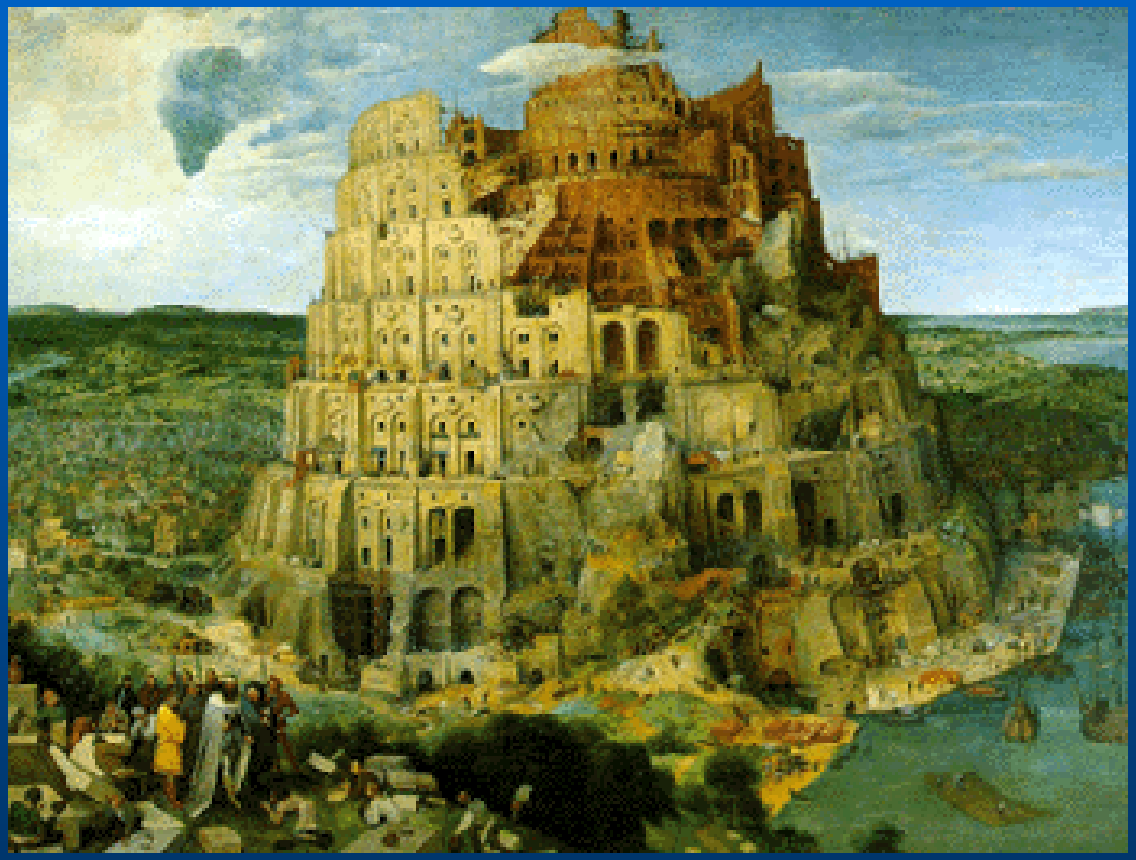
CS-034
From C to C++

Thomas Doeppner
Pascal Van Hentenryck

From C to C++


WELCOME TO THE OFFICIAL CIVILIZATION III WEB SITE
SID MEIER'S
CIVILIZATION III

From C to C++



Memory Allocation

```
int *i = (int*) malloc(sizeof(int));
```

```
int *i = new int;
```

Memory Allocation for Structs

```
struct Complex {
    int real;
    int imag;
};

int main() {
    Complex* c = new Complex;
}
```

Memory Allocation for Arrays

```
int *a = (int*) malloc(n*sizeof(int));
```

```
int *a = new int[n];
```

Memory Deallocation

```
struct Complex {
    int real;
    int imag;
};

int main() {
    Complex* c = new Complex;
    delete c;
}
```

Memory Deallocation

```
int main() {
    int *a = new int[10];
    delete[] a;
}
```

- Need to tell C++ that we delete an array

Memory allocation/deallocation

If you program in C

- use malloc and free (you have no choice anyway)

If you program in C++

- use new and delete
- do not mix malloc/free and new/delete on the same objects

Streams

Standard input/output and files

- elegant and typed specification of input/output

```
int main() {

   int i = 3;
   cout << "i: " << i << endl;
}
```

standard output

end of line

Classes

Constructor

Initializers

```
class Stack {
    int st[MaxStack];
    int top;
public:
    Stack() : top(0) { }
    void push(int v) { st[top++] = v; }
    int pop() { return st[--top]; }
    bool empty() { return top ==0; }
};
```

Classes

```
int main() {

    Stack s;

    s.push(3);
    s.push(5);
    s.push(6);

    cout << "Pop: " << s.pop() << endl;
    cout << "Pop: " << s.pop() << endl;
    cout << "Pop: " << s.pop() << endl;
}
```

Stack declaration

method calls

Classes in C++

Efficiency

- no VTBL for methods
- compiled as function calls
  - first argument is `this`

Stack allocation

- the class is allocated on the stack
  - how much space does it take?
- it disappears when the function returns
  - so long, baby! it was nice to meet you

Constructors

```
class Stack {
    int st[MaxStack]; int top;
public:
    Stack() : top(0) { }
```

- **Instance variables can be initialized before the body of the constructor is executed**

Classes

dynamic array declaration

```
class Stack {
   int* st;
   int size; int top;
public:
   Stack(int s) : top(0), size(s) {
     st = new int[s];
   }
   …
};
```

dynamic array

Classes

```
int main() {

    Stack s(10);

    s.push(3);
    s.push(5);
    s.push(6);
    cout << "Pop: " << s.pop() << endl;
    cout << "Pop: " << s.pop() << endl;
    cout << "Pop: " << s.pop() << endl;
}
```

Stack declaration

method calls

Classes

Static allocation

- The class is allocated on the stack
- The array is allocated on the heap

Dynamic allocation

- What happens when the function exits?

Welcome the Terminator

Welcome the Terminator

Terminators


```
class Stack {
   int* st; int size; int top;
public:
  Stack(int s):top(0),size(s){ st = new int[s];}
   ~Stack() { cout << “so long, stack!” <<endl;}
  …
};
```

The Terminator

Terminators

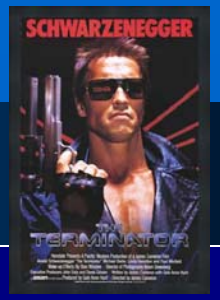

```
int main() {

    Stack s(10);

    s.push(3);
    s.push(5);
    s.push(6);
    cout << "P
    cout << "P
    cout << "P
}
```

```
/Users/pvh/courses/cs034 [48] -> a.out
Pop: 6
Pop: 5
Pop: 3
so long, stack!
FlyingComet /Users/pvh/courses/cs034 [49] ->
```

Terminators




```
class Stack {
   int* st; int size; int top;
public:
  Stack(int s):top(0),size(s){ st = new int[s];}
   ~Stack() {
     cout << “so long, stack!” <<endl;
     delete[] st;
   }
  …
};
```

The Terminator

Terminators

Called

- when the object goes out of scope

Clean the mess

- deallocate memory
- possibly delete other objects
- may close files/databases/…

Because of copyright issues

- they are also called **destructors**

Dynamic Object Allocation

```
int main() {

    Stack* s = new Stack(10);

    s->push(3)
    s->push(5)
    s->push(6)
    cout << "P
    cout << "P
    cout << "Pop: " << s->pop() << endl;
}
```

```
/Users/pvh/courses/cs034 [51] -> a.out
Pop: 6
Pop: 5
Pop: 3
FlyingComet /Users/pvh/courses/cs034 [52] ->
```

Dynamic Object Allocation

```
int main() {

    Stack* s = new Stack(10);

    s->push(3)
    s->push(5)
    s->push(6)
    cout << "P
    cout << "P
    cout << "P
    delete s;
}
```

```
/Users/pvh/courses/cs034 [51] -> a.out
Pop: 6
Pop: 5
Pop: 3
so long, stack!
FlyingComet /Users/pvh/courses/cs034 [52] ->
```

Terminators/Destructors

Called

- when a static object goes out of scope
- when a dynamic object is deleted

Separation of Interface/Implementation

```
class Stack {
   int* st; int top;
Public:
   Stack(int s) : top(0) { st = new int[s]; }
   ~Stack() {
     delete[] st; □□□
     cout << “so long, stack!” << endl;
   }
   void push(int v) { st[top++] = v; }
   int pop() { return st[--top]; }
   bool empty() { return top ==0; }
};
```

Separation of Interface/Implementation

```
class Stack {
    int* st; int top;
Public:
    Stack(int s);
    ~Stack();
    void push(int v);
    int pop();
    bool empty();
};
```

Separation of Interface/Implementation

```
Stack::Stack(int s) : top(0) {
     st = new int[s];
}
Stack::~Stack() {
     delete[] st;
     cout << "so long, stack!" << endl;
}
void Stack::push(int v) { st[top++] = v; }
int Stack::pop() { return st[--top]; }
bool Stack::empty() { return top ==0; }
```

Class Qualifier

Inheritance

```
class EStack : public Stack {
    int size;
public:
    EStack(int s);
    int getTop();
}
```

new class

super class

Inheritance

```
class EStack : public Stack {
    int size;
public:
    EStack(int s);
    int getTop();
}
```

Parent constructor

```
EStack::EStack(int s) : Stack(s), size(s) {}
```

Inheritance

```
int EStack::getTop() {
    return st[top];
}
```

class.c: In member function `int EStack::getTop()':
class.c:6: error: `int*Stack::st' is private
class.c:20: error: within this context
class.c:7: error: `int Stack::top' is private
class.c:20: error: within this context

Inheritance

Inheritance requires estate planning

Inheritance

Visibility of instance variables

- Determines who can access them

Private

- only the class can access

Public

- world visible

Protected

- only the subclasses can access

Estate Planning

```
class Stack {
protected:
    int* st;
    int top;
Public:
    Stack(int s);
    ~Stack();
    void push(int v);
    int pop();
    bool empty();
};
```

accessible to subclasses

Inheritance

```
class EStack : public Stack {
    int size;
Public:
    EStack(int s);
    int getTop();
    void resize();
};
```

growing the array on demand!

Resize

```
void Estack::resize() {
   if (top >= size) {
     int *nst = new int[size*2];
     for(int i = 0; i < top; i++)
          nst[i] = st;
     delete[] st;
     st = nst;
     size *= 2;
   }
}
```

No overhead asymptotically

Push

```
void EStack::push(int s) {
    resize();
    Stack::push(s);
}
```

Daddy's push

Who is pushing here?

```
int main() {

    EStack* s = new EStack(10);

    s->push(3);
    s->push(5);
```

```
FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
EStack::push
Stack::push
FlyingComet /Users/pvh/courses/cs034 [90] ->
```

Who is pushing here?

```
int main() {

    Stack* s = new EStack(10);

    s->push(3);
    s->push(5);
```

```
FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
Stack::push
FlyingComet /Users/pvh/courses/cs034 [90] ->
```

Inheritance

```
class EStack : public Stack {
    int size;
Public:
    EStack(int s);
    ~Estack() {
      cout << “so long, Estack!” << endl;
    }
    int getTop();
    void resize();
};
```

Which Terminator?

```
int main() {

    Stack* s = new EStack(10);
    delete s;
}
```

Which Terminator?

```
int main() {

   EStack* s = new EStack(10);
   delete s;
}
```

```
FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
so long, Estack!
so long, stack!
FlyingComet /Users/pvh/courses/cs034 [90] ->
```

Which Terminator?

```
int main() {

   Stack* s = new EStack(10);
   delete s;
}
```

```
FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
so long, stack!
FlyingComet /Users/pvh/courses/cs034 [90] ->
```

Virtual Methods

```
class Stack {
    int* st; int top;
Public:
    Stack(int s);
    ~Stack();
    virtual void push(int v);
    int pop();
    bool empty();
};
```

Who is pushing here?

```
int main() {

    Stack* s = new EStack(10);

    s->push(3);
    s->push(5);
```

```
FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
EStack::push
Stack::push
FlyingComet /Users/pvh/courses/cs034 [90] ->
```

Inheritance

Virtual methods

- like Java methods

Distinguish two types

- static type: the type of the variable at hand
- dynamic type: the type when the object was created

```
EStack* es = new EStack(10);
Stack s = es;
```

Inheritance

Virtual methods

- like Java methods

Distinguish two types

- static type: the type of the variable at hand
- dynamic type: the type when the object was created

Traditional methods

- dispatch based on the static type

Virtual methods

- dispatch based on the dynamic type

Virtual Terminators

```
class Stack {
    int* st; int top;
Public:
    Stack(int s);
    virtual ~Stack();
    virtual void push(int v);
    int pop();
    bool empty();
};
```

Virtual Terminator

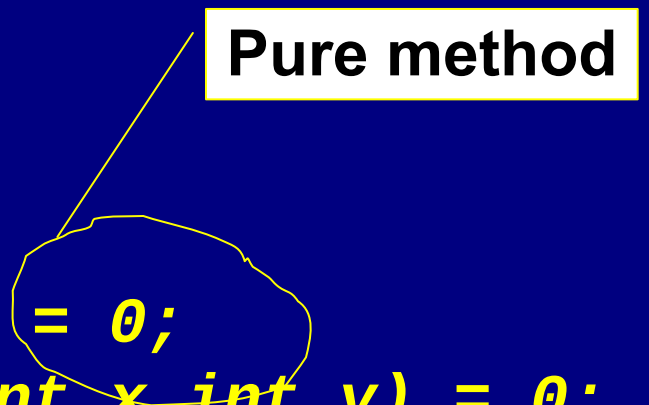
```
int main() {

    Stack* s = new EStack(10);
    delete s;
}
```

FlyingComet /Users/pvh/courses/cs034 [89] -> a.out
so long, Estack!
so long, stack!
FlyingComet /Users/pvh/courses/cs034 [90] ->

Virtual Pure Methods

```
class Shape {
Public:
    Shape() {}
    virtual ~Shape();
    virtual void draw() = 0;
    virtual void move(int x,int y) = 0;
};
```



Inheritance

Pure Virtual methods

- not defined in the superclass
- must be defined in a subclass

Objects

- can only be created when the pure virtual methods are implemented (by subclasses)

Multiple Inheritance











- A class may inherit from several classes
- It will inherit the instance variables!
- It will inherit the methods!

Multiple Inheritance

```
class Employee {
   char *name;
public:
   Employee(char *name);
   char *getName();
};
```

```
class Student {
   char *id;
public:
   Student(char *id);
   char *getId();
};
```

```
class TA: public Employee, public Student {
public:
   TA(char* name,char* id)
      : Employee(name), Student(id) {}
};
```

Multiple Inheritance

```
class Employee {
   char *name;
public:
   Employee(char *name);
   char *getName();
   char *getLogin();
};
```

```
class Student {
   char *id;
public:
   Student(char *id);
   char *getId();
   char *getLogin();
};
```

What happens with `getLogin()`?

```
class TA: public Employee, public Student {
public:
   TA(char* name,char* id);
};
```

Multiple Inheritance

Name clashes are handled at compile time

- **no method** `getLogin()` **in class** `TA`
- **access to the superclass methods:** `Student::getLogin()`
- **possibly redefine** `getLogin()` **in the subclass**

Observe

- **essentially the opposite of Java**
- **inheritance of instance variables**
- **no inheritance of common names;**

Operator Overloading

```
class Complex {
    float r;
    float i;
 public:
    Complex(float _r,float _i): r(_r), i(_i) { }
    void print();
    float getReal() { return r; }
    float getImg() { return i; }
};
```

Operator Overloading

```
Complex operator+(Complex c1,Complex c2) {
   return Complex(c1.getReal() + c2.getReal(),
                  c1.getImg()  + c2.getImg());
}
```

```
FlyingComet /Users/pvh/courses/cs034 [33] -> a.out
(7,8)
```

```
   Complex c1(5,3);
   Complex c2(2,5);
   Complex r = c1 + c2;
   r.print();
}
```

Operator Overloading

```
class Complex {
   float r;
   float i;
 public:
   Complex(float _r,float _i) : r(_r), i(_i) { }
   void print();
   float getReal() { return r; }
   float getImg() { return i; }

   friend Complex operator+(Complex c1,Complex c2);
};
```

gives access to private information

Operator Overloading

```
Complex operator+(Complex c1,Complex c2) {
   return Complex(c1.r + c2.r,c1.i + c2.i);
}

int main() {
   Complex c1(5,3);
   Complex c2(2,5);
   Complex r = c1 + c2;
   r.print();
}
```

Operator Overloading

```
Complex operator+(Complex c,float r) {
    return Complex(c.r + r,c.i);
}

int main() {
    Complex c1(5,3);
    Complex c2(2,5);
    Complex r = c1 + 5.0;
    r.print();
}
```

Operator Overloading

Plenty of operators

- **+, -, =, >=, ==,…, <<, ….**

Fundamental

- **in implementing various expressions**
- **natural reading**

Parameter Passing

Passing by value (Java, c, …)

- pass the value on the stack
  - the value may be an address, an int, a float

Passing by reference (Pascal, C++)

- pass the address of an object
- combines the flexibility of address with the readibility of values
  - no more stars, …

Operator Overloading

```
void swap(int& a,int& b) {
    int t = a;
    a = b;
    b = t;
}
int main() {
    int x = 4, y = 6;
    swap(x,y);
    cout << "x: " << x << "  y: " << y << endl;
}
```

Pass the address

Operator Overloading

```
void swap(int& a,int& b) {
    int t = a;
    a = b;
    b = t;
}
int main() {
    int x = 4, y = 6
    swap(x,y);
    cout << "x: " <<
}
```

```
void swap(int* a,int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int main() {
    int x = 4, y = 6;
    swap(&x,&y);
}
```

Questions…