Where do we go from here?

- **C++ (3-4 lectures)**
  - classes and objects again, with all the moving parts visible
  - operator overloading
  - templates, STL

- **Visual Basic**
  - user interfaces
  - component-based software
  - viruses?

- **C#, .NET**

- **XML and friends**
  - SOAP, XSLT, WSDL, ...

- **Guest lectures**
  April 1: Chris Karr ’02, Northwestern University
  “I have a $150,000 computer science degree. Now what?”
  April 15: Sean Dorward ’91, Google
  April 22: Clayton Marsh ’85, General Counsel’s Office

Stacks in C: a single stack

```c
int stack[100];
int *sp = stack; /* first unused */

#define push(n) (*sp++ = (n))
#define pop() (*--sp)

for (i = 0; i < 10; i++)
    push(i);
```
Stacks in C: a stack type

typedef struct {
    int stk[100];
    int *sp;
} stack;

int push(stack s, int n) {
    return *s.sp++ = n;
}

int pop(stack s) {
    return *--s.sp;
}

stack s1, s2;
for (i = 0; i < 10; i++)
    push(s1, i);

Another stack implementation

typedef struct {
    int *stk;  // allocated dynamically
    int *sp;
} stack;

int push(stack *s, int n) {
    return *s->sp++ = n;
}

int pop(stack *s) {
    return *--s->sp;
}

stack *s1, *s2;  not initialized
s1 = (stack *) malloc(sizeof(stack));
s1->stk = (int *) malloc(100*sizeof(int));
    ugly
for (i = 0; i < 10; i++)
    push(*s1, i);
Problems

- representation is visible, can't be protected (e.g., s1->stk)
- creation and copying must be done very carefully
  - and you don't get any help with them
- no initialization
  - you have to remember to do it
- no help with deletion
  - you have to recover the memory when not in use
- weak argument checking between declaration and call
  - easy to get inconsistencies

- the real problem: no abstraction mechanisms
  - complicated data structures can be built,
    but access to the representation can't be controlled
  - you can't change your mind once the first implementation has been done

- abstraction and information hiding are
  - nice for small programs
  - absolutely necessary for big programs

C++

- designed & implemented by Bjarne Stroustrup
  - began ~ 1980; ISO standard 1998

- a better C
  - more checking of interfaces (ANSI C)
  - other features for easier programming

- data abstraction
  - you can hide HOW something is done in a program,
    reveal only WHAT is done
  - HOW can be safely changed as program evolves

- object-oriented programming
  - inheritance -- new types can be defined that inherit properties from previous types
  - polymorphism or dynamic binding -- function to be called is determined by data type of specific object at run time

- parameterized types
  - define families of related types, where the type is a parameter
  - templates or "generic" programming
C++ classes

• data abstraction and protection mechanism
derived from Simula 67  (Kristen Nygaard, Norway)

```cpp
class thing {
    public:
        methods -- functions that define what operations can
                   be done on this kind of object
    private:
        variables and functions that implement the operations
};
```

• defines a data type 'thing'
  - can declare variables and arrays of this type, create
    pointers to them, pass them to functions, return
    them, etc.

• object: an instance of a class variable

• method: a function defined within the class

• private variables and functions are not accessible
  from outside the class

• it is not possible to determine HOW the
  operations are implemented, only WHAT they do.

C++ synopsis

• data abstraction with classes
  - a class defines a type that can be used to
    declare variables of that type,
    control access to representation

• operator and function name overloading
  - all C operators (including assignment, (), [], ->,
    argument passing and function return) can be
    overloaded so they apply to user-defined types

• control of creation and destruction of objects
  - initialization of class objects
  - recovery of resources on destruction

• inheritance: derived classes built on base classes
  - virtual functions override base functions
  - multiple inheritance: inherit from more than one class

• exception handling
• namespaces for separate libraries
• templates (generic types)
• Standard Template Library
  - generic algorithms on generic containers

• compatible (almost) with C
  - except for new keywords
Stack class in C++

// stk1.c: stack of ints, 1st draft; no checking!!!

class stack {
    private: // this is the default
        int stk[100];
        int *sp; // next free place
    public:
        int push(int);
        int pop();
};

int stack::push(int n) // push n onto stack
{
    return *sp++ = n;
}

int stack::pop() // pop top element
{
    return *--sp;
}

Testing stk1.c

#include <stdio.h>

main()
{
    stack s;
    int i;

    for (i = 0; i < 10; i++)
        s.push(i);
    for (i = 0; i < 10; i++)
        if (s.pop() != 9-i)
            printf("oops: %d\n", i);
}

$ g++ -g stk1.c
$ a.out
Constructors: making a new object

// stk2.c: constructors
class stack {
  private:
    int stk[100];
    int *sp;    // next free place
  public:
    stack();   // constructor
    int push(int);
    int pop();
  
  stack::stack() { sp = stk; }
  int stack::push(int n)   // push n onto stack
  {
    return *sp++ = n;
  }
  int stack::pop()        // pop top element
  {
    return *--sp;
  }
};

Testing stk2.c

main()
{
  stack s1, s2;
  int i;

  for (i = 0; i < 10; i++)
    s1.push();
  for (i = 0; i < 10; i++)
    s2.push(s1.pop());
  for (i = 0; i < 10; i++)
    if (s2.pop() != i)
      printf("oops: %d\n", i);
}
Constructors and destructors

• constructor:
  creating a new object (including initialization)
  - implicitly, by entering the scope where it is declared
  - explicitly, by calling `new`

• destructor:
  destroying an existing object (including cleanup)
  - implicitly, by leaving the scope where it is declared
  - explicitly, by calling `delete` on an object created by `new`

• construction includes initialization, so it may be parameterized
  - by multiple constructor functions with different args
  - an example of function overloading

• `new` can be used to create an array of objects
  - in which case `delete` can delete the entire array

Implicit and explicit

• implicit:

  `f() {
    stack s;
    // calls constructor stack::stack()
    ...
    // calls stack::~stack() implicitly
  }`

• explicit:

  `f() {
    stack *sp = new stack;
    // calls stack::stack()
    ...
    delete sp; // calls stack::~stack()
    ...
  }`
Implicit and explicit

• implicit:

```cpp
f() {
    int i;
    stack s;
    // calls constructor stack::stack()
    ...
    // calls stack::~stack() implicitly
}
```

• explicit:

```cpp
f() {
    int *ip = new int;
    stack *sp = new stack;
    // calls stack::stack()
    ...
    delete sp; // calls stack::~stack()
    delete ip;
    ...
}
```

Memory allocation: `new` and `delete`

• `new` is a type-safe alternative to `malloc`
  - `delete` is the matching alternative to `free`

• `new T` allocates an object of type `T`, returns pointer to it
  - `stack *sp = new stack;`
  - `new T` uses `T`'s constructor for objects of type `T`
  - need a default constructor for array allocation

• `new T[n]` allocates array of `T`'s, returns pointer to first
  - `int *stk = new int[100];`
  - by default, throws exception if no memory
  - `new uses T's destructor ~T()`

• `delete p` frees the single item pointed to by `p`
  - `delete sp;`

• `delete [] p` frees the array beginning at `p`
  - `delete [] stk;`
  - `delete []` frees the array beginning at `p`

• use `new/delete` instead of `malloc/free`
  - `malloc/free` provide raw memory but no semantics
  - this is inadequate for objects with state
  - `never` mix `new/delete` and `malloc/free`
Overloaded functions; constructors

• two or more functions can have the same name if
the number and/or types of arguments are
different

abs(int); abs(double); abs(complex)
atan(double x); atan(double y, double x);

int abs(int x) { return x >= 0 ? x : -x; }
double abs(double x) { return x >= 0 ? x : -x; }
...

• multiple constructors for a class are a common
instance

stack::stack( );
stack::stack(int stacksize);

stack s; // default stack::stack()
stack s1(100); // stack::stack(100)
stack s2 = 100; // also stack::stack(100)

Dynamic stack with new, delete

// stk3.c: new, destructors, delete: explicit size
class stack {
private:
int *stk;    // allocated dynamically
int *sp;      // next free place
public:
stack(); // constructor
stack(int n); // constructor
~stack();   // destructor
int push(int);
int pop();
};

stack::stack()
{
    stk = new int[100]; sp = stk;
}

stack::stack(int n)
{
    stk = new int[n]; sp = stk;
}

stack::~stack() { delete [ ] stk; }
Where are we?

• a class is a user-defined type
• an object is an instance (variable or value) of that type
• public part defines interface it supports
  - member functions in the public part define legal operations on an object
• private part defines implementation
  - functions and data values that implement interface
• constructors are members that define how to create new instances
• destructor is a member that defines how to destroy an instance (e.g., how to recover its resources)

• there's more to constructors
  - show up implicitly in declarations, function arguments, return values, assignment
  - the meaning of explicit and implicit copying must be part of the representation

Inline definitions, default arguments

// stk4.c: inline definitions, default size

class stack {
    int *stk;       // allocated dynamically
    int *sp;        // next free place
public:
    stack(int n);
    ~stack()        { delete [] stk; }
    int push(int n) { return *sp++ = n; }
    int pop()       { return *--sp; }
    int top()       { return sp[-1]; }
};
inline stack::stack(int n = 100)
{                        // ^ default argument
    stk = new int[n];   // default argument
    sp = stk;
}

main() {
    stack s1(10), s2;

    // could use 2 constructors instead of default arg
    // stack(); stack(int n);
Overloaded functions / default args

- default arguments: syntactic sugar for a single function
  - stack::stack(int n = 100);
- declaration can be repeated if the same
  - explicit size in call
    - stack s(500);
- omitted size uses default value
  - stack s;

- overloaded functions: different functions, distinguished by argument types
  - these are two different functions:
    - stack::stack(int n);
    - stack::stack();

---

Change of representation

// stk5.c: change of representation (no checking)

class stack {
  private:
    struct blk {  // private to this class
      int n;
      blk *nb;
      blk(int sz = 0, blk *next = 0);
    };
    blk *sp; // top == head of the list
  
  public:
    stack(int = 0) { sp = 0; }
    ~stack() { while (sp) pop(); }  
    int push(int n);
    int pop();
    int top() { return sp->n; }
};
Representation as linked list

```c
stack::blk::blk(int sz, blk *next)
{
    n = sz;
    nb = next;
}

int stack::push(int n)
{
    blk *bp = new blk(n, sp);
    sp = bp;
    return n;
}

int stack::pop()
{
    blk *bp = sp;
    int n = sp->n;
    sp = sp->nb;
    delete bp;
    return n;
}
```

Aside on implementation

- a class is just a struct
  - no overhead
  - no "class Object" that everything derives from
  - member functions are just names
  - definition is such that C++ can be translated into C
  - original C++ compiler was a C++ program ("cfront") that generated C

```c
struct stack { /* sizeof stack == 8 */
    int *stk__5stack ;
    int *sp__5stack ;
};
...
struct stack __1s1 ;
struct stack __1s2 ;
int __1i ;
...
Cfront output, continued...

```c
main() {
    stack s1(10), s2;
    int i;
    for (i = 0; i < 10; i++)
        s1.push(i);
    for (i = 0; i < 10; i++)
        s2.push(s1.pop());
    for (i = 0; i < 10; i++)
        if (s2.pop() != i)
            printf("oops: %d\n", i);
}
```

Where are we now?

- hiding representation with `private`
- member functions for public interface
  - `classname::member()`
- constructors to make new instances and initialize them
- destructors to delete them cleanly
- change of representation
  - as long as the public part doesn't change
- nothing magic about implementation

What we have ignored (besides error checking):

- implications of assignment and initialization
  - declarations, function arguments, function return values
  - if we don't do anything, will get memberwise assignment and initialization

The meaning of explicit and implicit copying MUST be part of the representation