Where do we go from here?

- **C++**
  - classes and objects again, with all the moving parts visible
  - operator overloading
  - templates, STL
- **Visual Basic, C#, .NET**
  - user interfaces
  - component-based software
  - viruses ?
- **XML and friends**
  - XSLT, XPath, XQuery, ...
  - web services; SOAP, WSDL, ...
- **???**
- **Guest lectures**

  April 14: Peter Ullman '91, Woodcock Washburn
  software intellectual property

  April 19: Mary Fernandez *96, AT&T Research
  XML etc.

Complicated data types in C

- **representation is visible, can't be protected**
  - opaque types are sort of an exception
- **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**
  Bell Labs (1979-95) -> AT&T Labs (1995-) -> TAMU (2003)
  - 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)**

```
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:   simple-minded stack class

class stack {
   private:            // default visibility
        int stk[100];
        int *sp;
   public:
        int push(int);
        int pop();
        stack();       // constructor
};

int stack::push(int n)
{
        return *sp++ = n;
}
int stack::pop()
{
        return *--sp;
}
stack::stack() {    // constructor implementation
        sp = stk;
}
```

Inline definitions

- **member function body can be written inside the class definition**
- **this normally causes it to be implemented inline**
  - no function call overhead

```
// stk2.c:   inline member functions

class stack {
    int stk[100];
    int *sp;

  public:
    int push(int n)  { return *sp++ = n; }
    int pop()        { return *--sp; }
    stack()          { sp = stk; }
};
```

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[n]` allocates array of T's, returns pointer to first**
  ```
  int *stk = new int[100];
  ```
  - by default, throws exception if no memory
- **`delete p` frees the single item pointed to by p**
  ```
  delete sp;
  ```
- **`delete [] p` frees the array beginning at p**
  ```
  delete [] stk;
  ```

- **new uses T's constructor for objects of type T**
  - need a default constructor for array allocation
- **`delete` uses T's destructor ~T()**

- **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

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:
        int push(int);
        int pop();
        stack();       // constructor
        stack(int n);    // constructor
        ~stack();        // destructor
};

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

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

stack::~stack() { delete [ ] stk; }
```

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

- **explicit:**

```
f() {
    int *ip = new int;
    stack *sp = new stack;
                 // calls stack::stack()
    …
    delete sp; // calls sp->~stack()
    delete ip;
    …
}
```

Constructors; overloaded functions

- **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();          // same
stack s2(100);       // stack::stack(100)
stack s3 = 100;      // also stack::stack(100)
```

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();
```

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

```
struct stack {  /* sizeof stack == 8 */
int *stk__5stack ;
int *sp__5stack ;
};
...
struct stack __1s1 ;
struct stack __1s2 ;
int __1i ;
 ...
```

Where are we?

- **hiding representation with <u>private</u>**
- **can change representation**
  - as long as the public part doesn't change
- **member functions for public interface**
  - classname :: member()
- **constructors to make new instances and initialize them**
- **destructors to delete them cleanly**
- **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

Operator overloading

- **almost all C operators can be overloaded**
  - new meaning can be defined when one operand is a user-defined (class) type
  - define **operator +** for object of type **T**
    T T::operator+(int n) { ... }
  - define regular + for object(s) of type **T**
    T operator +(T f, int n) { ... }
  - can't redefine operators for built-in types
    int operator +(int n, int m) { ... } is ILLEGAL
- **3 examples**
  - complex numbers
  - IO streams (very briefly)
  - subscripting

Complex numbers

- **a complex number is a pair of doubles**
  (real part, imaginary part)
- **supports arithmetic operations like +, -, ***

- **a basically arithmetic type for which operator overloading makes sense**
  - `complex` added as explicit type in 1999 C standard
  - in C++, can create it as needed

- **also illustrates**
  - `friend` declarations
  - implicit coercions
  - default constructors

Class complex, version 1

```
class complex {
        double re, im;
  public:
        complex(double r, double i)  { re = r; im = i; }
        complex(double r)            { re = r; im = 0; }
        complex()                    { re = im = 0; }

        complex add(complex c);
        complex mul(complex c);
};

complex complex::add(complex c)
{
        complex temp(re, im);  // or complex temp = c;

        temp.re += c.re;
        temp.im += c.im;
        return temp;
}
```

- **multiple constructors for different initializations**
- **no such thing as an uninitialized complex**
  - C runtime error is a C++ compile time error

- **awkward notation: for c = a + b * c:**

```
c = a.add(b.mul(c));
```

Version 2: operator overloading

```
class complex {
        double re, im;
  public:
        complex(double r, double i)  { re = r; im = i; }
        complex(double r)            { re = r; im = 0; }
        complex()                    { re = im = 0; }

        complex operator+(complex c);
        complex operator*(complex c);
};

complex complex::operator+(complex c)
{
        complex temp(re, im);

        temp.re += c.re;
        temp.im += c.im;
        return temp;
}
```

- much better notation:

```
c = a + b * c;
```

- only works if left operand is a complex

Version 3: friend functions, coercions

```
class complex {
        double re, im;
  public:
        complex(double r = 0, double i = 0)
                { re = r; im = i; }

        friend complex operator +(complex, complex);
        friend complex operator *(complex, complex);
};

        complex a(1.1, 2.2), b(3.3), c(4), d;

        c = 2 * a + b * c;
```

- coercion of 2 -> 2.0 -> complex(2.0)
- default arguments achieve same results as overloaded function definitions
- normally write initializers as

```
complex(double r = 0, double i = 0) : re(r), im(i) { }
```

Notes on operator overloading

- **applies to all operators except . and ?:**
  - operator ( )   left-side function calls
  - operator ,   simulates lists
  - operator ->   smart pointers
- **works well for algebraic and arithmetic domains**
  - complex, bignums, vectors & matrices, ...

- **BUT DON'T GET CARRIED AWAY:**
- **you can't change precedence or associativity of existing operators**
  - e.g., if use ^ for exponentiation, precedence is still low
- **you can't define new operators**
- **meanings should make sense in terms of existing operators**
  - e.g., don't overload - to mean + and vice versa

Simple vector class (v0.c)

- **based on overloading operator [ ]**

```
class ivec {
        int *v;         // pointer to an array
        int size;       // number of elements
  public:
        ivec(int n) { v = new int[size = n]; }

        int operator [ ](int n) {      // checked access
                assert(n >= 0 && n < size);
                return v[n];
        }
        int elem(int n) { return v[n]; }   // unchecked
};

main()
{
        ivec iv(10);            // declaration
        int i;

        i = iv.elem(10);        // unchecked access
        i = iv[10];             // checked access
}
```

What about lvalue access?

- **vector element as target of assignment**

```
main()
{
        ivec iv(10);            // declaration

        iv[10] = 1;             // checked access
        iv.elem(10) = 2;        // unchecked access
}
```

```
$  g++ v1.c
v1.c:22: non-lvalue in assignment
v1.c:23: non-lvalue in assignment
$ CC v1.c
"v1.c", line 22: Error: The left operand cannot be
  assigned to.
"v1.c", line 23: Error: The left operand cannot be
  assigned to.
```

- **need a way to access object, not a copy of it**
- **in *C*, use pointers**
- **in *C++*, use references**

References (swap.c)

- **attaching a name to an object**
- **a way to get "call by reference" (var) parameters without using pointers**

```
void swap(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

- **a way to access an object without copying it**

```
stack s;
stack t = s; // may not want to copy

f(s);        // ...
return s;    // ...

stack s, t;
t = s; // want to control the assignment
```

Lvalue access (v2.c)

```
class ivec {
        int *v;         // pointer to an array
        int size;       // number of elements
  public:
        ivec(int n) { v = new int[size = n]; }

        int& operator [](int n) {
                assert(n >= 0 && n < size);
                return v[n]; }

        int& elem(int n) { return v[n]; } // unchecked
};

        ivec iv(10);            // declaration
        iv.elem(10) = 2;        // unchecked access
        iv[10] = 1;             // checked access
```

- **reference gives access to object so it can be changed**

Iostream library (very quick sketch only)

- **how can we do I/O of user-defined types with non-function syntax**

- **C printf can be used in C++**
  - no type checking
  - no mechanism for I/O of user-defined types
- **Java System.out.print(arg) or equivalent**
  - type checking only in trivial sense:
    calls toString method for object
  - bulky, notationally clumsy
    one call per item

- **can we do better?**

- **Iostream library**
  - overloads << for output, >> for input
  - permits I/O of sequence of expressions
  - type safety for built-in and user-defined types
  - natural integration of I/O for user-defined types
    same syntax and semantics as for built-in types

Basic use

- **overload operator << for output, >> for input**
  - very low precedence
  - left-associative, so
    `cout << e1 << e2 << e3`
  - is parsed as
    `(((cout << e1) << e2) << e3)`

- **take an [io]stream& and a data item**
- **return the reference**

```
#include <iostream>
ostream&
operator<<(ostream& o, const complex& c)
{
  o << "(" << c.real() << ", "
           << c.imag() << ")";
  return o;
}
```

- **iostreams cin, cout, cerr already open**
  - correspond to stdin, stdout, stderr

Input with iostreams

```
#include <iostream>

main()
{
        char name[100];
        double val;

        while (cin >> name >> val) {
              cout << name << " = " << val << "\n";
        }
}
```