C++ at Velocity, Part 2

Ben Yuan
CS 002 - WI 2014
January 8, 2014

General remarks

- Course website is up on Moodle
  - Enrollment key: 'CS2EnrollMe'
- Assignment 1 is out
  - 12 points of mostly easy C++
  - 3 points of tic-tac-toe
  - up to 8 points of "other stuff"
  - report bugs to cs2-tas@ugcs.caltech.edu
- Assignments distributed via Git
  - to clone: `git clone https://bitbucket.org/...`
  - to update: `git pull`
  - Try to `git pull` before starting every day

Last time…

- C++ basics
- Arrays
- Pointers!

Coding style guidelines

- We don't really enforce a particular style in this course.
- Only requirements:
  - Clean
  - Readable
  - <u>Consistent</u>
- The point: someone else eventually needs to maintain your code!
  - ...you, after you've forgotten the details
  - ...your coworkers

(C++ 2.25)

Coding style suggestions

- Spacing between operands
  - What's easier to read?

    `int y=a+b;` vs. `int y = a + b;`
- Control statement spacing
  - What's easier to read?

    `if(a==b)` vs. `if (a == b)`
- Block indentation
  - use spaces, not tabs
  - keep indentation consistent (as if Python)
- Variable and function naming
  - try to pick names that convey <u>semantic</u> meaning
    - common exception: loop variables **`i`**, **`j`**, **`k`**

(C++ 2.27-29)

Coding style guidelines

- Commenting: explain, don't repeat
  - Don't tell me `i++` increments `i`; tell me why it's there
- Explain complex or subtle code
- Write function headers (even small ones)
  - Invalid arguments? Return value?

If you woke up one day without your memories then how would you reconstruct your life?

(C++ 2.26)

Coding style suggestions

- Working on someone else's codebase?
- Keep style consistent
  - either adopt the style already in the file,
  - or convert the entire file
    - don't do that unless you absolutely have to
- Code should look like one person wrote it

std::cout

- Part of the C++ Standard Library
- Automatically converts values to text (where overload is defined)
- Syntax is often cleaner
  - no crazy-looking format string
- Some features less easy to use than printf()
  - e.g. advanced formatting, printing as hex, etc.
- Use either in this course
  - only rule: be consistent

```
#include <iostream>

int a = 900;
double b = 3.14159;
const char * s = "string";

// print some values
std::cout << a << " "
          << b << " "
          << s
          << std::endl;
```

(C++ 1.11-14)

Pointers

- When a variable `i` is declared, some memory is reserved for its contents.
- This memory has an **address** `&i`.

```
int i = 10;
printf("i is at %p\n", &i);
```

- This prints something like "i is at 0xff831f2c".
- This number is `i`'s address.

(C 4.6-7)

Pointers

- A pointer is a variable that holds an **address**

```
int i = 10;
int * j = &i; // j 'points' to i
```

name	address	contents
i	**0xff831f2c**	10
j	0xff831f30	**0xff831f2c**

- **&** is the **address-of** operator.

(C 4.7-4.9)

Pointers

- A pointer is a variable that holds an **address**

```
int i = 10;
int * j = &i; // j 'points' to i
printf("j = %p\n", j);
printf("j points to: %d\n", *j);
```

- `*j` is the contents of memory at the address in `j`; `*` operator **dereferences** j
  - "What is j pointing to?"

(C 4.8-9)

The many uses of *

```
c = a * b; // multiplication

int * p1; // pointer declaration

int foo = *p2; // dereferencing
```

(C 4.10)

Pointers and call-by-reference

- Function calls in C++ copy arguments by default.
  - normally can't change a variable we pass to a function
- Passing a memory address instead lets us make changes.
- We also avoid copying large amounts of data
  - imagine having to copy an entire picture each time you want to change it!

```
void incr(int * i)
{
   (*i)++;
}

// ... later ...

int j = 10;
incr(&j);
// j is now 11
```

(C 4.19-23)

C++ references and call-by-reference

- C++ allows us to mark arguments as references.
  - Use them exactly as you would the variable.
  - Changes are passed through.
- Often cleaner syntax!
  - fewer parentheses and * floating around
- Can't do everything pointers can do.

```
void incr(int & i)
{
  i++;
}

// ... later ...

int j = 10;
incr(j);
// j is also now 11!
```

(C++ 1.30-35)

Pointer arithmetic

- We can change where a pointer points
  - Usually by assigning a new memory address
- We can also add to, and subtract from, pointer variables
  - Changing the memory address!

```
int i[50];
int * j = &i[0];
j += 5; // j now points to i[5]
```

(C 5.2-15)

Pointer arithmetic

- Note that we are adding and subtracting multiples of the type size!

```
int i[50]; // sizeof(int) usually 4
// suppose i[0] lives at 0xff000000
int * j = &(i[0]);
j += 5; // j now points to 0xff000014
        // NOT 0xff000005!
        // we added 5 * sizeof(int)
```

(C 5.2-15)

Array-pointer equivalence

- Array notation is syntatic sugar for pointers.

```
int arr[5] = {1, 2, 3, 4, 5};
printf("arr[3] = %d\n", arr[3]);
printf("arr[3] = %d\n", *(arr + 3));
```

- `arr[3]` and `*(arr + 3)` are identical!
- `arr` is identical to `&arr[0]`

- Any array operation can be replaced by a pointer operation.

(C 5.16)

Dynamic memory allocation

- Pointers become useful when we need to allocate arrays on the fly
  - Here's how:

```
int n = 9001;
// ... later ...
int * buf = new int[n];
```

  - This creates a new block of integers of size n

(C++ 2.9, 2.11-12)

Dynamic memory allocation

- Free your memory after using it!
- C++ does not clean up automatically
- Forgetting to free memory == memory leaks

```
double a = new double;
int * buf = new int[90001];
// ... later ...
delete a;      // destroy singleton
delete[] buf; // destroy array
```

(C++ 2.9, 2.11-12)

The `struct`

- Allows us to define compound data types

```
struct point {
  double x;
  double y;
}; // MUST have semicolon

point p; // in C, use 'struct point'
p.x = 3.0; // 'dot notation'
p.y = 4.0;
printf("dist %f\n",
          sqrt(p.x * p.x + p.y * p.y));
```

(C 6.15-17)

The `struct`

- Can include pointers to other structs

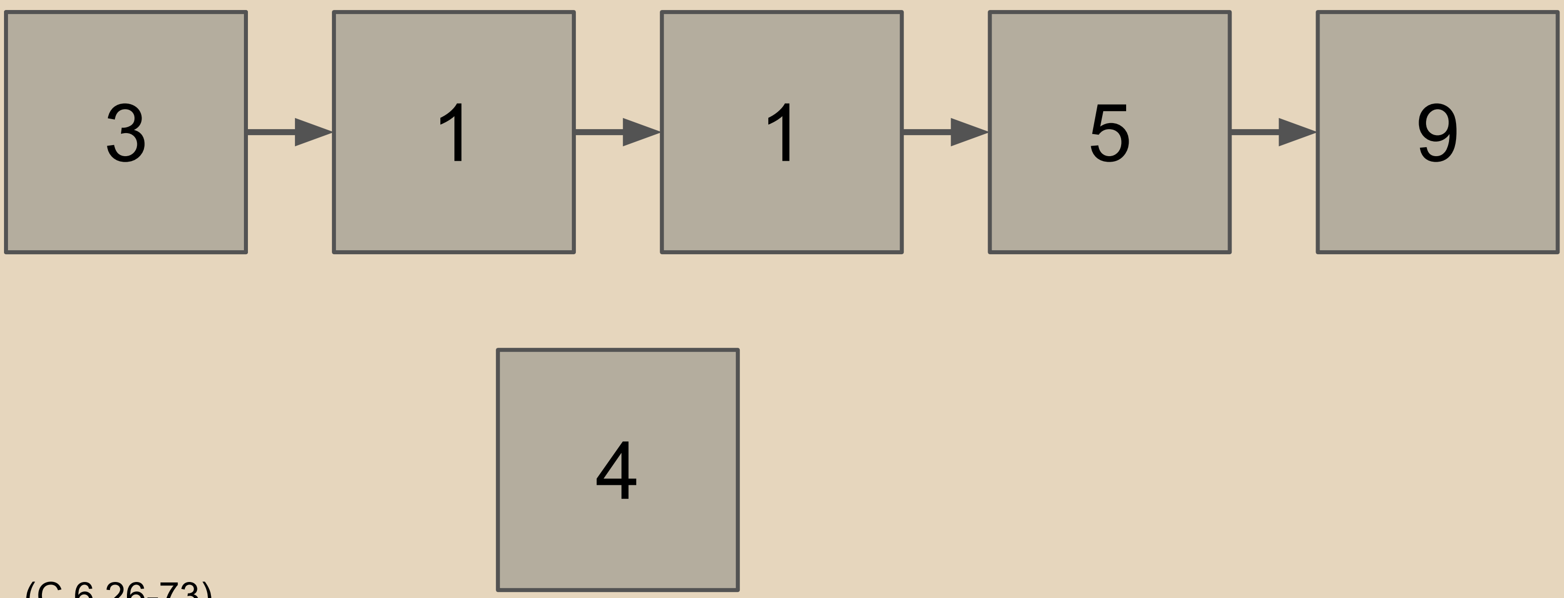
```
struct node {
    int value;
    node * next;
}

void append_to(node * self, node * prev)
{
    prev->next = self; // 'arrow' notation
}
```

(C 6.19-21)

Linked lists

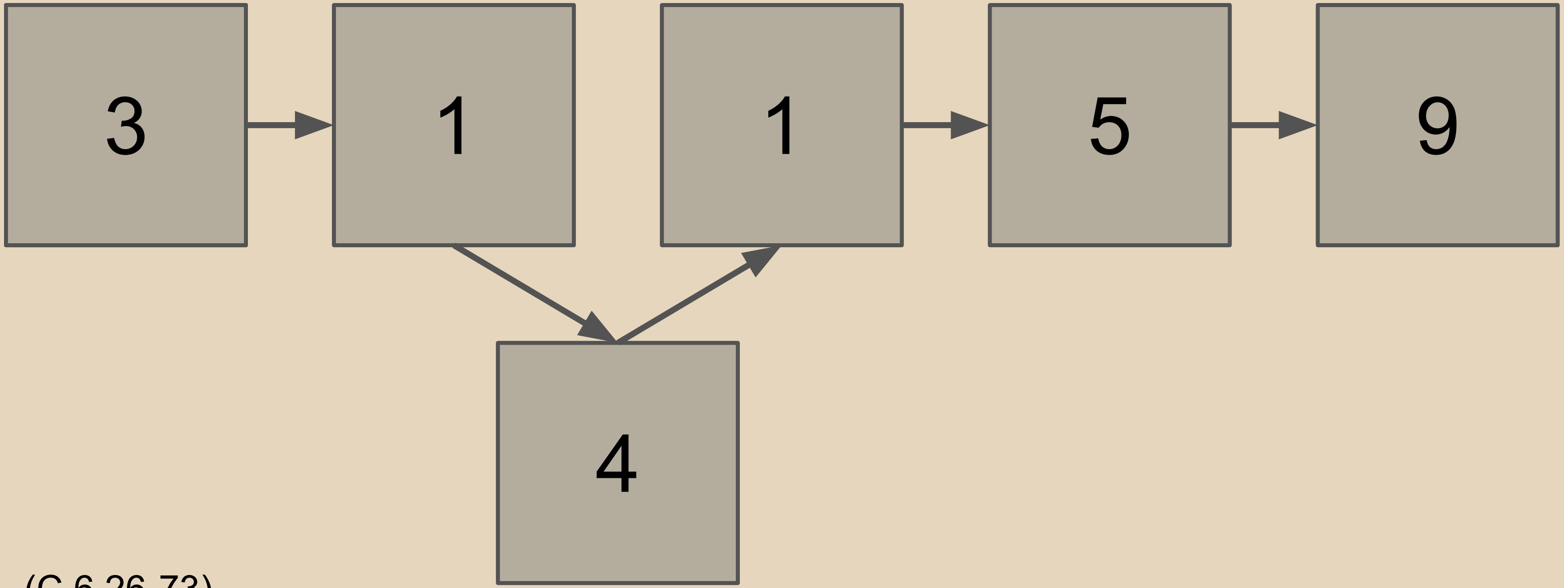
- Simple 'complex' data structure
- Supports iteration, constant-time insertion, constant-time deletion
- No constant-time random access!



(C 6.26-73)

Linked lists

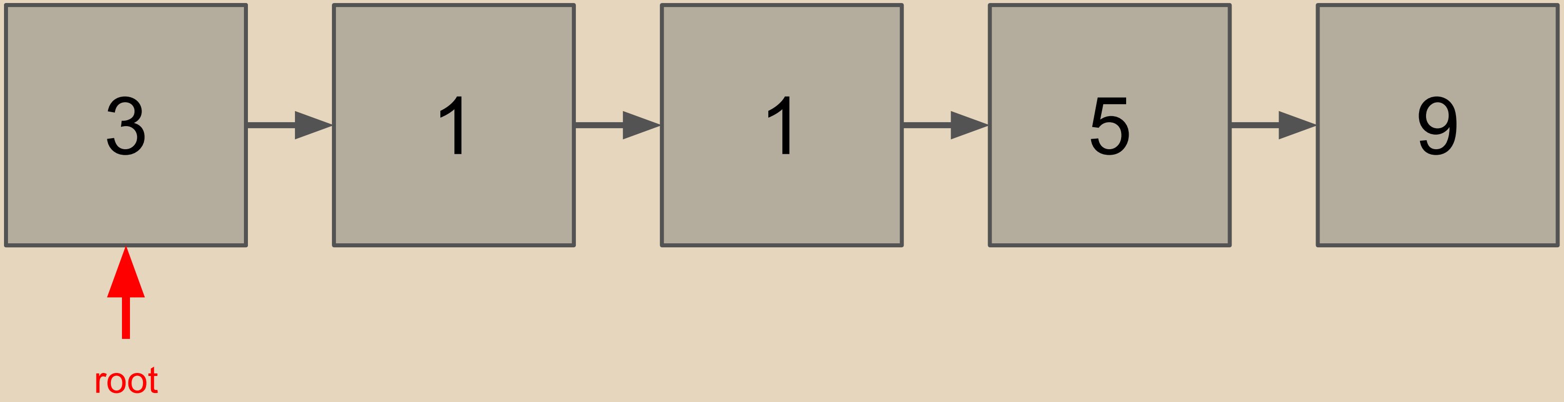
- Simple 'complex' data structure
- Supports iteration, constant-time insertion, constant-time deletion
- No constant-time random access!



(C 6.26-73)

Linked lists

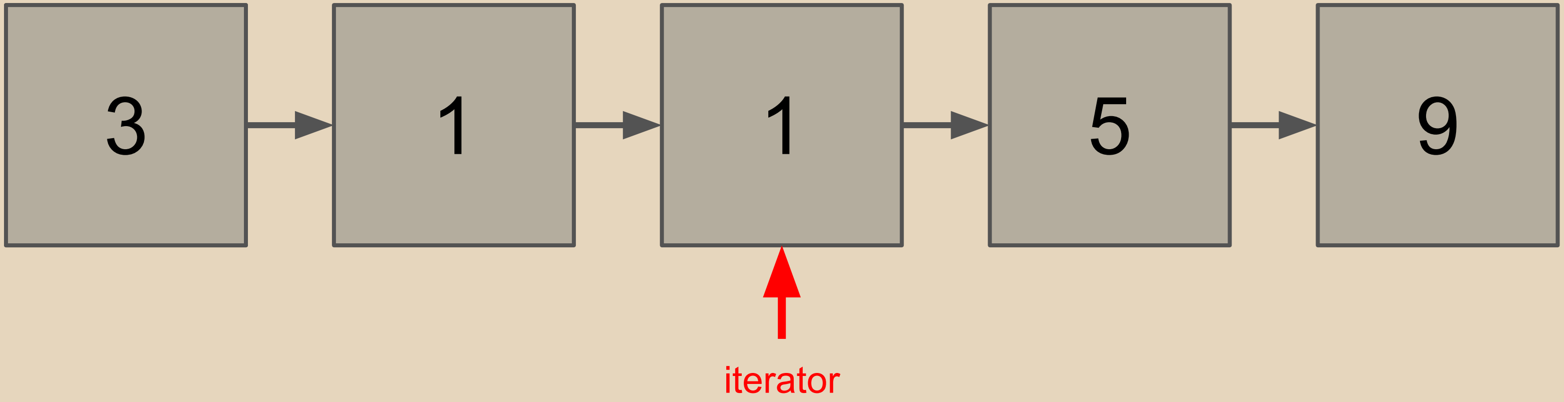
- Simple 'complex' data structure
- Supports iteration, constant-time insertion, constant-time deletion
- No constant-time random access!



(C 6.26-73)

Linked lists

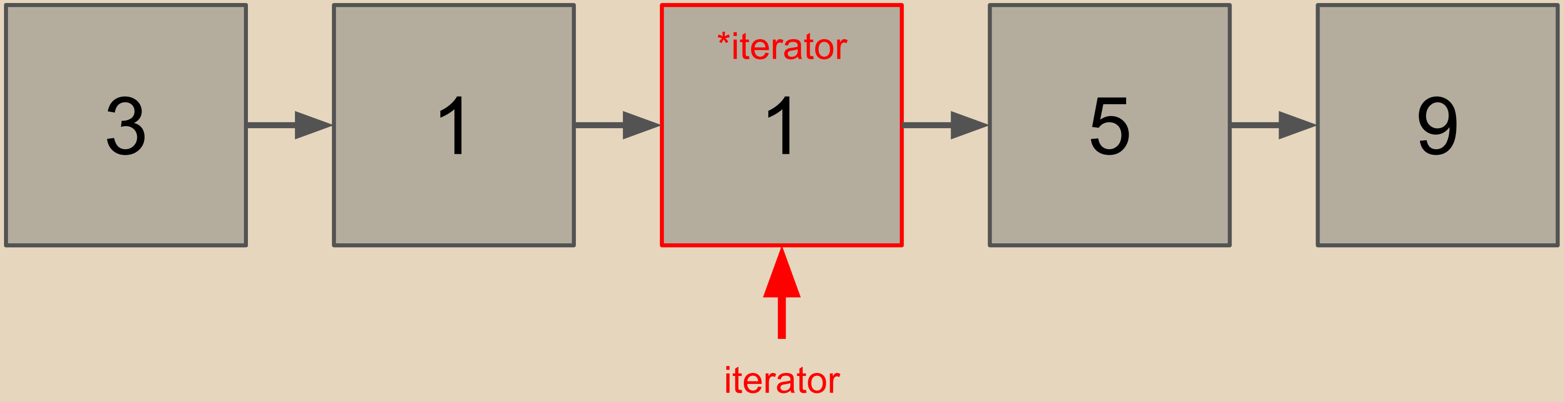
- Simple 'complex' data structure
- Supports iteration, constant-time insertion, constant-time deletion
- No constant-time random access!



(C 6.26-73)

Linked lists

- Simple 'complex' data structure
- Supports iteration, constant-time insertion, constant-time deletion
- No constant-time random access!



(C 6.26-73)

Linked lists

- With some creativity we can come up with ways to...
  - Add elements
  - Delete elements
  - Retrieve elements by index
  - Print elements in order
- But we're still working with 'loose' data structures
- Want some way to neatly bundle <u>functionality</u> with <u>data</u>

Classes!

- Classes and OOP are a very deep topic.
- We can only hope to scratch the surface in these two days.
- For an in-depth coverage, refer to the CS11 C++ slides.
- Definitely take CS11 C++ for a deeper treatment of object-oriented programming.

Classes!

- structs with extra-fancy features

```
class Polygon {
private:
   double width, height;
public:
   Polygon() { ... }
   ~Polygon() { ... }
   void SetValues(double w, double h)
    { ... }
};
```

(C++ 1.16)

Classes

- Unlike C-style structs, classes have **member functions.**
  - These can act on the calling instance.
- Unlike C-style structs, classes have **member visibility.**
  - **Private**: accessible only to class code (default)
  - **Public**: accessible to all code
  - **Protected**: accessible to class code and subclasses

(Note: in C++, structs are just classes with default public visibility. They are much more limited in C.)

(C++ 1.17, 1.23)

Classes

What's the point, when we already have structs?

- **Abstraction** - present a simple interface; hide details from the user.
- **Encapsulation** - protect internal state from unwanted changes; control state changes.

Classes

- We usually place a class's declaration in a header file (e.g. Vector2.hpp).

```
class Vector2
{
private:
    double x, y;
public:
    Vector2();                          // Constructors
    Vector2(double a, double b);
    ~Vector2();                         // Destructor
    double GetX();                      // Accessors
    double GetY();
    double GetLength();
    void SetX(double val);              // Mutators
    void SetY(double val);
};
```

(C++ 1.24-25)

Classes

- We place a class's implementation in a source file (e.g Vector2.cpp)

```
// constructor
Vector2::Vector2()
{
    x = 0;
    y = 0;
}
double Vector2::Length()
{
    return sqrt(x*x + y*y);
}
```

(C++ 1.26-27)

Classes

- `Vector2::Vector2()` is a **constructor**
  - Run whenever an instance is instantiated
  - Used to initialize member variables, acquire resources, etc.
  - Can receive arguments
- `Vector2::~Vector2()` is a **destructor**
  - Run whenever an instance is deleted or goes out of scope
  - Used to clean up dynamic resources
  - Never receives arguments
- These functions never return anything
  - not even `void`

(C++ 1.19)

Classes

- `Vector2::GetX()` is an **accessor**
  - Retrieves internal state
  - Object controls <u>how</u> and <u>when</u> data can be retrieved
  - External modules don't worry about internal state
- `Vector2::SetX()` is a **mutator**
  - Changes internal state
  - Object controls <u>how</u> and <u>when</u> data can be changed
  - External modules don't care how this data is stored

Classes

- Classes are <u>blueprints</u> for **objects**
  - also called **instances**
- We can create as many independent instances of a given class as we want

```
// static
Vector2 x;
Vector2 y(3.0, 6.0);
Vector2 z = Vector2(5.0, 8.0);
// dynamic
Vector2 * p = new Vector2;
Vector2 * q = new Vector2(7.5, 6.3);
Vector2 * arr = new Vector2[80];
```

(C++ 1.19)

Classes

- We can now use the `Vector2` class as a new type

```
#include "Vector2.hpp"
...
Vector2 v1;

v1.SetX(314.159);
v1.SetY(265.358);
printf("len(v1) = %f\n", v1.GetLength());
```

- But we have no direct access to the internals

```
v1.x = 3.78; // NOT ALLOWED (compile error)
```

`this`

- In class context, `this` is a pointer to the calling object.

```
void Vector2::SetX(double x)
{
  this->x = x;
}
```

Templates

- Recall that C++ is a **statically typed** language.
- Recall our linked-list example from earlier.
    - Linked list of integers
- What if we want a linked list of **`Vector2`**?
- Don't want to copy a whole bunch of code!
    - Where possible, <u>don't repeat yourself</u>
    - What if we copy an implementation with bugs?

(C++ 6.19)

Templates

- Templates allow us to apply one code pattern to many data types.
- Templates take one or more types as "parameters".

```
template <class T>
T sum(T a, T b)
{
    T result = a + b;
    return result;
}
...
printf("%d\n", sum<int>(8, 11));
```

(C++ 6.20-31)

Templates

- Classes and structs can also be templated.

```
template <class T>
struct node
{
   T data;
   node<T> * next;
};
...
node<double> n;
n.data = 3.14159;
```

(C++ 6.20-31)

Templates

- N.B.: As a general rule, template *classes* must be completely defined in header file!

```
template <class T>
class Vector2
{
...
public:
    Vector2(T x, T y)
    {
        this->x = x;
        this->y = y;
    }
...
}
```

(C++ 6.20-31)

Next time...

- Debugging tools
- Extra topics (as time permits)
    - Operator overloading
    - Class hierarchy
    - The C++ STL