Simplest version of DayOfYear

```
class DayOfYear {
public:
  void output();
  int month;
  int day;
};
void DayOfYear::output() {
  cout << "month = " << month
       <<", day = " << day << endl;
}
```

What's wrong with DayOfYear?

- Most important: data are exposed to users
- Why is that a problem?
- Two major reasons:
  - No way to insure consistent object states – e.g. user could `birthday.month = 74;` // huh?
  - Developer can't change data names/meanings – e.g. can't change int to string for month, can't save Date instead of month, day, ...
- What's the solution (in C++)?

An access specifier: private

- Private members of a class can only be referenced within the definitions of member functions (and friends – later)
  - If the program tries to access a private member, the compiler gives an error message
- Private members can be data or functions
  - Should have public set methods to change data
  - Need public get methods to access the data
- Btw: default for class is private (public for struct)

Better class DayOfYear

```
class DayOfYear {
public:
  void input();
  void output();
  void set(int new_month, int new_day);
  int get_month();
  int get_day();
private:
  void check_date();
  int month;
  int day;
};
```

Creating and assigning (=) objects

- Declaring an object creates the object
  DayOfYear today, tomorrow;
  // two objects are created on stack
- Different if declaring pointers (or references)
  DayOfYear *soon, &r = today; // no object
  soon = new DayOfYear; // now object on heap
- Assignment operator copies object's data
  r = *soon; // no new object–just copy on stack
  // original (today) object data overwritten

Another class example: BankAccount

- Has operations appropriate for a bank account
  (implemented with public member functions)
  - Add a private utility function
  - Stores an account balance and an interest rate
Method overloading –
BankAccount::set

- A method’s signature includes its name and its parameter list
- Can overload a name like set with a different parameter list
  - Number, types, order

More implementing BankAccount

Sample BankAccount results

Constructors

- A constructor (a.k.a. ctor) is a member function
  - Usually declared public
- One is always called when an object is created
- Main purpose – initialize instance variables
  - Also useful to allocate resources if needed
- Constructor’s name must be the name of the class
- A constructor cannot return a value
  - No return type, not even void

A BankAccount constructor

- Declare in public part of class definition
  BankAccount(int dollars, int cents, double rate);
- Implement essentially like other methods
  BankAccount::BankAccount(int dollars, int cents, double rate) {
    if ((dollars < 0) || (cents < 0) || (rate < 0)) {
      cout << "Illegal values for money or rate!
      exit(1);
    }
    balance = dollars + 0.01 * cents;
    interest_rate = rate;
  }

Constructor call is automatic

- May not invoke (i.e., call) it directly:
  account1.BankAccount(10, 50, 2); // ERROR
- Instead invoke indirectly
  - On stack: BankAccount account1(10, 50, 2);
  - Or free store: ... new BankAccount(10, 50, 2);
- But class must have a matching constructor
  - e.g., BankAccount() if just new BankAccount;
  - Default constructor is called – but oops: ERROR if explicit constructor is defined and not overloaded!
Overloading and the default ctor
- Another possible BankAccount ctor:
  BankAccount (double balance, double interest_rate);
- Or can have either one of the following. Why not both?
  BankAccount (double balance);
  BankAccount (double interest_rate);
- Also either explicitly define default ctor:
  BankAccount ( );
  Or implicitly via default arguments in other ctors:
  BankAccount (double balance = 0.0);
- Tip: good idea to always include a default ctor even if
  there is no need to initialize variables
  - So clients can: BankAccount checking, savings;
  - Important for inheritance reasons too (a future topic)

Base/member initialization list
- An initialization section in a constructor definition
  provides an alternative way to initialize member variables
  BankAccount::BankAccount() : balance(0), interest_rate(0)
  { } // still need a body (even if intentionally empty like this case)
  - Can use parameter names too – even if same name as member!
- Note: order of initialization matches the order in which the
  variables are declared in the class, not their order in the list
- Must use such a list for constants and reference variables
  (since references are always constant)
  - Also must use to initialize private data in a base class (later topic)
- Should always use for user-defined types if default ctor not
  appropriate – to avoid extra ctor (and destructor) calls

Back to the OS – processes
- A process is an executable, machine language program
  that the OS (Linux) has been asked to run
  - Copied to memory, and assigned a process ID (PID)
  - Scheduled for execution by the CPU
- Processes create other processes via system calls
  - A program (e.g., in C or C++) creates a new process
    and terminates itself with a call to exec
  - A program creates a child process by calling fork
    - e.g.: ./.myscript
      - First line is: #!/bin/bash
      - bash runs (interprets script)

Steps to execute a program (sort)
- Step 1: Shell uses fork to create a child
- Step 2: Child uses exec to overwrite itself with the executable file corresponding to
  the next command.
- Step 3: Parent waits for the command to finish. When said
  finishes, the child process terminates and 'bash' starts a new
  prompt, waiting for the user to give another command to execute.

Steps to execute a shell script
- Step 1: bash runs (interprets script)
- Step 2: Parent
- Step 3: Child
Process hierarchy

- init – is PID 1, but all other processes have parents (so PPID)
  - The process hierarchy’s depth is limited only by available virtual memory
- A process may control the execution of any of its descendants
  - Can suspend or resume it
  - Can alter its relative priority
  - Can even terminate it completely
- By default, terminating a process will terminate all of its descendants too
  - So terminating the root process will terminate the session

Example Linux process hierarchy

Linux process states

- Just one process can be "running" at any one time
- OS has other processes in various states
- A process may be cycled through many states before it terminates

Meanings of Linux process states

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>The process is ready to run, but doesn't have the CPU, so the scheduler selects another process (the CPU) to run.</td>
</tr>
<tr>
<td>Running</td>
<td>The process is actively running (using the CPU).</td>
</tr>
<tr>
<td>Waiting</td>
<td>The process is waiting for an event. Possible events are an I/O (e.g., disk) or a timeout.</td>
</tr>
<tr>
<td>Zombie</td>
<td>When the parent of a process terminates before it executes the fork call, it becomes a zombie. The zombie process is terminated and deals with the parent's resources.</td>
</tr>
</tbody>
</table>

Foreground and background

- When a command is executed from the prompt and runs to completion at which time the prompt returns, it is said to run in the foreground
- When a command is executed from the prompt followed by the token ‘&’ on the command line, the prompt immediately returns while the command is said to run in the background
- Programs that interact with a user should be run in the foreground
- Programs that execute slowly and without intervention belong in the background – so other work can get done!
  - e.g., daemons (background processes for system administration)

More states

From: Bulletproof Unix by Tim Gottleber, 2003
User control of process state

- **Terminate a foreground process with ctrl-C**
- **Send running foreground process to background by ctrl-Z**

```bash
-bash-4.2$ find / *.txt > /dev/null 2> /dev/null
```

- Entered ctrl-Z here

```bash
-bash-4.2$
```

Can execute more commands while find works

```bash
-fg
```

Now, job 1 will execute in foreground again

- **Use ps to find PIDs of running processes**

```bash
-bash-4.2$ ps
```

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>20637</td>
<td>pts/4</td>
<td>00:00:00</td>
<td>bash</td>
</tr>
<tr>
<td>21581</td>
<td>pts/4</td>
<td>00:00:02</td>
<td>find</td>
</tr>
<tr>
<td>21632</td>
<td>pts/4</td>
<td>00:00:00</td>
<td>ps</td>
</tr>
</tbody>
</table>

- **Terminate a background process with kill command**

```bash
-bash-4.2$ kill -9 21581
```

-9 is the "sure kill" signal number

```bash
```

Fields of `ps -l` output (cont. next slide)

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Flags: Flags associated with the process. It indicates things like whether the process is a user or kernel process, and why the process stopped or went to sleep.</td>
</tr>
<tr>
<td>UID</td>
<td>User ID: Process owner's user ID.</td>
</tr>
<tr>
<td>PID</td>
<td>Process ID: Process ID of the process.</td>
</tr>
<tr>
<td>PPID</td>
<td>Parent PID: PID of the parent process.</td>
</tr>
<tr>
<td>PRI</td>
<td>Priority: Priority number of a process that dictates when the process is scheduled.</td>
</tr>
<tr>
<td>NI</td>
<td>Nice value: The nice value of a process; another parameter used in the computation of a process's priority number.</td>
</tr>
<tr>
<td>VSZ</td>
<td>Virtual size: The number in this field is the size of the memory image of a process (code+data+stack) in blocks.</td>
</tr>
</tbody>
</table>

Fields of `ps -l` output (cont.)

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSS</td>
<td>Resident set size: The amount of physical memory in kilobytes; it does not include space taken by the page table and kernel task structure for the process.</td>
</tr>
<tr>
<td>WCHAN</td>
<td>Wait channel: Null for running processes, or processes that are ready to run and are waiting for the CPU to be given to them. For a waiting or sleeping process, this field lists the event the process is waiting for—the kernel function where the process resides.</td>
</tr>
<tr>
<td>STAT</td>
<td>Process state: See <code>state</code> field.</td>
</tr>
<tr>
<td>TTY</td>
<td>Terminal: The terminal name a process is attached to.</td>
</tr>
<tr>
<td>TIME</td>
<td>Time: The time (in minutes and seconds) a process has been running, or previously ran for before sleeping or stepping.</td>
</tr>
<tr>
<td>COMMAND</td>
<td>Command: Lists the command line that was used to start this process. The <code>-l</code> option is needed to see the full command line; otherwise only the last component of the pathname is displayed.</td>
</tr>
</tbody>
</table>

Process state abbreviations

<table>
<thead>
<tr>
<th>Process State</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Uninterruptible sleep (usually doing I/O or waiting for I/O)</td>
</tr>
<tr>
<td>W</td>
<td>Low-priority process (a process that has been moved)</td>
</tr>
<tr>
<td>R</td>
<td>Runnable process; waiting to be scheduled to use CPU</td>
</tr>
<tr>
<td>S</td>
<td>Sleeping</td>
</tr>
<tr>
<td>T</td>
<td>Traced or stepped</td>
</tr>
<tr>
<td>Z</td>
<td>A zombie (defunct) process</td>
</tr>
<tr>
<td>W</td>
<td>A process that is completely swapped on the disk (no resident pages)</td>
</tr>
</tbody>
</table>