defines integer constants */
enum colors { red, yellow = 3, green, blue };

int main(void)
{
    /* Declaration defines variables of type enum colors */
    /* Can take values of red, yellow, green or blue */
    enum colors fgcolor = blue, bgcolor = yellow;

    std::cout << fgcolor << " " << bgcolor << "\n";
    /* Will print 5 3 */

    return 0;
}
```

Enum Example (2)

```
/* This program uses enumerated data types
   to access the elements of an array */
#include <stdio.h>
int main(void)
{
    int August[5][7] = {{0,0,1,2,3,4,5},
                        {6,7,8,9,10,11,12},
                        {13,14,15,16,17,18,19},
                        {20,21,22,23,24,25,26},
                        {27,28,29,30,31,0,0}};
    enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
    enum week {week_one, week_two, week_three, week_four,
                   week_five};

    printf ("Monday the third week of August "
            "is August %d\n", August[week_three][Mon]);
}
```

C Structures

- A `C struct` is a derived data type composed of members that are each fundamental or derived data types.

- A single `C struct` would store the data for one object. An array of `C struct`s would store the data for several objects.

- A `C struct` can be defined in several ways as illustrated in the following examples.

Declaring a C Structure

- Syntax of the structure type declaration:

```
struct structure_tag {
        type1 member1;
        type2 member2;
        ...
} variable_list;
```

  - *`structure_tag`* specifies the name of the structure
  - *`structure_tag`* and *`variable_list`* are optional
  - If *`structure_tag`* is not specified, *`variable_list`* should be specified; otherwise, there is no way to declare variables using the unnamed structure type

Declaring a C Structure

- Syntax of the structure type declaration:

```
struct structure_tag {
        type1 member1;
        type2 member2;
        ...
} variable_list;
```

- Structure members can be
  - Basic data types
  - Derived and user-defined types
  - Pointers to basic, derived and user-defined data types

Examples

- struct declaration that only defines a type:

```
struct student_info { // named struct
      char name [20];
      int student_id;
      int age;
}; // does not reserve any space
```

- struct declaration that defines a type and reserves storage for variables:

```
struct student_info { // named struct
      char name [20];
      int student_id;
      int age;
} s, t; // reserves space for s and t
```

Examples

- Declaring a variable `current_student`

```
struct student_info current_student;
```

Above statement reserves space for:

- 20 character array,
- integer to store student ID, and
- integer to store age

- Declaring array of structures to store information of enrolled students in a class

```
struct student_info nwen241class[250];
```

Reserves space for 250 element array of records (structs) for students enrolled in NWEN241.

typedef

- The keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types

```
typedef existing_type_name new_type_name;
```

- Examples:

```
typedef unsigned long ulong;
typedef unsigned char uchar;

ulong i;  /* unsigned long i; */
uchar c;  /* unsigned char c; */
```

Creating New User-Defined Types

- Instead of typing `struct student_info` every time we declare a variable, we can define it as a new data type, e.g.

```
typedef struct { // unamed struct
      char name [20];
      int student_id;
      int age;
} StudentInfo;
```

- This makes `StudentInfo` a new user-defined type, and you can declare a variable as follows:

```
StudentInfo current_student;
```

New struct and data type

- If `struct student_info` has been previously defined, then we can create a new data type using `typedef` :

```
typedef struct student_info StudentInfo;
```

Accessing and Manipulating Structs

- We can reference a component of a structure by the **direct component selection operator**, which is a **period**, e.g.

```
strcpy(student1.name, "John Smith");
student1.age = 18;
printf("%s is in age %d\n", student1.name,
student1.age);
```

- The **direct component selection operator** has level 1 priority in the operator precedence
- The copy of an entire structure can be easily done by the assignment operator

```
student1 = student2;
```

Example

```
#include <stdio.h>
#include <string.h>

int main() {
   typedef struct student_info {
       char name[20];
       int student_id;
       int age;
   } StudentInfo;

   StudentInfo current_student;     // declare new variable using
                                    // new type StudentInfo
   struct student_info new_student; // declare using struct format

   // do stuff – see next slide
}
```

Example (continuation)

```
#include <stdio.h>
#include <string.h>
int main() {
   // declarations in previous slide
   …
   // create new student record
   strcpy(new_student.name , "John Smith");
   new_student.student_id = 300300300;
   new_student.age = 22;

   current_student = new_student;

   printf("Student name : %s\n", current_student.name);
   printf("Student ID   : %.9d\n", current_student.student_id);
   printf("Student Age  : %d\n", current_student.age);

}
```

Struct as Function Input Parameter

- Suppose a structure defined as follows

```
typedef struct {
   char name[20];
   double diameter;
   int moons;
   double orbit_time, rotation_time;
} planet_t;
```

Struct as Function Input Parameter

- When a structure variable is passed as an input argument to a function, all its component values are copied into the local structure variable

```c
/*
 * Displays with labels all components of a planet_t structure
 */
void
print_planet(planet_t pl) /* input - one planet structure */
{
      printf("%s\n", pl.name);
      printf("  Equatorial diameter: %.0f km\n", pl.diameter);
      printf("  Number of moons: %d\n", pl.moons);
      printf("  Time to complete one orbit of the sun: %.2f years\n",
             pl.orbit_time);
      printf("  Time to complete one rotation on axis: %.4f hours\n",
             pl.rotation_time);
}
```

- Not efficient when struct is a large object

Array of Structures

- An array of structures can be defined as follows:

```
typedef struct {
   int student_id;
   double gpa;
} student_t;

student_t student_list[50];

student_list[3].student_id = 300922023;
student_list[3].gpa = 8.0;
```

Array of Structures

- Can be simply manipulated as arrays of simple data types

	.student_id	.gpa
student_list[0]	300981683	6.5
student_list[1]	300961592	5.1
student_list[2]	300182652	7.3
student_list[3]	300922023	8.0
…	…	…
student_list[49]	300139414	9.0

student_list[0].gpa

student_list[3].student_id

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

```
union union_tag {
      type1 member1;
      type2 member2;
      ...
} variable_list;
```

  - *union_tag* specifies the name of the structure
  - *union_tag* and *variable_list* are optional
  - If *union_tag* is not specified, *variable_list* should be specified; otherwise, there is no way to declare variables using the unnamed structure type

Unions

- A union is like a struct, but the different fields take up the **same** space within memory

```
union union_tag {
      type1 member1;
      type2 member2;
      ...
} variable_list;
```

  - At any given time, only one of the members can contain a value
  - The size of a union is determined by the largest member

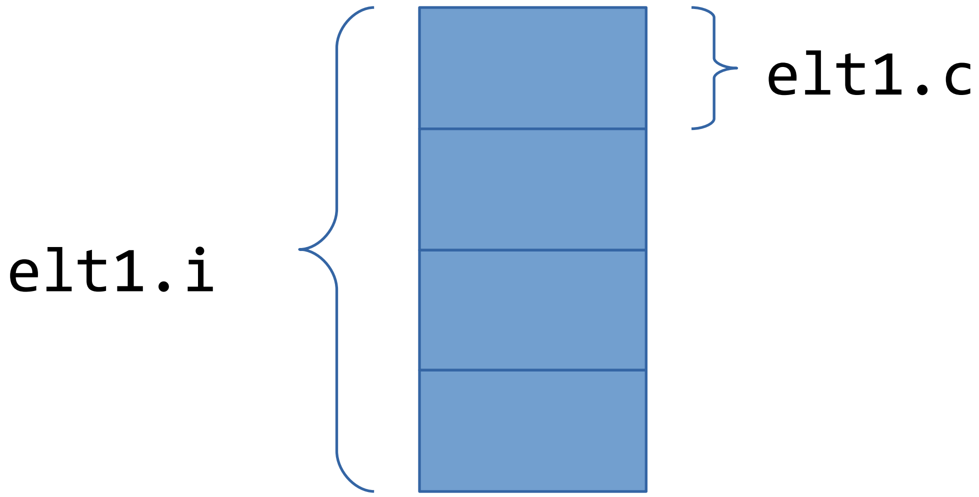
```
sizeof(union space) =
max(sizeof(member1),sizeof(member2),...)
```

Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up
32 bits (4 bytes):

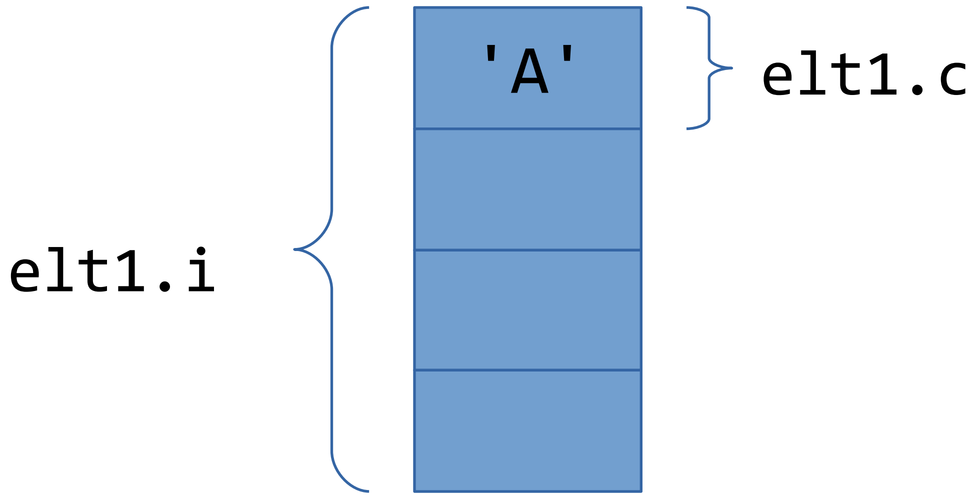


Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up 32 bits (4 bytes):

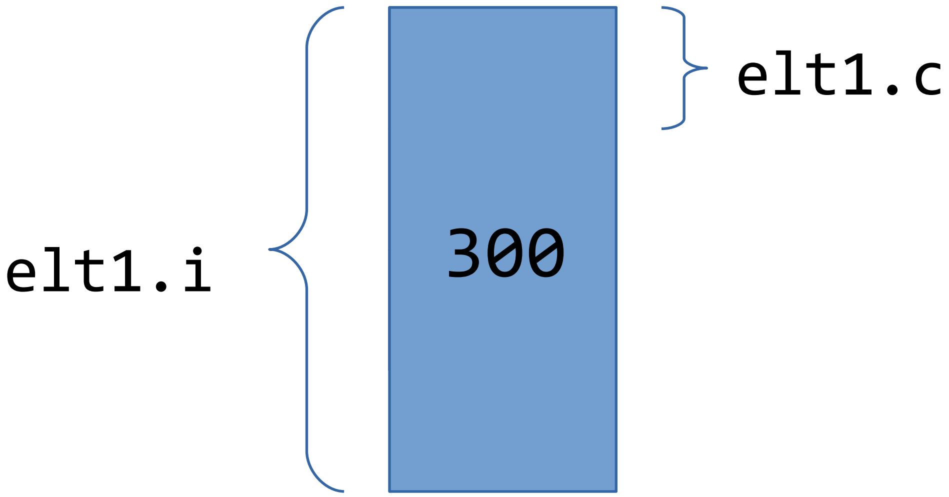


Example

```
union elt {
    int   i;
    char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;
```

Assuming an int takes up 32 bits (4 bytes):



Union Doesn’t Know What It Contains

- How should your program keep track whether `elt1` holds an `int` or a `char`?

- Basic answer:
  Use another variable to hold that info

```
union elt {
   int   i;
   char  c;
} elt1;

elt1.c = 'A';
elt1.i = 300;

if (elt1 currently has a char)
  …
```

Tagged Unions

- *Tag* every value with its case
- Pair the type info together with the union – implicit in other programming languages like Java

```
enum u_tag { INT, CHAR };
struct tagged_union {
   enum u_tag  tag;
   union {
      int   i;
      char  c;
   } data;
};
```

`enum` must be external to **`struct`**, so constants are globally visible.

`struct` field must be named.

C++ Classes

From Structure to Class

C++ Structures

- C++ structures are similar to C structures
- Same declaration syntax
- **But C++ structures can have functions as members and can be extended (supports inheritance)**

- Key disadvantage of C++ structures: **member variables and functions are all public**

Classes

C++ classes generalizes structures in an object-oriented sense:

- Classes are types representing groups of similar instances
- Each instance has certain fields that define it (instance variables)
- Instances also have functions that can be applied to them– also known as *methods* in OOP parlance
- Access to parts of the class can be limited

Classes allow the combination of data and operations in a single unit

Defining a Class

- A class is a collection of fixed number of components called **members** of the class

- General syntax for defining a class:

```
class class_identifier {
      class_member_list
};
```

  - `class_member_list` consists of variable declarations and/or functions

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

Member access specifiers

Possible specifiers:

- `private`
- `protected`
- `public`

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

Constructors

- Named after class name
- Similar to Java

When class performs dynamic memory allocation, **destructor** is also needed

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

Member functions

const at end of function specifies that member function cannot modify member variables

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

Member variables

Constant member variables can be initialized during declaration

Member Access Specifier

- C++ does not impose order on declaring `private`, `protected` and `public` members
- When member access specifier is not indicated, default access is `private`

- **Private members** – can only by accessed by member functions (and friends) and not accessible by descendant classes
- **Protected members** – can only be accessed by member functions (and friends) and inherited by descendant classes
- **Public members** – can be accessed by anyone and inherited by descendant classes

Constructors

- Default constructor: constructor without input parameters
  - When no default constructor is defined, the compiler will automatically generate one with empty body
- Properties of constructors:
  - Constructor name is the name of the class itself
  - No type: neither a value-returning function nor a void function
  - Class can have more than 1 constructor
  - If more than 1 constructor, constructors must have different formal parameter lists
  - Constructors are executed automatically when a class object is declared
    - Which constructor is executed depends on the actual parameters passed in the declaration

Example

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

```
Time myTime;
```

```
Time myTime(9, 0, 0);
```

How about these?

```
Time myTime();
```

```
Time myTime(9, 0);
```

Example

```
Time myTime;
```

```
Time myTime(9, 0, 0);
```

- These declarations creates instances of class `Time`
- Space automatically allocated for the objects in stack*
- Not possible in Java
  - Classes are instantiated using `new` statement

* We will talk more about stack (and heap) in dynamic memory allocation

Member Functions

- Member functions can be declared in 2 ways:
  - By specifying the function prototype
  - By specifying the function implementation

- How about in Java?

```
class Time {
public:
        void print() const;
        void set(int h, int m, int s) {
                hour = h;
                minute = m;
                second = s;
        }
        Time();
        Time(int, int, int);

private:
        int hour;
        int minute;
        int second;
};
```

Inline Functions

- Including the implementation of a function is an implicit *request* (to the compiler) to make a function **inline**
- When a function is inline, the compiler does not make a function call
  - The code of the function is used in place of the function call (and appropriate argument substitutions made)
  - Compiled code may be slightly larger, but will execute faster because function call overhead is avoided
- Can explicitly request to make member functions inline
  - Add `inline` keyword before return type in function declaration and definition

Inline Functions

- Not all inline requests are granted by the compiler
- Reasons for not granting inline requests:
  - Function contains a loop (for, while, do-while)
  - Function contains static variables
  - Function is recursive
  - Function return type is other than void, and the return statement doesn’t exist in function body
  - Function contains switch or goto statement

Implementing Functions Separately

- For member functions that are not implemented in the class declaration, they must be implemented separately

```
class Time {
public:
  void print() const;
  void set(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
#include <cstdio>

void Time::print() const
{
  printf("%2d:%2d:%2d", hour,
    minute, second);
}
```

Explicit Inline Request

- Add `inline` keyword before return type in function declaration and definition

```
class Time {
public:
  inline void print() const;
  void set(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
#include <cstdio>

inline void Time::print() const
{
  printf("%2d:%2d:%2d", hour,
    minute, second);
}
```

Implementing Constructors

- Implementation of constructs follow the same rules as member functions
- They can be implemented within the class declaration (implicit inline)

```
class Time {
public:
  …
  Time() {
    hour = 0;
    minute = 0;
    second = 0;
  }
  Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

Implementing Constructors

- Implementation of constructs follow the same rules as member functions
- They can be implemented within the class declaration (implicit inline)
- Or separately

```
Time::Time() {
    hour = 0;
    minute = 0;
    second = 0;
}

Time::Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
}
```

Another Syntax For Initializing Member Variables

```
class Time {
public:
  …
  Time() {
    hour = 0;
    minute = 0;
    second = 0;
  }
  Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
  }
  …
};
```

```
class Time {
public:
  …
  Time() : hour(0), minute(0),
    second(0) {}

  Time(int h, int m, int s) :
    hour(h), minute(m),
    second(s) {}
  …
};
```

Order is important: must be the same as order of declaration in class

Another Syntax For Initializing Member Variables

```
Time::Time() {
    hour = 0;
    minute = 0;
    second = 0;
}

Time::Time(int h, int m, int s) {
    hour = h;
    minute = m;
    second = s;
}
```

```
Time::Time() : hour(0), minute(0),
    second(0) {
}

Time::Time(int h, int m, int s) :
    hour(h), minute(m),
    second(s) {
}
```

Example: Accessing Members

```
class Time {
public:
      void set(int, int, int);
      void print() const;
      Time();
      Time(int, int, int);

private:
      int hour;
      int minute;
      int second;
};
```

```
// Creates instance using
// default constructor
Time myTime;

// Invokes member function
myTime.set(10, 30, 0);

// Is this allowed?
myTime.hour = 12;
```

Member access operator

Where to Declare and Implement Classes and Member Functions

- You may declare classes and implement the member functions in the same C++ source file

- Disadvantage: other sources will not be able to use the class

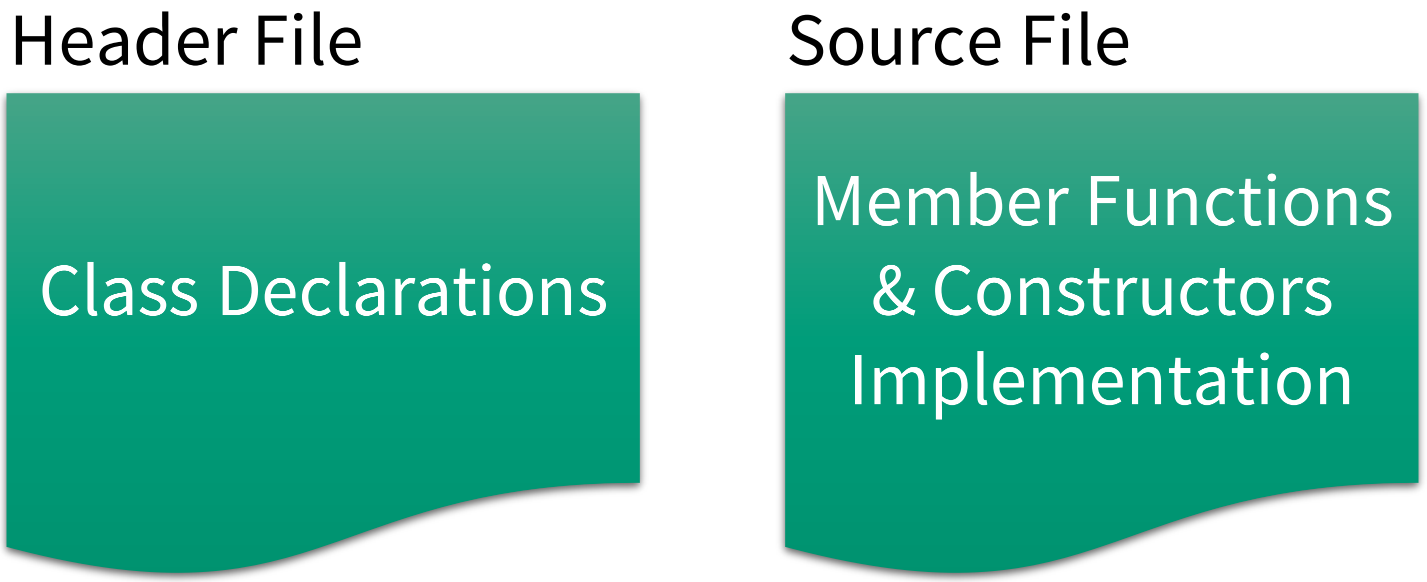
```
class Time {
public:
  void set(int, int, int);
  void print() const;
  …
};

void Time::set(int h, int m, int s)
{
  hour = h;
  minute = m;
  second = s;
}

void Time::print() const
{
  printf("%2d:%2d:%2d", hour, minute, second);
}
```

Where to Declare and Implement Classes and Member Functions

- Good programming practice is to declare the class in a header file
- Separate the implementation of the member functions (and possibly constructors) in another source file



Example

time.h

```
class Time {
public:
  void set(int, int, int);
  void print() const;
  Time();
  Time(int, int, int);

private:
  int hour;
  int minute;
  int second;
};
```

time.cc

```
…
#include "time.h"

Time::Time() : hour(0), minute(0), second(0) {
}

Time::Time(int h, int m, int s) :
 hour(h), minute(m), second(s) {
}

void Time::set(int h, int m, int s) {
  hour = h; minute = m; second = s;
}

void Time::print() const {
  printf("%2d:%2d:%2d", hour, minute, second);
}
```

Static Members

- C++ classes can contain static members
- A static member variable is a variable that is shared by all instances of a class
- Often used to declare class constants
- A static member function is function that can be invoked outside class instance

- Member functions and variables can be made static by using the `static` qualifier

Example

```
class Time {
public:
  void set(int, int, int);
  void print() const;
  static int getCounter();
  Time();
  Time(int, int, int);

private:
  int hour;
  int minute;
  int second;
  static int counter;
};
```

```
…
#include "time.h"

Time::Time() : hour(0), minute(0), second(0) {
  counter++;
}

Time::Time(int h, int m, int s) :
 hour(h), minute(m), second(s) {
  counter++;
}
…
// Initialize static member variable
int Time::counter = 0;

// Define static member function
int Time::getCounter()
{
  return counter;
}
```

Example (continued)

```
#include <iostream>
#include "time.h"

using namespace std;

…

int main(void)
{
    cout << Time::getCounter() << "\n";
    Time t1;
    cout << Time::getCounter() << "\n";
    Time t2(10,0,0);
    cout << Time::getCounter() << "\n";

    return 0;
}
```

Output:

```
0
1
2
```

Extending Classes

- Just like Java, C++ supports class **inheritance**

- **Sub Class or Derived class** – a class that inherits member fields from another class

- **Super Class or Base Class** – a class whose fields are inherited by sub class

- **The sub class is said to extend the base class**

Extending Classes

- Syntax of extending a single base class:

```
class subclass_name : access_mode baseclass_name {
      class_member_list
};
```

  - *subclass_name* is the identifier given to the sub class being declared
  - *access_mode* controls the access of inherited fields
  - *baseclass_name* is the identifier of the super class being extended

Example

Base Class:

```
class Animal {
public:
  const char *getName() const;
  void sleep();
  void eat(int food);
  Animal();
  Animal(const char *);

protected:
  int age;

private:
  char name[100];
};
```

Sub Class:

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Example

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Access mode – controls access to inherited fields

Base class member access specifier	Access mode		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	(Not accessible)		

Example

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Member functions specific to sub class Dog

Member function present in base class - will be **overridden** by sub class

Member variable specific to sub class Dog

Example

```
class Dog : public Animal {
public:
  int bark(int loudness);
  int bite(int strength);
  void run(int speed);
  void eat(int food);
  Dog(const char *n) : Animal(n) {}

private:
  int skills;
};
```

Constructor of Dog invokes appropriate super class constructor

- Unlike Java, C++ does not have super keyword for invoking super class constructor

Overriding Member Functions

- Declare the member function to be overridden in the class declaration
  - Prototype must be exactly the same as member function to override
- Provide overriding implementation

How to invoke base class function:

```
class Dog : public Animal {
public:
  …
  // Overridden function
  void eat(int food);
  …
};
```

```
Animal::eat(10);
```

Within member function

```
Dog d;
d.Animal::eat(10);
```

From an instance

Abstract Classes

- A class that contains at least one **pure virtual function** member
- Abstract classes cannot be instantiated
  - Similar to Java interfaces
- Pure virtual functions must be implemented by a sub class that need to be instantiated (**concrete)**

```
class Shape {
public:
  // Pure virtual function
  virtual float draw() = 0;

  // Virtual function
  virtual int getSides() {
    return 1;
  }
};
```

Multiple Inheritance

- C++ supports multiple inheritance

- Syntax for extending multiple classes:

```
class subclass_name : access_mode1 baseclass_name1, … ,
  access_modeN baseclass_nameN {
      class_member_list
};
```

Example

- Suppose that Human and Dog are existing classes

```
class Werewolf : public Human, public Dog {
public:
  void transform();
  …
  Werewolf(const char *n) : Human(n), Dog(n) {}

private:
  int transformCount;
  …
};
```

Member Function Clash

- What happens if base classes of a derived class have a common member function?

```
class A : {
public:
  void foo();
  …
};

class B : {
public:
  void foo();
  …
};
```

```
class C : public A, public B {
  …
};
```

Class C must override foo(), example:

```
class C : public A, public B {
public:
  void foo() {
    A::foo(); // Use A's implementation of foo!
  }
};
```

Other Aspects of C++ Classes (For Self-Study)

- Operator overloading
- Friend functions and classes

Next Lectures

- Strings

- Pointers