60. Classes

Objectives

- Understand the relationship between `struct` and `class`
- Create classes
- Write methods that accept parameters of basic type
- Write methods that accept object parameters
- Write methods that accept object reference parameters
- Write functions that accept object parameters
- Write functions that return objects
- Declare arrays of objects using the default constructor
- Declare and use pointers to objects

Object-oriented Thingies

So far you have been using C++ in a particular way. The programming style (or paradigm) you have been using is called **structured programming**.

You can recognize structured programming when you see blocks of code including but not limited to the following:

- branching such as if and if-else
- loops such as for-loop, while-loop
- functions

But this is not the only paradigm. The next paradigm we will focus on is called **Object-Oriented Programming, OOP**.

(There are many other styles/paradigms of programming. For instance, you will learn functional programming in CISS445. Then there's aspect-oriented programming …)

Both structured programming and OOP came out around 60s. Structured programming was popularized in the 80s through the introduction of the Pascal language by N. Wirth. (I learned that when I was in college.) Although OOP also came out around that time, it took off only around the 90s. So in terms of widespread use, OOP is actually pretty "new" or "recent". (But in the computer science world, anything older than 5 years is considered "old" ...)

While structured programming and OOP refer to using certain language features to **write** programs, structured analysis and object-oriented analysis refers to high level **analysis** of a program (usually using diagrams) with a view toward implementing the ideas using a structured or an object-oriented language respectively.

Some research shows that once a program goes beyond 100,000 lines, structured programming methodology breaks down because of the complexity of the project. (That's not to say that it cannot be done and that's also not saying that OO is the only way to handle large projects.)

Instead of going through all the philosophical reasons why OO is superior to old-style programming (such as structured programming), as many books do, we'll go through examples. It's pointless to discuss philosophy first because if you have not seen OO language features, it would be hard to judge. So ...

`Date` class version 1

We want to implement a `Date` type.

First I use `struct` to package up year, month, day and have some useful functions:

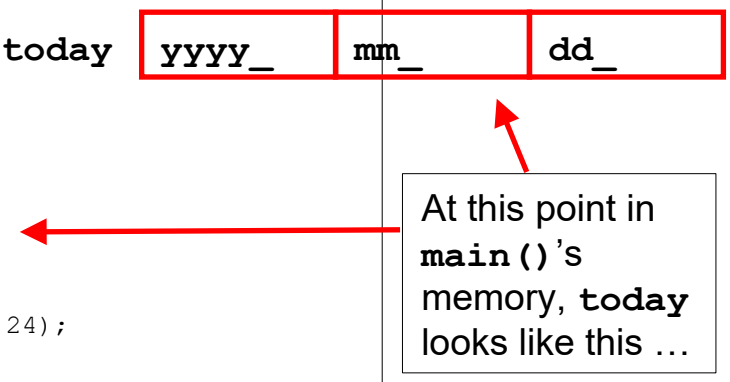
- `init` to set the month, day, year value of a `Date` variable
- `add_y` to add a year increment value to `Date` variable
- `add_m` to add a month increment value to `Date` variable
- `add_d` to add a day increment value to `Date` variable

Here we go (and this is nothing new) ...

```
// Date.h

#ifndef DATE_H
#define DATE_H

struct Date
{
    int yyyy_, mm_, dd_;
};

void init(Date &, int, int, int);
void print(const Date &);
void add_y(Date &, int);
void add_m(Date &, int);
void add_d(Date &, int);

#endif
```

```
// Date.cpp

#include "Date.h"

void init(Date & date, int yyyy, int mm, int dd)
{
    date.yyyy_ = yyyy;
    date.mm_ = mm;
    date.dd_ = dd;
}

void print(const Date & date)
{
    std::cout << date.yyyy_ << '/'
              << date.mm_ << '/'
              << date.dd_ << '\n';
}

void add_y(Date & date, int inc)
{
    date.yyyy_ += inc ;
}

// etc.
```

```
// main.cpp

#include "Date.h"

int main()
{
    Date today, yesterday;

    init(today, 2014, 12, 25);
    print(today);

    init(yesterday, 2014, 12, 24);
    print(yesterday);

    return 0;
}
```



Now in your `main.cpp`, go ahead and call `add_d(yesterday, 1)` and then `print(yesterday)`.

Note that the above `main()` that uses the `Date` type (`Date.h` and `Date.cpp`) does not need to know about the details (i.e., the members) of a `Date` variable: you do not need to know that a `Date` variable such as `today` contains `yyyy_`, `mm_`, `dd_`.

This is the beginning of the concepts of **information hiding** and **encapsulation**.

Specifically, in the above, we collect up the concepts of a year, a month and a day and create the concept of a date. That's **encapsulation**.

Encapsulation allows you to **think at a higher level** of abstraction. It's easier to focus on higher level concepts once lower level details are encapsulated.

Note that in version 1 the functions are not tied to `Date` struct.

We want to think of the functions as "belonging" to the `Date` struct. You can actually put the functions into the `struct` definition to get **member functions** .

So we get a `struct` with **member variables** and **member functions** in one package.

After moving the functions into the `struct`, you need to make some modifications.

Exercise. Complete the `add_m` and `add_d` functions.

`Date` class version 2

Make the following changes:

In `Date.h`: move the function prototypes into the `struct` and remove all first parameters which are `Date` reference parameters.

In `Date.cpp`: remove all first parameters which are `Date` reference parameters and remove "`date.`" in the body of all functions.

In `main.cpp`: move all `Date` variables which are first arguments outside the function call. Add a dot after these `Date` variables.

Here are the files with the changes:

```
// Date.h

#ifndef DATE_H
#define DATE_H

struct Date
{
    int yyyy_, mm_, dd_;

    void init(Date &, int, int, int);
    void print(const Date &);
    void add_y(Date &, int);
    void add_m(Date &, int);
    void add_d(Date &, int);
};

#endif
```

```
// Date.cpp

#include "Date.h"

void Date::init(Date & date, int yyyy, int mm, int dd)
{
    date.yyyy_ = yyyy;
    date.mm_= mm;
    date.dd_= dd;
}

void Date::print(const Date & date)
{
    std::cout << date.yyyy_ << '/'
              << date.mm_ << '/'
              << date.dd_<< '\n';
}

void Date::add_y(Date & date, int inc)
{
    date.yyyy_+= inc ;
```

```
}

// etc.
```

```
// main.cpp

#include "Date.h"

int main()
{
    Date today, yesterday;

    today.init(today, 2014, 12, 25);
    today.print(today);

    yesterday.init(yesterday, 2014, 12, 24);
    yesterday.print(yesterday);

    return 0;
}
```

Here's what you get:

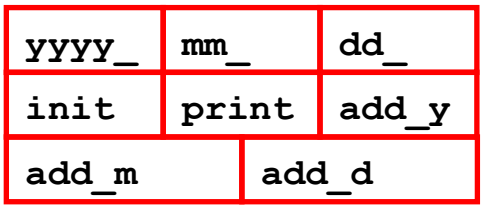
```
// Date.h

#ifndef DATE_H
#define DATE_H

struct Date
{
    int yyyy_, mm_, dd_;

    void init(int, int, int);
    void print();
    void add_y(int);
    void add_m(int);
    void add_d(int);
};

#endif
```

```
// Date.cpp

#include "Date.h"

void Date::init(int yyyy, int mm, int dd)
{
    yyyy_ = yyyy;
    mm_ = mm;
    dd_ = dd;
}
```

```
void Date::print()
{
    std::cout << yyyy_ << '/'
              << mm_ << '/'
              << dd_ << '\n';
}

void Date::add_y(int inc)
{
    yyyy_ += inc ;
}

// etc.
```

```
// main.cpp

#include "Date.h"

int main()
{
    Date today, yesterday;

    today.init(2014, 12, 25);
    today.print();

    yesterday.init(2014, 12, 24);
    yesterday.print();

    return 0;
}
```

today



At this point in **main()**'s memory, **today** looks like this …

As stated earlier, we put the function prototypes into the `struct` definition.

Note that in `Date.cpp`, we need to put **`Date::`** in front of the implementation of the member functions. This is because the `Date struct` creates a scope. Therefore outside the `struct`, you need to say "the `init` function **of `Date`**". If you just call it `init`, C++ will think of some `init` function outside the `Date struct`. This means that C++ actually allows you to do this:

```
// Date.h

...

struct Date
{
    ...
    void init(int, int, int);
    ...
};

void init(int, int, int);
```

```
#endif
```

In the code, you can see the concept of "variable calls a function". For instance, you see `today.init(2014, 12, 25)`.

In the implementation of `Date::init`, you see `yyyy_`, `mm_`, and `dd_`. Which `yyyy_` are we talking about? There's the `yyyy_` of `today` and the `yyyy_` of `yesterday`!!! Well, in the body of `Date::init`,

`yyyy_`	refers to `yyyy_` of the variable calling `init()`
`mm_`	refers to `mm_` of the variable calling `init()`
`dd_`	refers to `dd_` of the variable calling `init()`

```
...

int main()
{
    ...

    today.init(2014, 12, 25);

    ...
```

While executing **`today.init(2014, 12, 25)`**, the **`yyyy_`** refers to the **`yyyy_`** of **`today`** …

```
void Date::init(int yyyy, int mm, int dd)
{
  yyyy_ = y0;
  mm_ = m0;
  dd_ = y0;
}
```

```
...

int main()
{
    ...

    yesterday.init(2014, 12, 24);

    ...
```

While executing **`yesterday.init(2014, 12, 24)`**, the **`yyyy_`** refers to the **`yyyy_`** of **`yesterday`** …

Note that

- functions are part of `Date`
- `Date` variables can invoke `Date` member functions
- Member functions "know" which variables invoked them and which `yyyy_`, `mm_`, `dd_` they should work with.
- For instance, `today.mm_` : Refers to the member variable `mm_` in `today`

- For instance, `today.print()` : Calls/invokes the `print()` member function of `today` and while executing the function, `yyyy_` refers to the `yyyy_` of `today`.

Make sure you see the differences between version 1 and 2!!!

Exercise. Write a `struct Robot` with the following members:

- `int x`: the x-coordinate of the position
- `int y`: the y-coordinate of the position
- `void init(int a, int b)`: sets the `x`, `y` values to `a`, `b` respectively
- `void print()`: prints the `x`, `y` values
- `void moveLeft(int steps)`: sets `x` to `x - steps`
- `void moveRight(int steps)`: sets `x` to `x + steps`
- `void moveUp(int steps)`: sets `y` to `y + steps`
- `void moveDown(int steps)`: sets `y` to `y - steps`

Write a program to test your `Robot` structure:

- Declare and initialize `c3p0` of `Robot` type to `{5, 5}`
- Call `c3p0.print()`
- Call `c3p0.moveLeft(2)`
- Call `c3p0.print()`
- Call `c3p0.moveRight(3)`
- Call `c3p0.print()`
- Call `c3p0.moveUp(-4)`
- Call `c3p0.print()`
- Call `c3p0.moveDown(5)`
- Call `c3p0.print()`

In the above exercise, you will have statements such as

```
c3p0.moveRight(3);
```

You can and **should** think of the `Robot` variable `c3p0` as having the **ability** to `moveRight` by 3 on its own. In other words, you want to think of the `Robot` as having some **autonomous ability** to perform the `moveRight` operation. That is one very important philosophy behind packaging functions into a `struct` so that they become members (i.e. member functions) as the `struct` variable. In fact, historically, the earliest example of a programming language that allows this is from the MIT AI lab in the late 50s.

This is very different from say if you do

```
Robot_moveRight(c3p0, 3);
```

Written this way, it reads more like **your** program is **controlling**

the `Robot c3p0` and making him/her/it move.

For complex software engineering, we want to analyze and develop software not by focusing on data and functions separately, but rather by focusing on data **with** their functions. This is achieved in version 2 above. For instance, look at

```
today.init(12, 25, 2003);
```

You think of the `init()` function as being part of `today`, just like you have `today.mm_`, `today.dd_`, `today.yyyy_` etc.

In fact, to emphasize again that `today` has the autonomous ability to execute `init()`, I will frequently say:

`today` invokes `init()`

and not "your program invokes the `init()` of `today`".

Exercise. Add the following member function in the header file of your `Date`:

```
// Date.h

struct Date
{
    int yyyy_, mm_, dd_;
    ...
    void add_m_d(int, int);
};
```

And of course in your `Date.cpp`, you have:

```
// Date.cpp

#include "Date.h"

...

void Date::add_m_d(int inc_mm, int inc_dd)
{
    mm_ += inc_mm;
    dd_ += inc_dd;
}
```

Now … and here's the point of this exercise … instead of writing code to directly modify the member variables, use member functions instead because you already have member functions to increment the `mm_` and the `dd_` member variables. In other words, `Date` variables can call their member functions and in a member function, you can call another member function.

Exercise. Check your `add_y`, `add_m`, `add_d` functions and verify that the `Date` variable is correct after the increments. For instance if the month is greater than 12, what must you do? Also, assuming a `Date` is correct, after calling add_d, what must you do if the dd_ is greater than 40? Did you check the number of days for February for leap years and non-leap years?

Exercise. Now implement the following function that allows you to increment the `yyyy_`, `mm_`, `dd_` members by 3 integer values passed into the following function:

```
// Date.h

struct Date
{
    int yyyy_, mm_, dd_;
    ...
    void add_y_m_d(int, int, int);
};
```

Exercise. The above passes in 3 integer values to be used for incrementing the `yyyy_`, `mm_`, `dd_` member values of the `struct` variable invoking the function. Well … a `Date` variable already contains 3 values, the `yyyy_`, `mm_`, and `dd_`. So why not pass in a `Date` value instead? Now implement the following function that allows you to increment the `yyyy_`, `mm_`, `dd_` members by a `Date` value passed into the following function:

```
// Date.h

struct Date
{
    int yyyy_, mm_, dd_;
    ...
    void add_date(const Date & d);
};
```

Exercise. Create a function in `Robot` called `moveLeftUp()` that calls `moveLeft()` and `moveUp()`. Test!!!

Structured vs OO

We don't have classes and objects yet (what are they anyway?) But even with what we have done so far, we can ask ...

Why? Why are we writing code this way.

The main reason is to control complexity …

Structured programming focuses on functions. You start with a goal/task (big function) and subdivide task into simpler subtask (simpler functions). Data is passed between functions. Structured programming and structured analysis tend to separate the computational task and the data involved.

OOP and OO analysis tend to focus on both data and functions together at the same time. At a higher level of design and engineering, we think of OO analysis and design as finding objects and their responsibilities (functions), i.e., what the objects do. So in the discipline of OO analysis and design, we tend to think of and find high level concepts and analyze what the variable (i.e., the objects) associated with that high level concept should be be capable of doing.

Let me give you an example.

Suppose I work for a bank and I need to print a daily report of all the transactions (deposit to an account, withdrawal from an account, closing of an account, opening of an account, changing the address of the customer, etc.)

Structured thinking means this: I want to print a report. To do that, I need to read the database for all transaction. For each transaction, there's an transaction ID, the customer ID, a transaction type, etc. Now for each customer ID, I will read the customer file, look for the matching customer ID, read his firstname and lastname, etc. Next, for this transaction, I have to look up the meaning of the transaction from the transaction type (for instance say 1 means open account, 2 means close account, 3 means deposit, etc.) I get the description of the transaction ID from some file, so if the transaction is 1, the description for this entry is “Open Account”, “Close Account”, etc. This style of thinking is called structured analysis and design. And the type of code you write to reflect this way of thinking tends to have a certain look – structured programming code.

The object-oriented way is different. You want to think of a report as being made up of transaction objects. Each transaction object of course has some type and some description (“Open Account”, “Close Account”, etc.), but the main program will not get the transaction description. Rather for each transaction, you will probably call

```
transaction.getDescription()
```

in other words, each transaction object has the ability to provide useful computations. You think of “telling the transaction to do its work on its

own" and you wait for the description to be returned (probably a string). Also, each transaction knows (on its own) that it's a transaction for a particular customer. So to know more about the customer, you would ask transaction to tell you who is the customer:

```
Customer customer = transaction.getCustomer();
```

And with this customer object now available, you ask the customer to give you his/her first and last name (probably string) say for printing:

```
std::cout << customer.getLastName() << ", "
          <<  customer.getFirstName()
          << …
```

So as you can see, the style of thinking (the OO analysis and design) will give rise to very different code:

> for each transaction in today's collection of transactions:
> I ask the transaction for the transaction description and print it.
> I then ask the transaction for the time when it was created and print the date and time.
> I then ask the transaction for firstname and lastname of the customer responsible for this transaction this will result in the transaction asking the cusomter involved for his/her firstname and lastname. The customer gives his/her firstname and lastname to the transaction and the transaction passes the firstname and lastname back to me.

Note that when I ask a transaction for the firstname and lastname of the customer, the transaction talks to the customer (object) and asks the customer to provide his/her first and last name. When an object needs to perform some computation it will either do it himself/herself or will talk to some other object to do the work.

You don't need to know the full picture of structured thinking and OO thinking. The above is to give you a quick overview of the different philosophy between the two.

The goal in this course is to teach you the basics of OO syntax and the language features. You will NOT be able to engineer beautiful and well constructed OO systems immediately. In fact it takes a very long time to be an expert in either the structured or the object-oriented style of analyzing and designing systems. Right now, my goal is to give you the basic syntax and language features of an OO language. It will take many years before you will become a true OO guru because the road is challenging, tough, and is just extreme fun. (Which, frankly speaking, is a good thing – if OO is something you can learn in 1-2 years, you wouldn't expect it to be fun or worth much, would you?)

Information Hiding and Encapsulation

We also want to control another type of complexity.

There are many ways to implement a `struct` and the member functions. Outsiders using the `struct` should not have access to its member variables.

In fact outsiders should **not** even know how things are implemented in the `struct`. For instance, in the `Date struct`, you implemented the member variables with three integer variables names `yyyy_`, `mm_`, `dd_`.

Information hiding refers to hiding implementation details from users of your struct library which can very well change in the future.

Look at the struct definition:

```
// Date.h

...

struct Date
{
    int yyyy_, mm_, dd_;
    void init(int, int, int);
    void print();
    void add_y(int);
    void add_m(int);
    void add_d(int);
};

...
```

With complete documentation of the member functions, we can use the Date type without actually knowing the implementation of the data member variables or the member functions.

If someone wants to use your Date library in a different way, he/she just has to submit a feature request to you. For instance suppose he/she wants to have a print member function that prints with a '-' separating the year, month, and day instead of a '/'. You can just add a new print member function:

```
// Date.h

...

struct Date
{
    int yyyy_, mm_, dd_;

    void init(int, int, int);
    void print();
    void print_with_dash();
```

```
    ...
};

...
```

Now why should users of your library not know about the existence of `yyyy_`, `mm_`, `dd_`??? Is that just paranoia?

The problem with letting outsiders know about the internal implementation of the `Date` struct (i.e., information about the member variables) is that someone with itchy fingers will write something like this:

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

int main()
{
    ...

    Date today;

    ...

    std::cout << today.yyyy_ << '-'
              << today.mm_ << '-'
              << today.dd_ << '\n';

    ...

    return 0;
}
```

Now you might say … “So what? It does work, correct?”

Well … here's the problem. Say you have exposed the internal implementation of your struct. And you let others in your company use the member variables. After some time, the code in your company will have lots of access to the internals of the `Date` struct:

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

int main()
{
    ...

    Date today;

    ...

    std::cout << today.yyyy_ << '-'
              << today.mm_ << '-'
              << today.dd_ << '\n';
```

```
    ...

    if (today.yyyy_ > 2020)
    {
        ...
    }

    for (int i = 1; i < yesterday.dd_; ++i)
    {
        ...
    }

    return 0;
}
```

One fine day, you realize that it's a waste to use 3 integers to implement the internals of your `Date` struct. You can clearly fit the `yyyy_`, `mm_`, `dd_` into a single integer!!! In other words it would have been better if you did this:

```
// Date.h

...

struct Date
{
    int yyyymmdd_;

    void init(int, int, int);

    ...
};

...
```

```
// Date.cpp

...

void Date::init(int yyyy, int mm, int dd)
{
    yyyymmdd_ = yyyy * 10000 + mm * 100 + dd;
}

...
```

For instance, whereas the members variables of the old `today` looks like this:

```
today.yyyy_ = 2014
today.mm_ = 12
today.dd_ = 25
```

the new `today` would look like this:

```
today.yyyymmdd_ = 20141225
```

Wow … not bad! If your main program (say something that prints a financial report) uses 100 `Date` variables, you're saving a lot of memory. But … there's a problem … remember this program:

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

int main()
{
    ...

    Date today;

    ...

    std::cout << today.yyyy_ << '-'
              << today.mm_ << '-'
              << today.dd_ << '\n';

    ...

    if (today.yyyy_ > 2020)
    {
        ...
    }

    for (int i = 1; i < yesterday.dd_; ++i)
    {
        ...
    }

    return 0;
}
```

It uses `yyyy_`, `mm_`, and `dd_`. After changing your Date to the new one that uses `yyyymmdd_` as member variable, the above program will not compile. If there are 5 appearances of the old-style member variables, that's OK. But if your company has 10 people writing code that uses the internals of the old `struct` for the past 6 months and 100000 lines of code is written, there will be a LOT of work changing everyone's code!!!

It's a lot better if they had asked you for the following features:

`print_with_dash()`	prints Date with '–' instead of '/'
`get_year()`	computes and returns the value of the year
`get_day()`	computes and return the value of the day

and written their code like this:

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

int main()
{
    ...

    Date today;

    ...

    today.print_with_dash();
    ...

    if (today.get_year() > 2020)
    {
        ...
    }

    for (int i = 1; i < yesterday.get_day(); ++i)
    {
        ...
    }

    return 0;
}
```

Now, regardless of whether you're implementing the `Date` struct with `yyyy_`, `mm_`, `dd_` or with `yyyymmdd_`, their code will ALWAYS work.

The point: if you provide features through functions and tell others not to use the internal member variables, then when you need to change your code, you need only need to change the members functions in your `Date` library. Period.

Exercise. Add a `get_year()`, `get_month()`, `get_day()` in your `Date` library. Test it with this:

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

int main()
{
    Date today;
    today.init(2014, 12, 25);
    std::cout << today.get_year() << '\n';
    std::cout << today.get_month) << '\n';
    std::cout << today.get_day() << '\n';

    return 0;
}
```

Exercise. Add a `set_year(int)`, `set_month(int)`, `set_day(int)` in your `Date` library. Test it with this:

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

int main()
{
    Date today;
    today.init(2014, 12, 25);
    today.set_year(1770);
    std::cout << today.get_year() << '\n'; // 1770

    ...

    return 0;
}
```

Member functions to get and set basic values in the struct variable (regardless of how you implement the internals) are called **getters** and **setters**. Getters are member functions that get basic lower level concepts. Setters are members functions that set some internal variables to some other values.

The struct allows us to put member variables underneath another variable (example: hiding `yyyy_`, `mm_`, `dd_` inside today) besides putting functions inside struct variables. So it's hiding things in a header file. But …

The struct does **not** enforce **strict information hiding**.

Sure, you can tell others not to use `yyyy_`, `mm_`, `dd_`. But they can be stubborn and insist on using them.

So …

`Date` class version 3 (the real deal)

Now do this:

```
// Date.h

#ifndef DATE_H
#define DATE_H

class Date
{
public:
    void init(int, int, int);
    void print();
    void add_y(int);
    void add_m(int);
    void add_d(int);
    // etc.

private:
    int yyyy_, mm_, dd_;
};

#endif
```

```
// Date.cpp

#include <iostream>
#include "Date.h"

void Date::init(int yyyy, int mm, int dd)
{
    yyyy_ = yyyy;
    mm_ = mm;
    dd_ = dd;
}


void Date::print()
{
    std::cout << yyyy_ << '/'
              << mm_ << '/'
              << dd_ << '\n';
}


void Date::add_y(int inc)
{
    yyyy_ += inc ;
}

// etc.
```

```
// main.cpp

#include <iostream>
#include "Date.h"

int main()
{
    Date today, yesterday;

    today.init(2014, 12, 25);
    today.print();

    yesterday.init(2014, 12, 24);
    yesterday.print();

    return 0;
}
```

Note that there's no change in `main.cpp`.

The `struct` is now called a **`class`**.

Note the `public` and `private` sections in the `Date` class. Try the following code:

```
// main.cpp

#include "Date.h"

int main()
{
    Date today, yesterday;

    today.init(2014, 12, 25);
    today.dd_ = 1;

    ...

    return 0;
}
```

Read the error message from your C++ compiler. Get it?

Basically: Anything under `public` is accessible outside the `Date` class. Anything under `private` is not. (We'll see exceptions later). Inside the `Date` class, everything is freely available. For instance, since `Date::init()` is part of the `Date` class, of course the function body of `Date::init()` can access `yyyy_`. But `main()` is **outside** the `Date` class. So `main()` **cannot** touch `today.yyyy_` directly.

The practice of putting `yyyy_`, `mm_`, `dd_` under the private section to

disallow access from outside the class is called data or information hiding. This is a form of encapsulation.

You can also put member functions into the private section as well. And you can have as many private and public sections as you wish.

Exercise. Create a dummy private member function:

```
// Date.h

#ifndef DATE_H
#define DATE_H

class Date
{
public:
    void init(int, int, int);
    void print();
    void add_y(int);
    void add_m(int);
    void add_d(int);
    // etc.

private:
    int yyyy_, mm_, dd_;
    void f();
};

#endif
```

(You can make it do whatever you want.) Now call `today.f()` in your `main()`.

Now for some very important techno jargon …

The above `Date` is a **class**. You can think of a class as a rubber stamp for creating a certain kind of values. A value created from a class includes values such as member variables. But it also contains member functions. Such a value is called an **object**..

Sometimes we will call the name of the object as object as well. For instance when I do this:

```
Date today;
```

You can think of `today` as the name of a chunk of memory in your computer's RAM. I will usually call `today` an object. However (depending on who you talk to), sometimes, `today` is called an **instance** of the `Date` class. Make sure you think about the difference between the name `today` and the chunk of memory in your RAM that `today` refers to. I will however use object and instance interchangeably.

In C++, the variables inside the class are called **member variables**. For instance `yyyy_` is a member variable of `today`. I will also say that `yyyy_` is a member variable of Date. Outside of C++, when you talk about object-oriented programming in general, the general term is **instance variable**. So when you talk to someone using an object-oriented language such as Python or Java, do not use the term member variable – use instance variable. That's the safest thing to do. In other words, instance variable is a general OOP concept while member variable is a C++ term.

In C++, objects also have functions – we call them **member functions**. In general OOP, you would call them **methods**.

So, in general, an object contains instance variables and methods. Get it?
The point of a class is to (1) bundle data and methods together into objects and also to (2) provide some access control mechanism so that outsiders of the class cannot access certain parts of the class (or certain parts of the objects). The act of doing (1) and (2) is called **encapsulation**.

Information hiding refers to hiding the internal representation of the objects from outsiders of the class.

The above is a class. It's split into the header file and the cpp file. The header file specifies the interface of the class, i.e., the function prototypes in the public section(s) tell the outside world what an object can do and what they need to do anything. The cpp file implements the behavior promised in the header file. As long as the resulting object fulfills the promises made by the interface in the header file, nobody using your code cares how you implement the interface.

Source code that uses a class is called a **client** of the class. For instance, our `main.cpp` is a client of the `Date` class.
By default, every member of a `class` in C++ is `private`.

In C++, a `struct` is just a `class` where every member (member variable or member function) is `public`, i.e.,

```
struct X
{
    …
};
```

is just

```
class X
{
public:
    …
};
```

Scope

It's important to note that the `Date::init` refer to the `init` function inside the `Date` class, i.e., `init` is inside the scope called `Date`. That's right: classes create scopes just like the block of your for-loop forms a scope. The only difference is that in the case of the class, the scope has a name: its name is the name of the class.

It's perfectly OK to have another `init` function OUTSIDE the class like this:

```
// Date.h

...

class Date
{
public:
    void init(int, int, int);

    ...

private:
    int yyyy_, mm_, dd_;
};

void init(int, int, int);

#endif
```

```
// Date.cpp

...

void Date::init(int yyyy, int mm, int dd)
{
    yyyy_ = yyyy;
    mm_ = mm;
    dd_ = dd;
}


void init(int x, int y, int z)
{
    std::cout << "init outside Date\n";
}

...
```

```
// main.cpp

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

```
int main()
{
    Date today, yesterday;

    today.init(2014, 12, 25);
    init(0, 0, 0);

    ...
}
```

In C++, functions outside a class are sometimes called
nonmember functions.

Exercise. Does this compile?

```
// Date.h

...

class Date
{
public:
    ...

    void f();
    void g();

    ...
};

#endif
```

```
// Date.cpp

...

void Date::f()
{
    g();
}

void Date::g()
{
}

...
```

```
// main.cpp

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

```
int main()
{
    Date today, yesterday;

    today.init(2014, 12, 25);
    init(0, 0, 0);

    ...
}
```

Class and Type

In C++, a class is a type (in the sense of C++!!!). This means that, two classes are different types even if their internals are the same:

```
class X
{
    int i;
};

class Y
{
    int i;
};
```

This means that the following will actually compile:

```
#include <iostream>

class X
{
};

class Y
{
};

void f(const X & obj)
{
    std::cout << "f(X)" << std::endl;
}

void f(const Y & obj)
{
    std::cout << "f(Y)" << std::endl;
}

int main()
{
    X a;
    f(a);
    Y b;
    f(b);
}
```

Why? Because in C++, a function is distinguished by the name of the function and its prototype, i.e., C++ supports function overloading.

Passing and returning objects

It's OK to have object parameters and it's also OK to return objects.

Exercise. Now include an `equals` method that accepts a `Date` object and then returns `true` if the `Date` object has the same values as the object invoking the method.

```
// Date.h
...

class Date
{
public:
    ...
    bool equals(const Date &);
    ...
private:
    int yyyy_, mm_, dd_;
};

#endif
```

```
// Date.cpp
...

bool Date::equals(const Date & date)
{

}

...
```

```
// main.cpp
#include <iostream>
#include "Date.h"

int main()
{
    Date today, yesterday;
    today.init(2014, 12, 25);
    today2.init(2014, 12, 25);
    std::cout << today.equals(today2) << '\n';

    ...
}
```

Exercise. Now include an `add` method that accepts a `Date` object and then returns a `Date` object with values obtained by adding the values of the object invoking the call with the object passed in.

```
// Date.h
...

class Date
{
public:
    ...
    Date add(const Date &);
    ...
private:
    int yyyy_, mm_, dd_;
};

#endif
```

```
// Date.cpp
...

Date Date::add(const Date & date)
{

}

...
```

```
// main.cpp
#include <iostream>
#include "Date.h"

int main()
{
    Date today, yesterday;
    today.init(2014, 12, 25);
    Date one;
    one.init(1, 1, 1);
    Date x = today.add(one);
    std::cout << x << '\n';

    ...
}
```

Make sure that the date returned is valid!!!

Exercise. Write a method `neq` (i.e., not equals) for `Date` that does the obvious thing.

Exercise. Write a method `lt` (i.e., less than) for `Date` that does the obvious thing. Then write `le` (i.e., less than or equals), then `gt` (greater than), and then `ge` (greater than or equals).

Note that for the above methods, you should write them in such a way that they emulate expressions that you're used to using. For example, `x <= 2` would do something like this: `x.le(2)`. So the object passed into the `le()` (or other) function should be the value on the right of the expression, and the object calling the method is on the left. This will be discussed more later.

With the above you can now do this:

```
Date start;
start.init(1, 1, 1970);
Date end;
end.init(1, 1, 2015);
Date oneday;
oneday.init(0, 0, 1);

for (Date d = start; d <= end; d = d.add(oneday))
{
    d.print();
}
```

Default `operator=`

The assignment operator = copies values of members from one object to another. So, if `date1` and `date2` are `Date` object, then

```
date1 = date2;
```

will have the same effect as

```
date1.mm_ = date2.mm_;
date1.dd_ = date2.dd_;
date1.yyyy_ = date2.yyyy_;
```

This is just like assignment between `struct` variables. So no big surprises here.

You can re-define `operator=`. There are cases where you should. We'll see this later.

Array of Objects

Exercise. Be brave … Can you create an array of objects? Declare an array of `Date` objects. Set them to whatever dates you choose and print all of them.

Pointers to Objects

Make sure you review pointers … again!!!

You can of course create pointers to objects:

```
Date * p = new Date();
p->add_y(1); // i.e. (*p).add_y(1)
...
delete p;
```

Note the `->` operator. This is just like for the case of `struct`. In other words, if `p` is a pointer and `m` is a member, then `(*p).m` is the same as `p->m`. Likewise to call the `init()` of the `Date` `p` is pointing to, you do:

```
p->init(1, 1, 2003).
```

Exercise. Create a `Date` pointer `tomorrow`. Allocate memory for `tomorrow`. Set the `Date` object that `tomorrow` points to so that the `Date` object is 2014/12/26. Print the object that points to.

Exercise. Convert the `Robot` `struct` from earlier to a `class`. Create a pointer `c3p0` to a `Robot` object. Print its location, move right by 3 units then print its location again. Deallocate memory for `c3p0`.

Exercises.

- Write a `Time` class. The member variables (instance variables) are `hour`, `min`, `sec` (`int` of course!) using the 24-hour format, i.e. hour is 0..23, min is 0..59, sec is 0..59. The member functions (methods) are
- `void init(int, int, int)`
- `void print()`
- `int get_hour()`
- `int get_min()`
- `int get_sec()`
- `void set_hour(int)`
- `void set_min(int)`
- `void set_sec(int)`
- `void add_hour(int)`
- void add_min(int)
- `void add_sec(int)`
- The `add_hour`, `add_min`, `add_sec` must ensure that the resulting `Time` is valid. Of course you should test your `Time` class!
- Create a pointer `p` to a `Time` object, allocate memory for `p`, and call all the available methods through `p`. Deallocate memory at the end.
- Create an array of 1000 pointers to `Time` objects, allocate memory for all of them, set the first to time 00:00:00, the second

to 00:00:01, etc. and print their values. Deallocate memory for every pointer.

- Declare a `Time` pointer `q` and allocate 1000 `Time` objects for `q` and do as above. Deallocate memory for `q`.

Exercise. Does this program compile? (Private section before public)

```
#include <iostream>

class X
{
private:
    int x;
public:
    void n();
};

void X::n()
{
}

int main()
{
    return 0;
}
```

Exercise. Does this compile? (Everything private)

```
#include <iostream>

class X
{
private:
    int x;
    void n();
};

void X::n()
{
}

int main()
{
    return 0;
}
```

Exercise. Does this compile? (Everything public)

```
#include <iostream>

class X
{
public:
```

```
    int x;
    void n();
};

void X::n()
{
}

int main()
{
    return 0;
}
```

Exercise. Does this compile? (Missing prototype)

```
#include <iostream>

class X
{
private:
    int x;
};

void X::n()
{
}

int main()
{
    return 0;
}
```

Exercise. Does this program compile? (Several public)

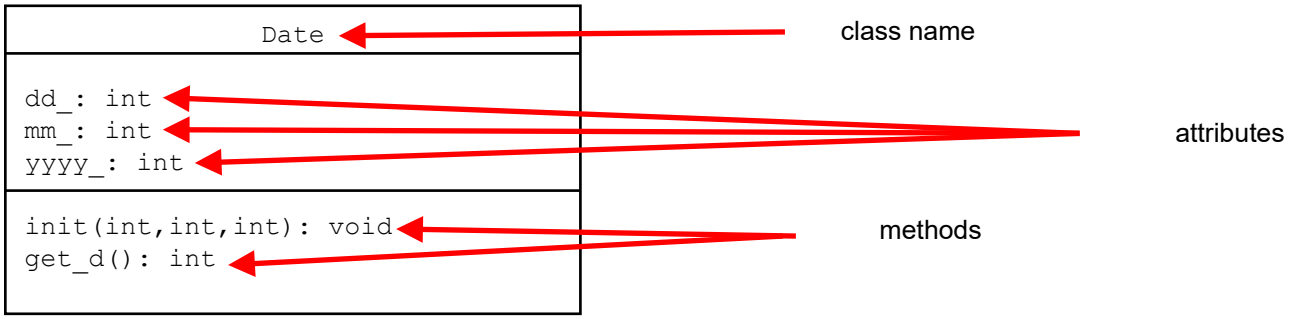
```
#include <iostream>

class X
{
public:
    int x;
private:
    int y;
public:
    int z;
};


int main()
{
    return 0;
}
```

UML

UML (Unified modeling language) is a standard language for modeling objects. It is an industrial standard for communication and it is used for designing systems.

UP (Unified Process) is a very popular process in software development. UP is tightly related to **RUP (Rational Unified Process)** where UML comes from, and RUP is used by >50% of the Fortune 500 companies

The following is a simplified version of the **class diagram** for Date:



Here's a slightly more decorated version: