

CSE 2017 Data Structures and Lab

Lecture #1: Introduction to C++

Eun Man Choi

What is 'Structure'?

- In arrays all elements must be same data type
- Structure allows data of different types to be stored, accessed, manipulated using one variable name

```
struct struct_name {
    member_1_type member_1_name;
    member_2_type member_2_name;
    ...
    member_n_type member_n_name;
};
```



Example of Sructure Definition

```
struct employee {
  char name[50];
  int employee_number;
  int begin_year;
  float salary;
  char history[5000];
};
```

홍길동 2810 2011 40000000 동국대 졸, S전자 입사,



Declaring a variable of struct type

Simple Variable

```
employee Bob;
Bob.employee_number = code;
Bob.salary = rate;
strcpy (Bob.name, emp_name);
```

Array

```
employee department[SIZE];
department[k].employee_number = code;
if (department[k].begin_year > 1990) {...}
strcpy (department[k].history, emp_hist);
```



Declaring a variable of struct type

Pointer

```
employee* person;
...
person = new employee;
```

Must use struct pointer

```
person->employee_number = code;
amount = person->salary;
strcpy (person->name, emp_name);
```



Nested Structures

```
struct date {
   int month;
   int day;
   int year;
};
struct employee {
   char name[50];
   int employee_number;
   date begin;
   float salary;
   char history[5000];
   date terminated;
```



Nested Structures

```
employee Bob;
Bob.begin.month = start_month;
Bob.begin.day = start_day;
Bob.begin.year = start year;
date start;
Bob.begin = start;
employee department[SIZE];
if (department[k].terminated.year > 1990) {...}
employee* person;
person->begin.day = today;
```



Structures as function parameters

```
void main () {
  employee foreman;
  ...
  which_employee (foreman);
}

void which_employee (employee& emp) {
  cout << emp->employee_number << endl;
  return;
}</pre>
```

 Use of reference means that only address of employee is passed -- not all data (including 5500 characters)



Structure member functions

```
void employee::which_employee () {
struct employee {
                                        cout << employee number << endl;</pre>
 char name[50];
                                        return;
 int employee number;
 date begin;
                                      float employee::salary_portion (int div) {
 float salary;
                                        return (salary/div);
 char history[5000];
 date terminated;
                                      void main () {
                                        employee foreman;
                                        foreman.which_employee();
 void which_employee ();
                                        monthly = foreman.salary_portion (12);
 float salary_portion (int div);
                  member functions
```



Overloading Structure Functions

```
struct employee {
  char name[50];
  int employee_number;
  date begin;
  float salary;
  char history[5000];
  date terminated;
  float salary_portion (int div);
  float salary_portion (float pct);
};
```

```
float employee::salary_portion (int div) {
    return (salary/div);
}
float employee::salary_portion (float pct)
{
    return (pct * salary);
}
```



Calling Overloading Funcion

```
void main () {
  employee foreman;
  ...
  monthly = foreman.salary_portion (12);
  quarter = foreman.salary_portion (0.25);
}
```





Classes and Objects

- Original name for C++ was ``C with Classes''
- Class in C++ is a programmer-defined data type
- Class is a C++ structure
- Class defines data members and includes associated (member) functions
- Example:
 - a new data type software_engineer with several data items that distinguish a software engineer and several functions
 that are relevant to each software engineer



Data type: software_engineer

- data members:
 - name: the 20 character (or less) name of the software engineer
 - ssn: software engineer's social security num
 - start_date: the month, day, and year that the software engineer began working for the company
 - salary: the yearly compensation the software engineer receives
 - number_programs_written: number of programs written by the software engineer



Data type: software_engineer

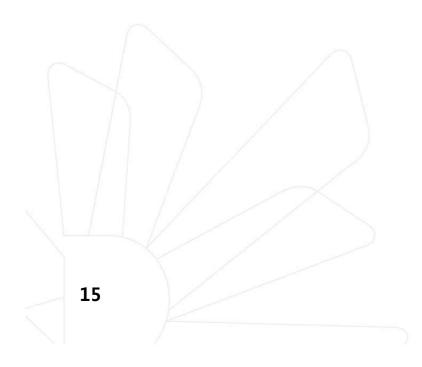
associated functions:

14

- create: create a new software engineer
- destroy: destroy existing software engineer
- change_salary: modify the software engineer's yearly compensation amount
- change_number_programs_written: modify the number of programs written by the software engineer
- report_name: report software engineer's name
- report_salary: report the software engineer's yearly compensation amount
- report_number_programs_written:
- report the number of programs written by the software engineer
- years_with_company: report number of years the software engineer has been working for the company

Data type: software_engineer

- Not every imaginable data member is included.
 - no phone number, years of computing experience
- Not all possible associated functions present
 - no functions to modify software engineer's social security number or starting date, month/day/year that software engineer began working for company
- May not be needed or may be added later





Defining a Class

- Class is defined similarly to a structure
- Class definition can contain data members, member functions, nested types
- Class preceded by the class keyword class

```
class software_engineer {
...
};
```



Defining a Class

 Data members can be any valid C++ data types including enumerated types, structure types, and even other classes

```
class software_engineer {
        char* name;
        int ssn;
        int start_date;
        int salary;
        int number_programs_written;
};
```

Data members are declared as in any structure



Member function

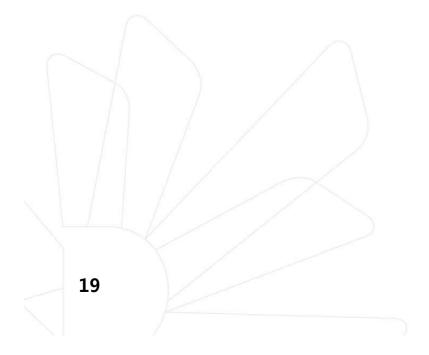
 In class definition via function prototype (return value, function name, and argument list)

```
class software engineer {
  char* name;
  int ssn;
  int start date;
  int salary;
  int number_programs_written;
  void create(char* who, int social_security_number, int begin, int
       amount, int programs);
                   // destroy function
  void destroy ();
  void change salary (int amount);
```



Member function

```
void change_number_programs_written(int programs);
  char* report_name ();
  int report_salary ();
  int report_number_programs_written ();
  int years_with_company (int today);
};
```





20

 function name is preceded by ClassName:: to indicate that function is member function of class ClassName

```
void software engineer::create (char* who, int social security number,
 int begin, int amount, int programs) {
 name = new char[20];
 strcpy (name, who);
 ssn = social_security_number;
 start date = begin;
 salary = amount;
 number_programs_written = programs;
void software_engineer::destroy () {
 delete [] name;
```



```
// member function to modify software engineer's
// salary
void software_engineer::change_salary (int amount) {
  salary = amount;
// member function to modify number of
// programs written by software engineer
void software_engineer:: change_number_programs_written (int
  programs) {
  number programs written = programs;
```



```
// member function to return engineer's name
char* software engineer::report name () {
   return (name);
// member function to return engineer's salary
int software_engineer::report_salary() {
   return (salary);
// member function to return number of
// programs written by software engineer
int software_engineer:: report_number_programs_written () {
   return (number_programs_written);
```



```
// member function to return number of years
// software engineer has been with company
int software_engineer:: years_with_company (int today) {
   int start_year, this_year;
   this_year = today - today/100*100;
   start_year = start_date - start_date/100*100;
   return (this_year - start_year);
```

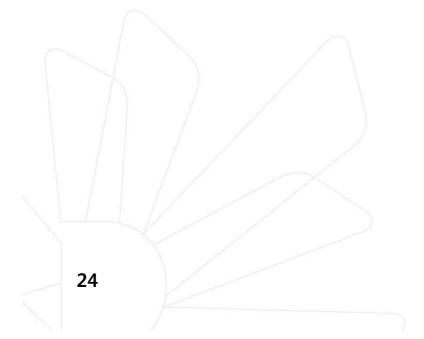


Creating an Object

- Class definition defines the class, but sets aside no memory
- Instance of class is called an object of that class
- Memory is allocated only when object of class is created
- Like structure variables, object is declared to be variable of appropriate class

software_engineer fred;

 Every object of class has its own set of data members and uses set of member functions of class





Class member access operators

 Data members and member functions of objects are accessed using class member access operators . and -> software engineer fred;

 data member ssn may be accessed by fred.ssn

- member function report_salary may be accessed by fred.report_salary ()
- Object definition by pointer
 software_engineer* fred = new software_engineer;
- data member ssn may be accessed by fred->ssn
- member function report_salary may be accessed by fred->report_salary ()



Main function

```
void main ()
   software_engineer fred;
   fred.create ("Fred", 408820391, 10185, 40000, 35);
   cout << "This software engineer's name is " <<
        fred.report_name () << endl;</pre>
  fred.change_salary (fred.report_salary () + 5000);
   cout << "This software engineer's salary is " <<
        fred.report_salary () << endl;
   fred.change_number_programs_written
        (fred.report_number_programs_written () + 1);
   cout << "The number of programs that " << fred.report_name () <<
      "has written is now " << fred.report_number_programs_written ()
      << endl;
```



Main function

```
cout << "The number of years that " << fred.report_name () << " has
  been with company is " << fred.years_with_company (122796) <<
  endl; fred.destroy ();
}</pre>
```

- Call to create function should appear immediately following object declaration, so that object is never used before it is initialized
- Since fred.start_date is 10185 (January 1, 1985) and parameter to years_with_company is 122796 (December 27, 1996), years_with_company will return 11 (96-85)



Private vs. Public

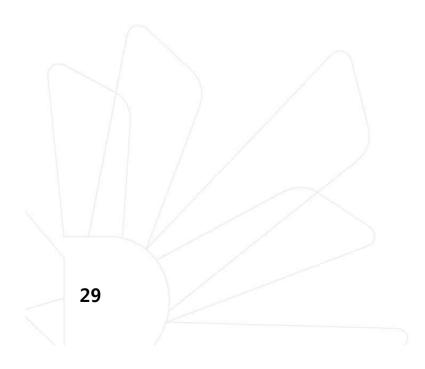
- Access specifier:
- public:
 - Declares that all data members and member functions that follow can be accessed anywhere in program
 - Possible to access or modify any data member of software_engineer object anywhere in program with statement like fred.salary = fred.salary + 5000;
 - Access specifier public typical for member functions of a class fred.change_salary (fred.report_salary () + 5000);
 - However, if data members of object are public, then data members can be viewed and modified without restriction



Private vs. Public

Private:

- Usually better to encapsulate each object by making its data members private
- If data members are private, they can only be viewed or modified by use of public member functions designed specifically for those purposes





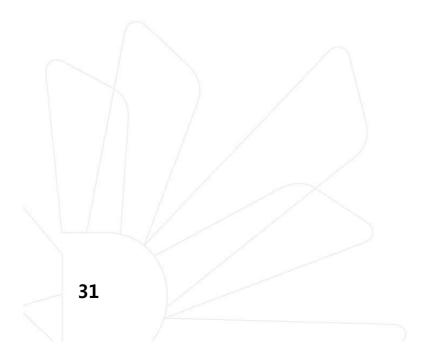
Encapsulation

```
class software engineer {
 private:
  char* name;
  int ssn;
  int start date;
   int salary;
   int number programs written;
 public:
  void create (char* who, int social_security_number, int begin, int
   amount, int programs);
   void destroy (); // destroy function
  void change_salary (int amount);
   void change_number_programs_written (int programs);
   char* report_name ();
   int report salary ();
```



Encapsulation

```
int report_number_programs_written ();
int years_with_company (int today);
int start_year ();
};
```





private:

- Declares that all data members that follow are private to software_engineer class
- Private data members of class can be accessed only by member functions of that class
- In class definition private access specifier is default
- Recommended that all data members be private unless there is some reason not to do this

```
software_engineer fred;
```

- attempt to access salary data member in statement cout << fred.salary << endl;
- not allowed because salary is private data member fred.change_salary (fred.report_salary () + 5000);



Private data members

- may be viewed or modified only via member functions set up for just that purpose
- Class member functions define interface between internal implementation of class (private data members and member functions) and rest of program
- If class implementer modifies internals of class, functions that reference objects of that class do not need to be changed
- Public member functions can be accessed by any other function that declares instance (that is, object) of that class
- Encapsulation process may be further enhanced by making some member functions private so that they may only be accessed by other member functions of that class



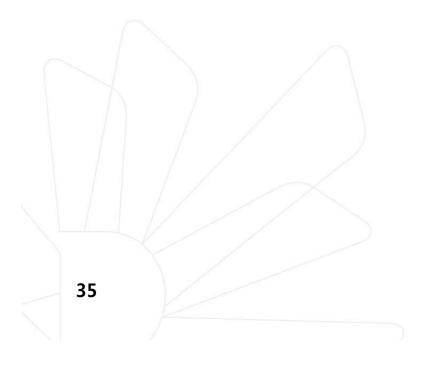
Private member function start_year:

```
class software engineer {
 private:
 char* name;
 int ssn;
 int start date;
 int salary;
 int number programs written;
 public:
 void create (char* who, int social security number, int begin, int
 amount, int programs);
 void destroy (); // destroy function
 void change_salary (int amount);
 void change_number_programs_written (int programs);
```



Private member function start_year:

```
char* report_name ();
int report_salary ();
int report_number_programs_written ();
int years_with_company (int today);
private:
  int start_year ();
};
```





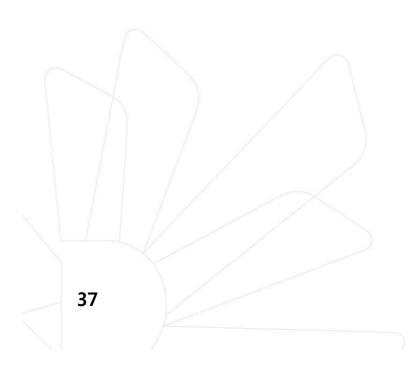
Private member function

```
// member function to return number of years
// software engineer has been with company
int software_engineer:: years_with_company (int today) {
   int this year;
   this_year = today - today/100*100;
   return (this_year-start_year());
// member function to return year software
// engineer joined company
int software_engineer::start_year () {
   return (start_date - start_date/100*100);
```



Private member function

- Any member functions used for internal implementation of class should be private
- Often used when member function would be useful only to other member functions for that class
- start_year uses knowledge about where year appears in start_date





Overloading

Member functions may be overloaded

```
class software_engineer {
  public:
  int report_salary ();
  void report_salary (int checks);
};
```

 First overloaded report_salary has no parameters and returns software engineer's current salary

```
// member function to return engineer's salary
int software_engineer::report_salary() {
  return (salary);
}
```



 Second overloaded report_salary has integer parameter checks and returns no value

```
// member function to print engineer's pay
// amounts based on number of annual checks
void software_engineer:: report_salary (int checks) {
  cout << "This software engineer receives " << checks << " checks per
  year each " << salary/checks << endl;
}</pre>
```



Inline member functions

```
class software_engineer {
 public:
   inline int report_number_programs_written ();
// function to return number of programs
// written by software engineer
inline int software_engineer:: report_number_programs_written () {
   return (number_programs_written);
```



Inline member functions

```
class software_engineer {
   public:
   inline int report_number_programs_written () {
      return (number_programs_written);
   };
};
```

Often inline member functions coded on single line



Using this Pointer

 In body of member function, pointer called this pointer always points at object for which function was called

```
// member function to modify software
// engineer's salary
void software_engineer:: change_salary (int amount) {
   salary = amount;
}
```

Statement

```
salary = amount;
is equivalent to statement
this->salary = amount;
```



Using this Pointer

 Potential use of this pointer is to create a pointer from one object of class to another

```
class software engineer {
  private:
 software_engineer* supv;
  public:
  void supervises (software_engineer& sofeng);
// member function to create pointer
// from object sofeng to its supervisor
void software engineer:: supervises (software engineer& sofeng) {
  sofeng.supv = this;
```



```
// application to use software_engineer class
void main () {
   software_engineer fred;
   fred.create ("Fred", 408820391, 10185, 40000, 35);
   software_engineer jennifer;
   jennifer.create ("Jennifer", 315243782, 112280, 60000, 200);
   jennifer.supervises (fred);
   fred.destroy();
   jennifer.destroy ();
```



- Member functions "create" and "destroy" created new software engineer object and destroyed existing software engineer object
- Certainly possible to construct such functions for each class
- C++ provides pair of special functions that do everything that create and destroy do
- Constructor and Destructor functions
- Constructor function is invoked automatically when object is defined
- Destructor function is invoked automatically when object goes out of scope



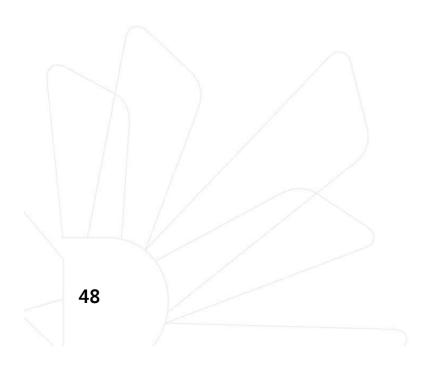
```
class software_engineer {
 private:
   char* name;
   int ssn;
   int start date;
   int salary;
   int number_programs_written;
 public:
   software_engineer (char* who, int
                                         social_security_number, int
   begin, int amount, int programs);
   ~software_engineer (); // destructor
};
```



```
// the constructor function
software engineer::software engineer (char* who, int
   social security number, int begin, int amount, int programs) {
   name = new char[20];
   strcpy (name, who);
   ssn = social_security_number;
   start_date = begin; salary = amount; number_programs_written =
   programs;
// the destructor function
software_engineer::~software_engineer() {
   delete [] name;
}
47
```



```
// application to use software_engineer class
void main () {
    software_engineer fred ("Fred", 408820391, 10185, 40000, 35);
    ....
}
```



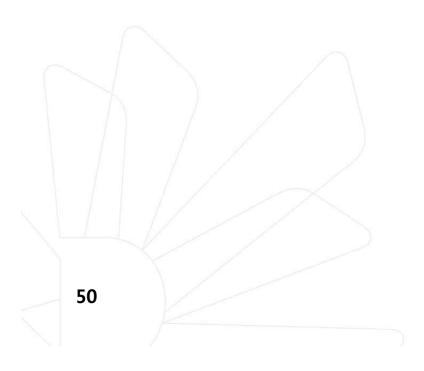


- Constructor function is typically first member function defined for each class
- Each constructor function of class has same name as class name
- No return type can be specified on prototype for constructor.
 Constructor function behaves as if it returns object of its class type
- When object is defined, necessary memory is allocated for object and constructor function is invoked
- Constructor function is typically used for any initialization required for object



```
classname::classname
(parameter-type parameter = default, ...,
parameter-type parameter = default);
```

where default is default value for that parameter





```
// constructor function
software engineer::software engineer (char* who = "Nameless", int
   social security number = 999999999, int begin = 10194, int amount
   = 40000, int programs = 0) {
   name = new char[20];
   strcpy (name, who);
   ssn = social_security_number;
   start date = begin;
   salary = amount;
   number_programs_written = programs;
```



```
// application to use software_engineer class
void main () {
   software_engineer fred ("Fred", 408820391,
        10185, 40000, 35);
   software_engineer mary ("Mary", 317264518,
        112480, 45000);
   software_engineer linda ("Linda", 487362514);
```



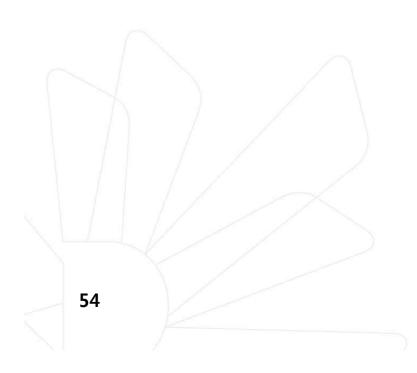
 If all parameters in constructor declaration have default argument values, constructor function serves as a default constructor as well

```
// application to use software_engineer class
void main ()
{
   software_engineer fred;
}
```



Destructor Function

- When object (instance of class) goes out of scope, Destructor Function is invoked
- Destructor function of class has name ``~Classname''
- There can be only one destructor function for each class.
 Destructor function cannot be overloaded
- Destructor function takes no parameters and returns nothing



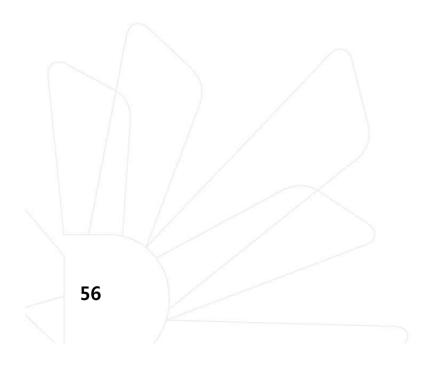


```
class software_engineer {
 inline ~software_engineer()
       delete [] name;
use of destructor
 fred.destroy ();
```



Templates

- Easily create generic functions or classes
 - Function template the blueprint of the related functions
 - Template function a specific function made from a function template
- Describes a function format that when instantiated with particulars generates a function definition
 - Write once, use multiple times





Function Template Example

 We rewrite functions Min(), Max(), and InsertionSort() for many different types

```
Indicates a template is being defined
```

```
Indicates T is our formal template
                                            parameter
                    template <class T>
                    T Min(const T &a, const T &b) {
                        if (a < b)
Instantiated functions
                            return a;
   will return a value
                                                Instantiated functions
                        else
   whose type is the
                                                   require two actual
    actual template
                                                    parameters of the
                            return b;
      parameter
                                                    same type. Their
                                                    type will be the
                                                    actual value for T
```



Min Template

Code segment

```
int Input1 = PromptAndRead();
int Input2 = PromptAndRead();
cout << Min(Input1, Input2) << endl;</pre>
```

 Causes the following function to be generated from our template

```
int Min(const int &a, const int &b) {
  if (a < b)
     return a;
  else
    return b;
}</pre>
```



Min Template

Code segment

```
double Value1 = 4.30;
double Value2 = 19.54;
cout << Min(Value1, Value2) << endl;</pre>
```

 Causes the following function to be generated from our template

```
double Min(const double &a, const double &b) {
  if (a < b)
     return a;
  else
     return b;
}</pre>
```



Min Template

Code segment

```
Rational r(6,21);
Rational s(11,29);
cout << Min(r, s) << endl;</pre>
```

 Causes the following function to be generated from our template

```
Rational Min(const Rational &a, const Rational &b) {
 if (a < b)
      return a;
 else
      return b;
```

Operator < needs to be defined for for the actual template parameter type. If < is not defined, then a compile-time error occurs



Class Template

- Rules
 - Type template parameters
 - Value template parameters
 - Place holder for a value
 - Described using a known type and an identifier name
 - Template parameters must be used in class definition described by template
 - Implementation of member functions in header file
 - Compilers require it for now





Array Template Class

```
Optional value is default constructed
template <class T>
 class Array {
 public:
      Array(int n = 10, const T &val = T());
      Array(const T A[], int n);
      Array(const Array<T> &A);
                                        Inlined function
      ~Array();
      int size() const {
         return NumberValues;
      Array<T> & operator=(const Array<T> &A);
      const T& operator[](int i) const;
      T& operator[](int i);
 private:
      int NumberValues;
      T *Values;
63,
```