Classes

Overview

- Structures
- Classes
  - Member functions
  - Constructors
- Enumerations
- Constness
- Constructor / Destructor, Copy / Assignment
- Operator overloading

Classes

- The idea:
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is plausible that it could be an object of a class
    - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a user-defined type that specifies how objects of its type can be created and used
  - In C++ (as in most modern languages), a class is the key building block for large programs
    - And very useful for small ones also
  - The concept was originally introduced in Simula67

Members and member access

- Class definitions (usually placed in header files)

```
class X {   // this class' name is X
  // data members       (they store information)
  // function members (they do things, using the information)
};
```

- Example

```
class X {      // class definition
public:
  int m;                                          // data member
  int mf(int v) { int old = m; m=v; return old; }  // member function
};             // end of class definition

X var;                      // create object var, variable of type X
var.m = 7;                  // access var's data member m
int x = var.mf(9);          // var's member function mf() set m of var
                            // to new value and returns old one
```

Classes

- A class is a new data type you can add to the language

```
class X {    // this class' name is X
  public:    // public members - that's the interface for users
             //  (accessible by all)
    // functions
    // data (better to keep private)
  private:   // private members --that's the implementation details
             // (accessible by member functions of this class only)
    // functions
    // data
};
```

Struct and class

Class members are private by default:

```
class X {
        int mf(int i);
        …
};
```

… means

```
class X {
  private:
        int mf(int i);
        …
};

X a;                    // create variable a of type X
int y = a.mf(3);        // error: mf is private (i.e., inaccessible)
```

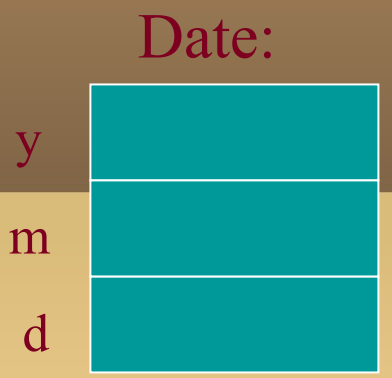
Struct and class

A **struct** is a class where all data members are public by default:

```
struct X {
        int m;
        …
};

        … means

class X {
  public:
        int m;
        …
};
```

struct are usually used for data-only structures where members can be accessed via "." operator

Struct

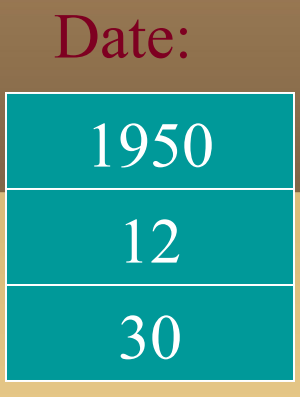



```
// simplest Date  class (only data)
struct Date {
    int y, m, d;   // year, month, day
};

Date my_birthday;       // create a Date-type variable (object)
// directly fill the structure
my_birthday.y = 12;     // wrong year
my_birthday.m = 30;     // wrong month
my_birthday.d = 1950; // wrong day

// It works but this is bad design. Not protected against wrong values.
```

Struct

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;     // year, month, day
};

Date my_birthday;       // create (instantiate) a Date variable (object)


// let's add external helper functions:

void init_day(Date& D, int y, int m, int d) {...};  // check for valid date
                                                    // and initialize D
void add_day(Date& D, int n) {...};       // increase the Date D by n days

usage:
init_day(my_birthday, 12, 30, 1950);  // must be protected against
                                      // impossible dates
```

Structs




```
// simple Date
//        initialization with constructor
//        provide some notational convenience
struct Date {
    int y,m,d;                  // year, month, day
    Date(int y, int m, int d);  // constructor: check validity of date
                                // and initialize
    void add_day(int n);        // increase the Date by n days
};

// your code
Date my_birthday;       // error: default constructor is not defined
Date my_birthday(12, 30, 1950);   // here it should throw exception
Date my_birthday(1950, 12, 30);   // ok
my_birthday.add_day(2);           // should change to January 1, 1951
my_birthday.m = 14;    // ouch! (now my_birthday is a bad date)
```

Classes

```
// simple Date (better access control)
class Date {
  private:
    int y,m,d;     // year, month, day (private now)
  public:
    Date(int y, int m, int d);  // constructor with parameters:
                                // check if date is valid and initialize
    void add_day(int n);        // increase the Date by n days
    // accessors   (public functions to access private data members)
    int month() { return m; }
    int day()     { return d; }
    int year()    { return y; }
};

//  your code
Date my_birthday(1950, 12, 30);   // OK. Object created and initialized
cout << my_birthday.month() << endl;    // we can read month
my_birthday.m = 14;                     // error: Date::m is private
```

Classes

- The notion of a “valid Date” is an important special case of the idea of a valid value
- We have to try to design your types so that values are guaranteed to be valid
  - Otherwise we have to check for validity all the time
- A rule which describes a valid value for your data type is called an “invariant”
  - The invariant for Date (“Date must represent a valid date in the past, present, or future”) is unusually hard to state precisely
    - Remember February 28, leap years, etc.
- Try to think hard of good invariant for your classes
  - that will saves you from poor buggy code

Classes

```
// simple Date (more implementation details. Data initializers)
class Date {
  public:
    Date(int y, int m, int d);  // constructor: check and initialize
    void add_day(int n);        // increase the Date by n days
    int month(); int day(); int year();  // accessors

  private:
    int y,m,d;     // year, month, day
};

// Constructor. Definition outside class. "::" means "member of class"

Date::Date(int yy, int mm, int dd) : y(yy), m(mm), d(dd) { … };
                                   // member initializers using " :"syntax

void Date::add_day(int n) { … };  // member function definition
```

Classes

```
// simple Date (more implementation details. Typical mistakes)
class Date {
  public:
    Date(int y, int m, int d);  // constructor: check for valid date and
                                // initialize
    void add_day(int n);        // increase the Date by n days
    int month(); int day(); int year();  // accessors

  private:
    int y,m,d;     // year, month, day
};

// Member function definition. Error: forgotten Date::
// this month() will be seen as a global function but not a class method

int month() {  return m; }   //  not the member, can't access members
```

Classes

```
// simple Date (what can we do in case of an invalid date?)
class Date {
  public:
    class Invalid { };                  // empty class. To be used as exception
    Date(int y, int m, int d);          // check for valid date and initialize
    …
  private:
    int y,m,d;                          // year, month, day
    bool check(int y, int m, int d);    // is (y,m,d) a valid date?
};

// Constructor definition
Date:: Date(int yy, int mm, int dd)
        : y(yy), m(mm), d(dd)           // initialize data members
{
    if (!check(y,m,d)) throw Invalid(); // check for validity and throw
                                        // some exception if not valid
}
```

Classes

- Why bother with the public/private distinction?

  Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To simplify debugging
    - Only a fixed set of functions can access the data
  - To allow a change of representation (interface)
    - You need only to change a fixed set of functions

Enumerations

- An **enum** (enumeration) is a very simple user-defined type, specifying set of values (its enumerators)
- For example:

```
enum Month {
    jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;        // declare m and set it to feb
m = 7;                // error: can't assign int to Month
int n = m;            // OK: we can get the numeric value of a Month
Month mm = Month(7);  // convert int to Month (without any checks)
```

Enumerations

- Simple list of constants (enum without name):

```
enum { red, green };          // the enum { } doesn't define a scope!
int a = red;                  // red is available here
enum { red, blue, purple };   // error: red defined twice
```

- User-defined type with list of constants

```
enum Color { red, green, blue,  … };
enum Month { jan, feb, mar,  … };

Month m1 = jan;
Month m2 = red;  // error red isn't a Month
Month m3 = 7;    // error 7 isn't a Month
int i = m1;      // OK: an enumerator is converted to its value, i==0
```

Enumerations – Values

- By default

```
// the first enumerator  has the value  0,
// the next enumerator has the value "one plus the value of the
// enumerator before it"
enum { horse, pig, chicken };   // horse==0, pig==1, chicken==2
```

- You can control numbering

```
enum { jan=1, feb, march, ... };                    // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 }; // <iostream>
int flags = fail+eof;          // flags == 10
stream_state s1 = flags; // error: can't assign int to a stream_state
stream_state s2 = stream_state(flags); // explicit conversion
                                       // (be careful!)
```

Classes

```
class Date {   // simple Date class (using  enumerator)
  public:
    enum Month {
          jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
    };
    Date(int y, Month m, int d) {...};  // constructor
    // …
  private:
    int y;          // year
    Month m;        // month
    int d;          // day
};
Date my_birthday(1950, 30, Date::dec); // error: 2nd argument not a
                                       // Month type (C++ type safety!)
Date my_birthday(1950, Date::dec, 30); // OK
```

Const

```
class Date {
  public:
    // …
    int day() const { return d; } // const function: can't modify object
    void add_day(int n) {...};    // non-const function: can modify
    // …
};

        Date   D(2019, Date::jan, 20);   // create object D
const Date cD(2019, Date::feb, 21);  // create const object cD

cout << D.day() << " – " << cD.day() << endl; // OK. Only reading
  D.add_day(1);          // Modify object content. OK.
cD.add_day(1);           // Error: cD is a const. Can't be modified.
```

Const

```
//
        Date   D(2019, Date::jan, 7);       // D is a variable
const Date cD(2019, Date::feb, 28);      // cD is a constant
cD = D;              // error: cD is const
cD.add_day(1); // error: cD is const
D = cD;              // fine. Copy const object to non-const
D.add_day(1);   // fine. D is non-const

cD.f();  // should work if and only if f() doesn't modify cD
         // how do we achieve that?
```

Const member functions

```
// To distinguish between functions that can work with "const"
// (immutable) objects
// and those that cannot so called "const member functions" exists.

class Date {
public:
    int day() { return d; }          // get the day
    void add_day(int n)  {...};      // move the date n days forward
    // …
};

const Date cD(2018, Month::nov, 4); // create const object cD
cD.add_day(4);          // error: can't modify "const" (immutable) object
int d = cD.day();       // also error !  day() can't access "const" object
```

Constness.

Constness *- the property of the object created as* **const Type**

Only ***const*** *member function can "work" with* ***const*** *object even if the function is not supposed to modify an object (like* **day()** *)*

```
class Date {

public:
    int day() const {return d;}    // get the day
    void add_day(int n) {...};     // move the date n days forward
    //…
};

const Date cD(2018, Month::nov, 4);   // create const object cD
int d = cD.day();  // OK. const function can access const object
```

Constness. const_cast.

It is a good practice to protect objects which you do not intend to modify creating them as **const.**

Nevertheless, C++ allows you "to break" the "constness" if it's really needed (usually it's a sign of the bad code design):

Method 1:

```
const Date cD(2018, Month::nov, 4);    // create const object cD

//  "Cast away" constness

int d = (const_cast<Date&>(cD)).add_day(3);   // OK
```

Constness. “mutable”

Method 2:

```
class Date {
public:
    int day() const {return d;}
    void add_day(int n) const {day += n; …. }
private:
    int y;                  // year
    Month m;                // month
    mutable int d;          // day
};
const Date cD(2018, Month::nov, 4); // create const object cD
cD.add_day(3);   // OK: you can modify mutable member of const cD
```

In principle, **month** and **year** also has to be done mutable, as **add_day()** can change them (e.g. for 31.12.2020)

Constructor, Copy, "=", Destructor

- Essential class components. Good practice – to provide them explicitly
  - Constructor (default: create members)
    - No default constructor if any other constructor is declared.
  - Copy constructor (default: copy the members)
  - Assignment operator (default: copy the members)
  - Destructor (default: destroy members)
- For example

```
Date d1;        // error: no default constructor (as one was defined)
Date d2(d1);    // ok: copy initialized (copy the elements)
d1 = d2;        // ok: assignment       (copy the elements)
```

Constructor

- **Constructor**
  - Member function with no return type and with the name of the class: **X();** or **X(int);** or ... for **class X**
  - Constructor is responsible for memory allocation
  - **Default constructor** – constructor with no arguments
  - If you do not define default constructor, it is automatically generated by compiler (Bad style!).
  - Automatic constructor call default constructors of all data members (built-in or user-defined) when object is created
  - If you've defined at least one constructor with arguments, you must also define default constructor

Copy Constructor

- **Copy Constructor**
  - Member function with no return type, with the name of the class and with reference to the class object as an argument: **X(const X& a)**;
- Copy constructor is used to create a copy of existing instance (object). E.g.

```
X a;        // create an object a of type X
            //..... do something with object a
X b(a);     // create an object b as copy of a
```

Copy Constructor

- Usually it is programmers' responsibility to copy all data-members of object **a** to newly created object:

```
X( const X& a ){
        // copy data-members of a to "this" object
        this->m1 = a.m1;   this->m2 = a.m2; // … etc
};
```

Here **m1**,**m2**,… are data-members of class **X**

Note: **"this ->"** could be omitted

Assignment operator =

- Assignment **operator**

  Member function which defines operator "**=**", with class object as return value (usually reference) with reference to the class object as an argument.

```
X& operator = ( const X& a ){
    // copy data-members of a to "this" object
    m1 = a.m1;   m2 = a.m2; // … etc.
    return(*this) // return reference to this object
};
```

  If data-members of this class (e.g. m1, m2 …) are private, one has to use "const" accessors as r-value (i.e. at right side of "=")

Assignment operator =

- Assignment operator allows to assign one existing instance (object) to another: **a2 = a1;**

  (here **a1** and **a2** are objects of class **X**)

- remark:

  X a2 = a1; // compiler calls copy constructor **X a2(a1);**

  X a2; // default constructor is called

  a2 = a1; // assignment operator is called

Destructor

- **Destructor**
  - Member function with no return type and with the name of the class perpended with "~" : **~X();**
  - Destructor is responsible for memory de-allocation
    - If you do not define destructor, it is automatically generated by compiler (Bad style!).
  - Automatic destructor call destructors of all data members (built-in or user-defined) when object is deleted.

Classes. Interface.

- What makes a good interface?
  - Minimal
    - As small as possible
  - Complete
    - … and not smaller
  - Type safe
    - Beware of confusing argument orders
  - **const** correct

Interfaces and “helper functions”

- Keep a class interface (the set of public functions) minimal
  - Simplifies understanding
  - Simplifies debugging
  - Simplifies maintenance
- BUT, when we keep the class interface simple and minimal, we often need extra “helper functions” outside the class (non-member functions)
  - E.g. **==** (equality operator) , **!=** (inequality operator),

Helper functions.

```
// Define "operator ==" (comparison) for objects of user-defined type Date
// Here this function is not Date class member.
// It is so called helper function

bool operator == (const Date& a, const Date& b) {
  return (a.year()  == b.year()  &&
          a.month() == b.month() &&
          a.day()   == b.day()
          );
}

// operator !=
bool operator != (const Date& a, const Date& b) { return !(a==b); }
```

Operator overloading

- You can define almost all C++ operators for your class (or enumeration)
  - that's often called "operator overloading". Example:

```
enum Month {
  jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov,dec
};

Month operator ++ (Month& m){          // prefix increment
   m = (m == dec) ? jan : Month(m+1);  // "wrap around"
   return m;
};

Month m = nov;
++m;        // m becomes dec
++m;        // m becomes jan
```

Operator overloading

- You can define only existing operators
  - e.g. **+ - * / % [] () ^ ! & < <= > >=**
- You can define operators only with their conventional number of operands
  - *e.g.* no unary **<=** (less than or equal) and no binary **!** (not)
- An overloaded operator must have at least one user-defined type as operand
  - **int operator + (int, int);** // error: you can't overload built-in **+**
  - **vector operator + (const vector&, const vector &);** // ok
- Advice (not a language rule):
  - Overload operators only with their conventional meaning
  - **+** should be addition, * be multiplication, **[]** be access, **()** be call, etc.
- Advice (not a language rule):
  - Don't overload unless you really have to

Operator overloading

- Overloaded unary operators defined in the class scope have no arguments ("this" object of the class will be modified)

  e.g. instead of **add_day(1):**

```
Date& operator ++ () {
    ++d;
    // .... some code to change month/year if it's needed
    return (*this); // returns self-reference
}
```

- Overloaded binary operators defined in the class scope have one argument: const reference to "right" operand. "left" operand is "this" object:

```
bool operator == (const Date& a) {
  return ( this->year()  == a.year()  &&
           this->month() == a.month() &&
           this->day()   == a.day() );
} // Remark: here this-> could be omitted
```

Operator overloading

- Overloaded unary operators defined as "helper" function (i.e. outside class scope) have one argument.

- Overloaded binary operators defined as "helper" function (i.e. outside class scope) have two arguments (see operators "==" and "!=" for **Date**).

  Remark:

  this is self-pointer (i.e. pointer to object itself.)

  (*this) is self-reference to the object of this class

  More about pointers – later

Next talk

- Class inheritance
- Type conversion