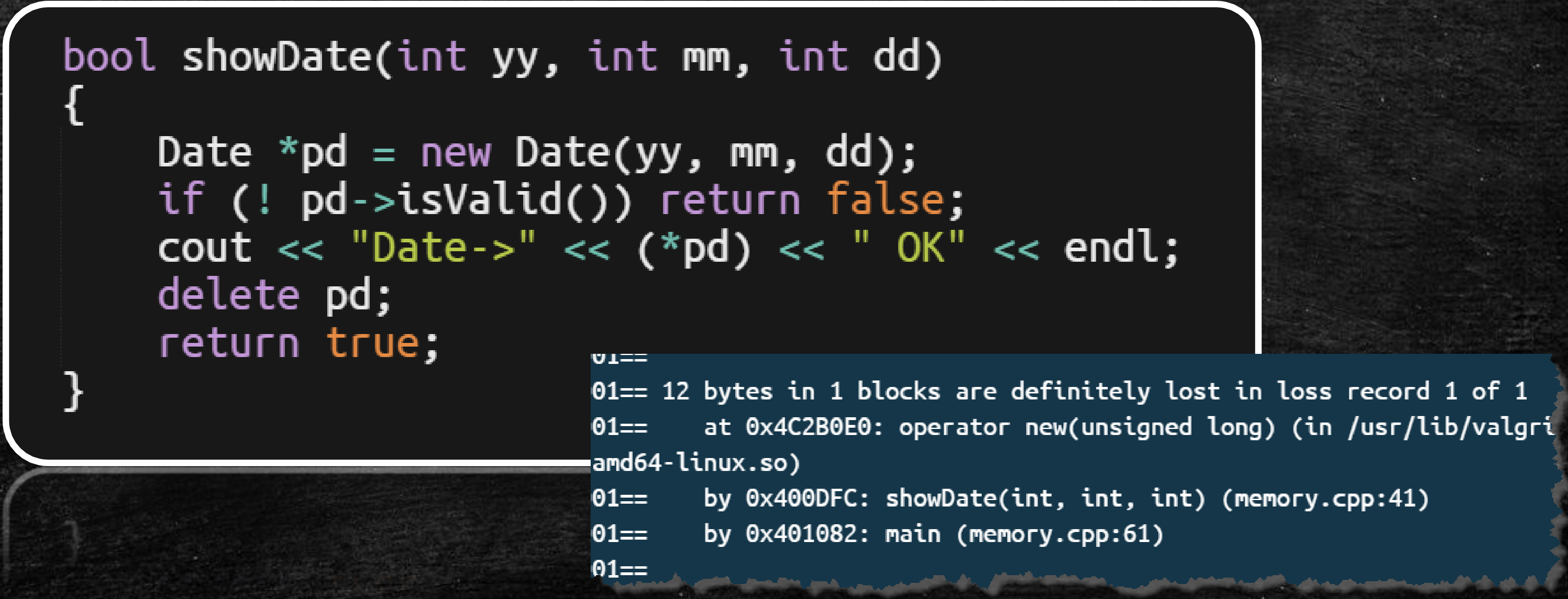
Memory & Information Hiding

CS 150 – C++ Programming I

Lecture 24

Dynamic Memory Pitfall 1: Memory Leaks

- Losing track of a pointer to dynamic memory
  - Solution? match every new with a delete

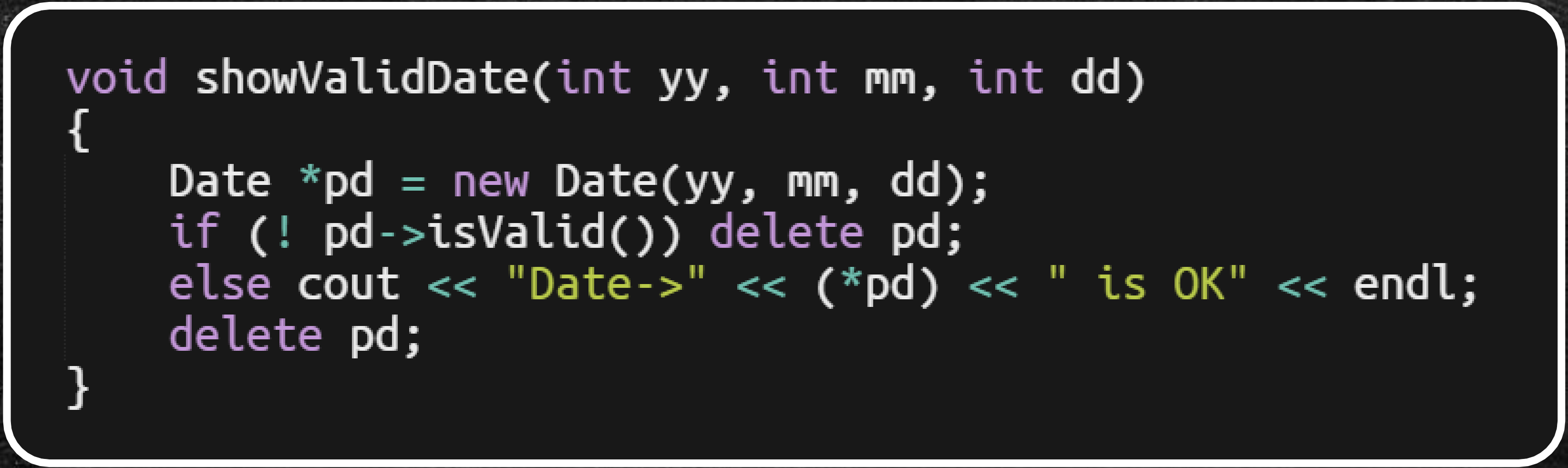


```
bool showDate(int yy, int mm, int dd)
{
    Date *pd = new Date(yy, mm, dd);
    if (! pd->isValid()) return false;
    cout << "Date->" << (*pd) << " OK" << endl;
    delete pd;
    return true;
}
```

```
01==
01== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
01==    at 0x4C2B0E0: operator new(unsigned long) (in /usr/lib/valgri
amd64-linux.so)
01==    by 0x400DFC: showDate(int, int, int) (memory.cpp:41)
01==    by 0x401082: main (memory.cpp:61)
01==
```

Dynamic Memory Pitfall 2: Double Delete

- Deleting an already deleted pointer can corrupt the memory manager information tables

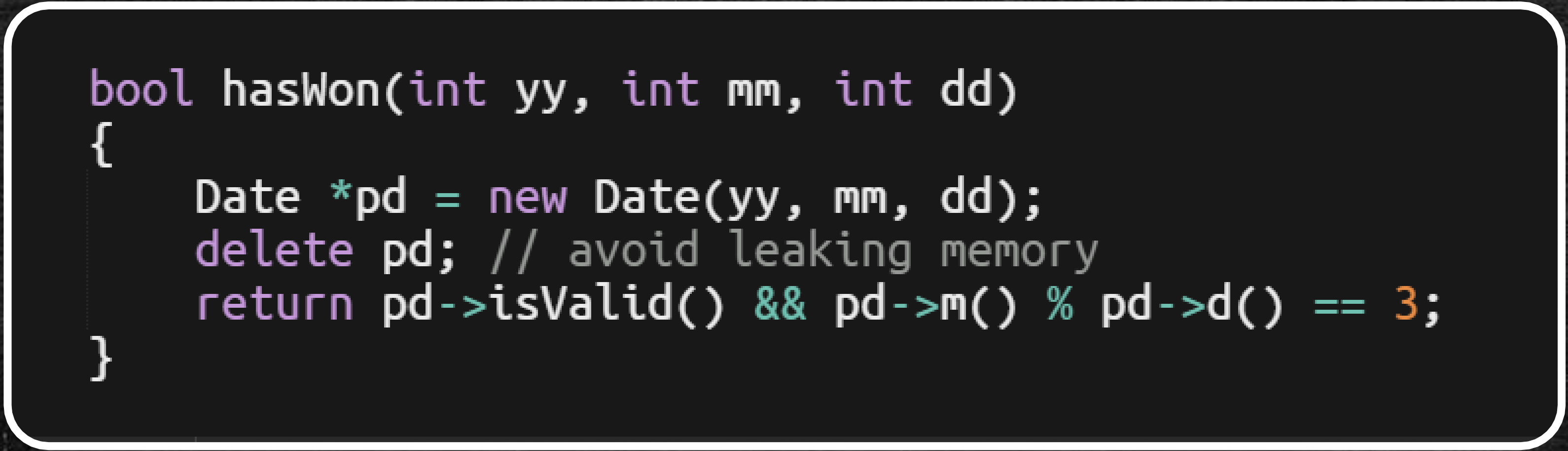


```
void showValidDate(int yy, int mm, int dd)
{
    Date *pd = new Date(yy, mm, dd);
    if (! pd->isValid()) delete pd;
    else cout << "Date->" << (*pd) << " is OK" << endl;
    delete pd;
}
```

- Can mitigate this by setting deleted pointer to `0` (`nullptr`)
  - But this may really only hides the problem (a coding error)

Pitfall 3: Dangling Pointers

- Using a pointer that has already been deleted



```
bool hasWon(int yy, int mm, int dd)
{
    Date *pd = new Date(yy, mm, dd);
    delete pd; // avoid leaking memory
    return pd->isValid() && pd->m() % pd->d() == 3;
}
```

- In this case, setting deleted pointer to nullptr illuminates

  -
    ```
    template <typename T>
    void deleteRawPtr(T* p) {
      delete p; p = nullptr;
    }
    ```

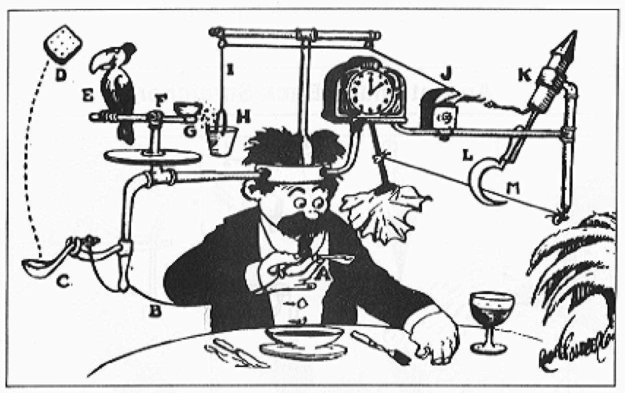
Your Turn

- Open `structPtr.cpp`
  - Write a function that initializes an `Employee`
  - Write a `doubleSalary` function that doubles an `Employee's` salary
  - Create two `Employee` variables, one on the heap and a second as a local (stack) variable
  - Initialize both variables (use different values)
  - Print the info on both
  - Don't forget to free the dynamic variable

Your Turn

- Open *`dynArray.cpp`*
  - Ask user how large an array to create
  - Create dynamic int array of that size
  - Ask user to fill in values (loop)
  - Sum and print the values
  - Do `make grind` to see the output

Data Abstraction

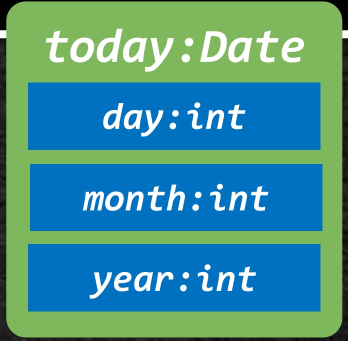


- A data type is a concrete model of a concept
  - E.g. double models reals, int models integers
- Unlike some languages, C++ is extensible
  - You may create your own new data types
  - Enumerated scalar types (like Weekday)
  - Structured or record types (like a Date)
  - Classes (first class structured types; act like the built-ins)
  -
    ```
    int n = 3;                 // built-in type
    string str = "Fred";       // class type
    ```

Monday
January
3
2009

A Date Structure

```
struct Date
{
    int day;
    int month;
    int year;
};
```



- Here is a Date defined as a structure
  - Create variables: Date today;
  - Initialize: Date bday = {2, 2, 1950};
  - Assign: today = bday;
  - Member Access: today.year = 2022;
- You can pass Date variables to functions

```
string toString(const Date& d);
int daysBetween(const Date& d1, const Date& d2);
void addYear(Date& d);
```

Date Invariants

- Some dates are valid, and others aren't (31-4-2022)
  - But, we don't want to check each function for an invalid date
  - Better to prohibit them from being created
- To create only valid dates use a factory function
  - When is the error found? `auto d = makeDate(1, 13, 2020);`

```
Date makeDate(int day, int month, int year)
{
    if (month < 1 || month > 12) throw . . .
    // Check day and year as well
    return Date{day, month, year};
}
```

Date & Strong Parameter Types

- What day is this? `auto d = makeDate(7, 4, 1776);`
  - Did you mean independence day or April 7th?
  - Month/day confusion is not caught by the compiler
- To fix, provide a strong parameter type for month

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

Date makeDate(int day, Month month, int year)
{
    // Check day and year as well
    return Date{day, month, year};
}
```

```
struct Date
{
    int day;
    Month month;
    int year;
};
```

Date Invariants II

- A factory (initializer) function avoids creating bad dates
  - What about those who use the date? Can you trust a Date passed to a function? No!!!!

```
Date d1 = {2, 2, 1950};
d1.day = 75;
```

```
struct Date {
   long long daysFromZero;
};
```

- Can you change the implementation to make it more efficient?
  - No!!!! With structures, the implementation is the interface
  - Since you can directly access the data members it is inherently unsafe, error prone, and inflexible