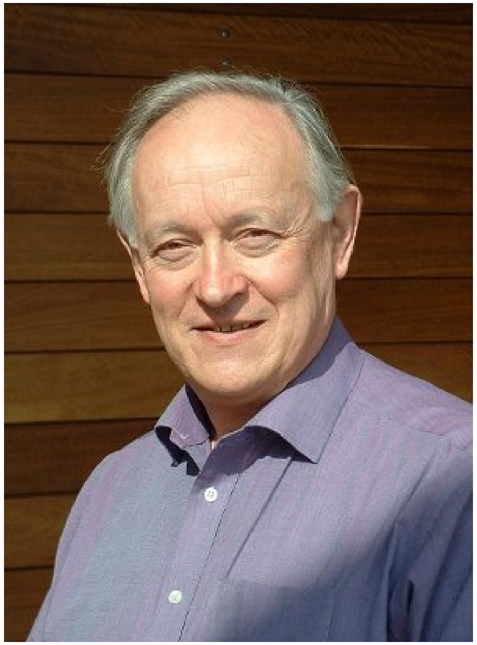
# CS32 Summer 2013

Intro to Object-Oriented Programming in C++

Victor Amelkin

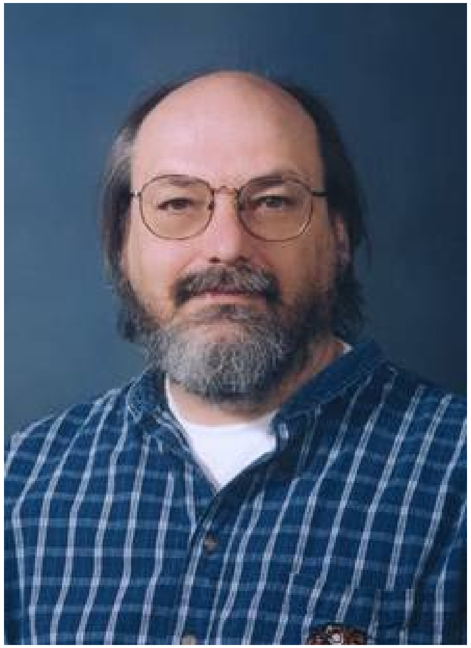
August 12, 2013

# History



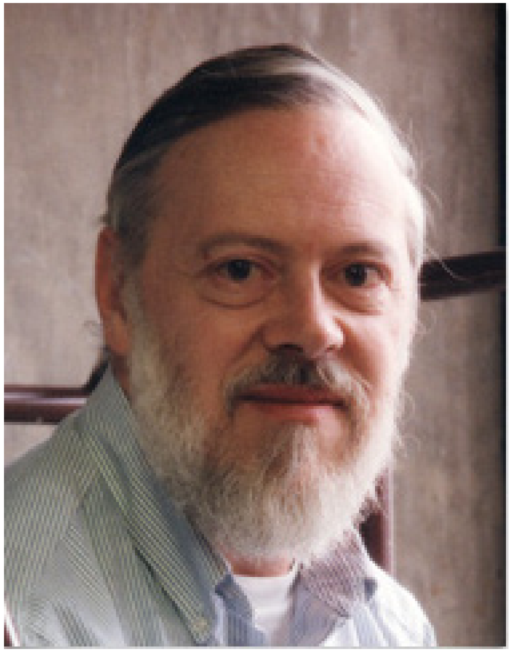
*Martin Richards*

**BCPL**
(1966)



*Ken Thompson*

**B**
(1970)



*Dennis Ritchie*

**C**
(1972-...)

C89
C90
C99
C11

*Bjarne Stroustrup*

**C++**
(1979-...)

C++98
C++03
C++TR1 ('07)
C++11

# Object-Oriented Programming

- Real word consists of objects
  - car, head, spoon, ...
- Objects have states
  - `car { nwheels = 4, current_gear = 2, color = red }`
- Objects act

  -
    ```
    car.start()
    car.drive(destination)
    car.crash_into("nearby tree")
    ```
- We want our programs to reflect the real world

  We want to write our programs in terms of *objects*, their *state* and *behavior*

# Objects in C: State

- Predefined C types (int, double, ...) are not sufficient to represent object states
  - `int car_state` – does not describe a car's state close enough

- Gather multiple variables in a structure

  -
    ```
    struct car_state {
      int n_wheels;
      int n_seats_available;
      double max_speed_mph;
      …
    }
    car_state car1;
    car1.n_wheels = 3;
    ...
    ```

- What about object's behavior?

# Objects in C: Behavior

-
  ```
  struct car_state {
     int n_wheels;
     int n_seats_available;
     double max_speed_mph;
     …
  }
  ```

- In C, object's behavior is “externally defined”:

  ```
  void add_passenger(car_state *c, person *p) {
     …
     c->nseats_available -= 1;
  }
  ```

- No protection: anyone can alter `car_state`'s fields.

# Better Objects

- Restrict access to objects' fields

- Allow only “trusted” functions to alter the state
  - In C, we cannot allow only some functions to access the object's state

- We want objects to incorporate both their *state* and *behavior*

# User-Defined Types in C++: Classes

```
class date {
    private:
        int _day, _month, _year;
    public:
        date(int day, int month, int year) {
            _day = day;
            _month = month;
            _year = year;
        }
        void print() {
            printf("%d-%d-%d\n", _day, _month, _year);
        }
};

int main() {
    date dt(12, 8, 2013);
    dt.print();
    // dt._day = 123; - does not work!
    return 0;
}
```

# User-Defined Types in C++: Classes

- C++ classes describe both
  - state through *fields*
  - and behavior though *methods*

- Class' fields and methods – *class members*

- Object of *class MyClass – instance of MyClass*

- Access control to members (*public*/*private*)

- No need to use `struct` in C++ (but some people do for POD-types)
  - In C++, `struct` ~= `class`
  - `struct`'s members are *public* by default
  - `class`'s members are private by default

# Access Control

- Class members can be *private* or *public*
  - In future, we will add *protected* members

```
class MyClass {
    private:
        int field1;
        float field2;
    public:
        char field3;
    private:
        method1() { field1 = 1; field3 = 'w'; /*OK*/ }
    public:
        method2() { field2 = 1; field3 = 'a'; /*OK*/ }
};

MyClass obj; // obj is an “instance” of class MyClass
obj.field1 = 1; // does not work!
obj.field3 = 'A'; // OK
obj.method1(); // does not work!
obj.method2(); // OK
```

# Object Construction

- *Constructor* – a method that *initializes* the state of an object
- Constructor is *named* as its class
- Class may have *multiple constructors* with different signatures

```
class date {
    private:
        int _day, _month, _year;
    public:
        date();
        date(int day, int month, int year);
        date(const char *datestr);
};

date d1; // using the first ctor
date d2(29, 8, 1985); // using the second ctor
date d3(“29-08-1985”); // using the third ctor
```

# Other Methods

- Constructors initialize the state of an object
- Other methods can change an object's state too

```
class date {
    private:
        int _day, _month, _year;
    public:
        void add_day();
        bool is_end_of_month();
        bool is_end_of_year();
};

void date::add_day() {
    if(is_end_of_month()) {
        day = 1; // or this->day = 1
        if(is_end_of_year()) {
            _month = 1;
            _year++;
        } else
            _month++
    } else
        _day++;
}
```

`MyClass *this` – hidden argument internally passed to each (non-static) member

# Creating Objects

- Memory allocation for `class`' objects is similar to C `struct`s:
  - Object creation on the **stack**:

    ```
    date dt1;
    date dt2(1, 12, 2011);
    dt1.print();
    // dt1, dt2 disposed automatically
    ```

  - Object creation in the **heap**:

    ```
    date *dt1 = new date();
    date *dt2 = new date(1, 12, 2011);
    dt1->print();
    delete dt1;
    delete dt2;
    ```

# Re-Creating Objects?

- Never attempt to re-create objects

```
date dt(12, 8, 2013);
dt.~date();
new (&dt) date(1, 2, 3);
dt.print();
```

**- NOT COOL!**

- Constructor is called only once at the moment of creation
- Need to *re-initialize* an object?
  - either use a custom assign/initialize member

```
date dt(12, 8, 2013); // want to change this object
dt.assign(1, 2, 3); // assigns values to the fields
dt.print(); // prints 1-2-3
```

  - or create a new object

```
date dt(12, 8, 2013);

dt = date(1, 2, 3);
```

# Object Destruction

- *Destructor* – a method that is called before an object dies
- Destructor is *named* as its class with ~ prefix
- Class may have *only one* destructor

```
class date {
   private:
       int _day, _month, _year;
   public:
       date(int day, int month, int year); // ctor
       ~date(); // dtor
};

// 1) memory is allocated
// 2) ctor is called
date *pd = new date(29, 8, 1985);

// 3) destructor is called
// 4) memory is released
delete pd;
```

# Interface vs. Implementation

- Definitions of methods are (*usually*) separated from declarations

```
class date {
    private:
        int _day, _month, _year;
    public:
        // Declarations (“interface”)
        date(int day, int month, int year);
        print();
};

// Definitions (“implementation”)

date::date(int day, int month, int year) {
    _day = day;
    _month = month;
    _year = year;
}

void date::print() {
    printf("%d-%d-%d\n", _day, _month, _year);
}
```

# Separate Compilation: Motivation

```
// date.cpp
class date {
    private:
        int _day, _month, _year;
    public:
        date(int day, int month, int year);
        print();
};

date::date(int day, int month, int year) {
    _day = day;
    _month = month;
    _year = year;
}

void date::print() {
    printf("%d-%d-%d\n", _day, _month, _year);
}

// user1.cpp
date dt1(1, 3, 1999);

// user2.cpp
date dt2(12, 8, 2013);
```

# Separate Compilation: Motivation

- In C++, before using something, it should be *declared*
- Bad solution:

```
// user1.cpp

// declaration
class date {
    public:
       date(int day, int month, int year);
       print();
};
// usage
date dt1(1, 3, 1999);

// user2.cpp

// declaration
class date {
    public:
       date(int day, int month, int year);
       print();
};
// usage
date dt2(12, 8, 2013);
```

What will happen to *user1.cpp* and *user2.cpp* if we decide to change the signature of the constructor? (Hint: lots of code rewriting.)

# Separate Compilation

```
// date.h - header file - contains declarations (“interface”)
class date {
    private:
        int _day, _month, _year;
    public:
        date(int day, int month, int year);
        print();
};

// date.cpp - implementation file - contains definitions
#include “date.h”
date::date(int day, int month, int year) { … }
date::print() { … }

// user.cpp
#include “date.h”
date dt1(1, 3, 1999);

// user2.cpp
#include “date.h”
date dt2(12, 8, 2013);
```

# Header Files

- Header files (“headers”) are named {name}.h
- Headers contain declarations of classes, functions, global vars
- Header may contain declarations for multiple classes
- Member *implemented* inside a header gets inlined (“one definition rule”)
- Use *#include guards* to prevent double inclusion of a header

```
// my_header.h
#ifndef __MY_HEADER_H__
#define __MY_HEADER_H__

        … header contents (included only once) …

#endif // __MY_HEADER_H

// user1.h
#include “my_header.h”

// user2.h
#include “user1.h”
#include “my_header.h”
```

# Chaining Constructors

*in pre-C++11*

- Class may have multiple constructors
- These constructors may want to share some code

```
car::car(color) {
   _color = color;
   init_engine();
   init_gps();
}

car::car(color, nwheels, owner) {
   _color = color;
   _nwheels = nwheels;
   _owner = owner;
   init_engine();
   init_gps();
}
```

- Can we “call” the first ctor from the second?

# Chaining Constructors

*in pre-C++11*

- Can we "call" the first ctor from the second ctor?

```
car::car(color) {
   _color = color;
   init_engine();
   init_gps();
}

car::car(color, nwheels, owner) {
   call car(color) for the current object
   // _color = color;
   _nwheels = nwheels;
   _owner = owner;
   // init_engine();
   // init_gps();
}
```

- In C++98, we cannot do it directly (in C++11 we can)

# Chaining Constructors

*in pre-C++11*

- Solution: extract an initializing method

```
car::car(color) {
   init(color);
}

car::car(color, nwheels, owner) {
   init(color);
   _nwheels = nwheels;
   _owner = owner;
}

// just a regular method (usually named init or assign)
car::init(color) {
   _color = color;
   init_engine();
   init_gps();
}
```

# Copy Constructor

- Objects are initialized with *constructors*
- *Copy constructor* – special constructor used for creating a *copy* of an existing object; default copy constructors are created automatically

```
class date {
   private:
       int _day, _month, _year;
   public:
       // Default copy ctors defined automatically
       // date(date &other); // copy ctor
       // date(const date &other); // copy ctor
};

// Default semantics of copy ctors - memberwise copy

date dt1;
const date dt2;
date dt3(dt1); // copy ctor is called
date dt4(dt2); // const copy ctor is called
```

# Copy Constructor

- We need an explicitly defined copy ctor to make a *deep copy* (i.e., follow pointers)

```
class myclass {
   private:
       int x;
       char *p;
   public:
    // Default copy ctors will copy pointer p, so
    // that all copies will point to the same string
       myclass(const myclass &other);
};

// creating a deep copy
myclass::myclass(const myclass &other) {
   x = other.x;
   int len = strlen(other.p);
   p = new char[len + 1];
   strcpy(other.p, p, len);
}
```

# Assignment Operator

- Similar to copy ctor (defaults created automatically)

```
class MyClass {
   private:
       int state;
   public:
       // MyClass& operator=(const MyClass &other);
       // MyClass& operator=(MyClass &other);
};

MyClass x;
MyClass y;
x = y; // assignment operator is called
```

- As with copy ctors, default semantics – memberwise copy

# Summary

- *Class* describes *state* and *behavior* of its objects
  - fields
  - methods
- *Access* to members: private / public
- Class' *interface* and *implementation* are usually separated
  - interface (declarations): myclass.h
  - implementation (definitions): myclass.cpp
- Constructors initialize class' objects
- Destructor may release some acquired resources
- Copy constructors and assignment operators are used for copying objects

# Object Life-Cycle Demo

- Want a class with all of the following:
  - Fields
  - Regular methods
  - Constructors
    - default ctor
    - constructors accepting arguments
    - copy ctors
  - Destructor
  - Assignment operators

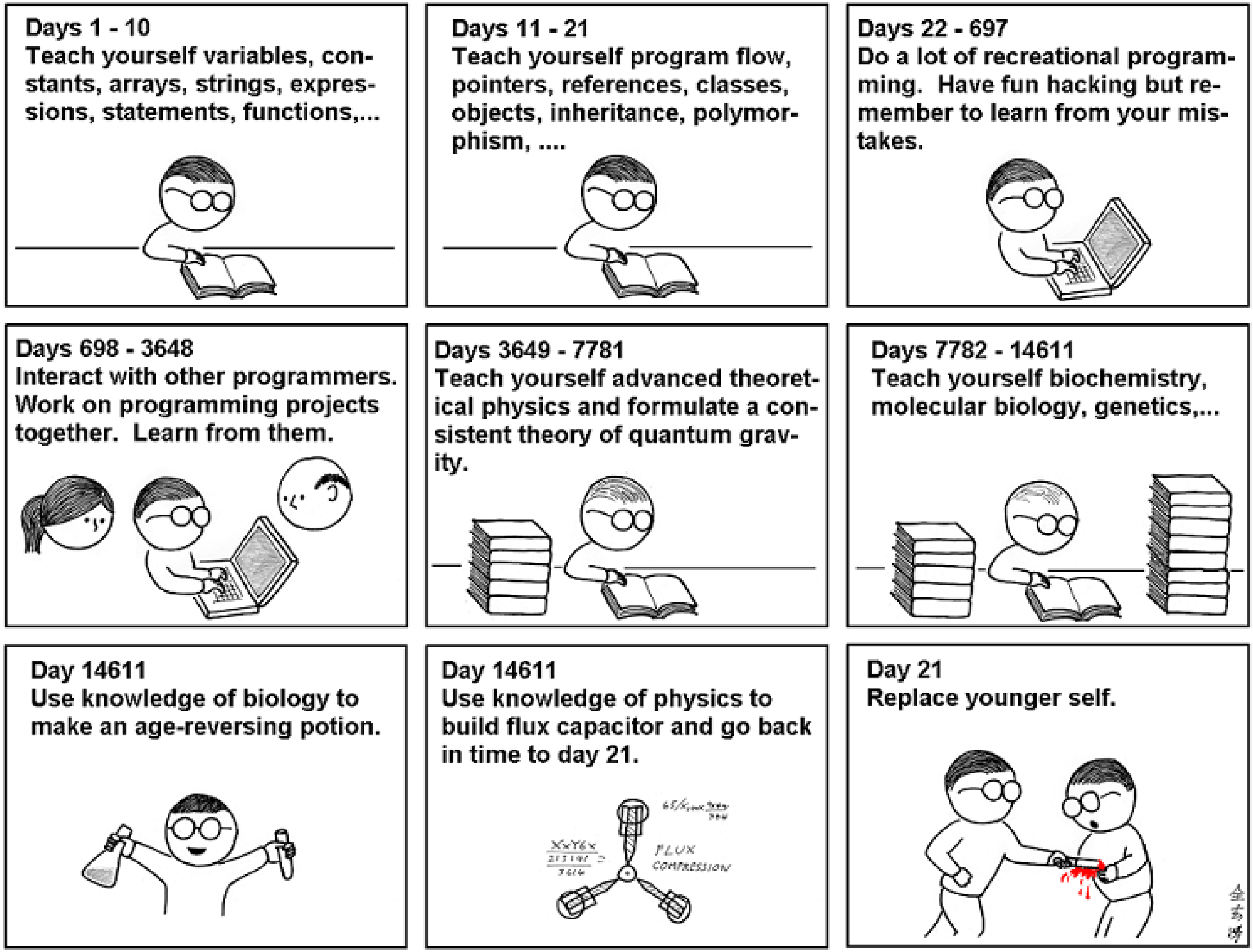
# Object Life-Cycle Demo

```
// xstring.h
class xstring {

private:
    int _length;
    char *_chars;

public:
    xstring();
    xstring(const int length, const char filler);
    xstring(const char *str);
    xstring(const xstring &other);
    ~xstring();

    xstring& operator=(const xstring &other);

    void clear();
    int get_length() const;
    void print() const;

private:
    void init(const char *other);
};
```

# Object Life-Cycle Demo

- http://cs.ucsb.edu/~victor/ta/cs32/lect-aug-12/ex/
- Example index:
  - main1.cpp – default ctor; stack
  - main2.cpp – paramed ctor; stack
  - main3.cpp – paramed ctor; heap
  - main4.cpp – copy ctor; stack
  - main5.cpp – heap; memory leak; valgrind
  - main6.cpp – assignment op; stack
  - main7.cpp – assignment op; heap
  - main8.cpp – unnecessary objects
  - main9.cpp – ultimate wisdom; gdb

# Questions?


Days 1 - 10
Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...
Days 11 - 21
Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism, ....
Days 22 - 697
Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.
Days 698 - 3648
Interact with other programmers. Work on programming projects together. Learn from them.
Days 3649 - 7781
Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.
Days 7782 - 14611
Teach yourself biochemistry, molecular biology, genetics,...
Day 14611
Use knowledge of biology to make an age-reversing potion.
Day 14611
Use knowledge of physics to build flux capacitor and go back in time to day 21.
FLUX COMPRESSION
Day 21
Replace younger self.

As far as I know, this
is the easiest way to
"Teach Yourself C++ in 21 Days".