Outline

• Monday: design, interfaces, representation of information
• Tuesday: testing, debugging, mechanization
• Thursday: programming style

Evolution of programming languages

1940's: machine level
  - raw binary
1950's: assembly language
  - names for instructions and addresses
  - very specific to each machine
1960's: high-level languages
  - Fortran, Cobol, Algol
1970's: system programming languages
  - C
  - Pascal (more for teaching)
1980's: object-oriented languages
  - C++, Ada, Smalltalk, Modula-3, Eiffel, ...
    • strongly typed (to varying degrees)
    • better control of structure of really large programs
    • better internal checks, organization, safety
1990's: scripting, Web, component-based, ...
  - Perl, Java, Visual Basic, ...
    • strongly-hyped languages
2000's: cleanup, or more of the same?
  - C#, Python, ...
    • increasing focus on interfaces, components
Evolution of language features

• better control flow:
  – structured programming
  – recursion
  – exceptions, threads, remote procedure call
• more intrinsic data types:
  – IEEE floating point
  – characters & strings
  – pointers and references
  – complex, ...
• user-defined data types:
  – aggregates: structures / records
  – modules, classes & objects
  – interfaces & abstract data types
• more control of storage management:
  – from COMMON to malloc/free to garbage collection

• compiler technology
• development tools and environments
• mutual influences among languages

Structured programming

• a hot topic in the early 1970's!
• program with a restricted set of control flow constructs
  – statement grouping
  – if-then-else
  – some kind of loop
  – procedures
  – no GOTO statement

• integral part of most languages
• slow to arrive in Fortran (not until Fortran 90)
• and even slower to be accepted in some environments
From Fortran ...

```fortran
   do 1 j=2,jmm
   do 1 k=2,kmm
      matl = mask(112).and.ist(k,j)
      if (matl-1) 30,31,32
  30   loc = shiftr(ist(k,j),32)
      den=fq(loc+1)
      go to 33
  31   den=d(k,j)
      go to 33
  32   den= 1.e-3*d(k,j)
  33   wa(k,j)=den
  1   continue
```

(courtesy of Jim Stone, who is not the author!)

... to Fortran 90

```fortran
   do j = 2, jmm
      do k = 2, kmm
         matl = mask(112).and.ist(k,j)
         if (matl < 1) then
            loc = shiftr(ist(k,j),32)
            den = fq(loc+1)
         else if (matl == 1) then
            den = d(k,j)
         else ! matl > 1
            den = 1.e-3*d(k,j)
         end if
         wa(k,j) = den
      end do
   end do
```

- Fortran: only Arithmetic if
- Fortran 66: if's and goto's
- Fortran 77: if-then-else
- Fortran 90: do ... end do (several forms)
  free-form input, inline comments, ...

•
Subroutines & functions

- subroutines and functions break up a big program into small pieces that interact in controlled ways

- "no subroutine should be longer than a page"?

- design issues
  - argument lists and return types
  - scope
    - local static and automatic variables, private names, ...
  - efficiency
    - internal functions
    - inline functions
    - macros

- recursion
  - subroutines can call themselves

- threads
  - separate threads of control in a single process

- exceptions
  - alternate return path for errors and exceptional conditions

User-defined data types

- structures and other aggregates
  - structures vs parallel arrays
  - e.g., a point type for graphics or simulation or ...
    - real x(100), y(100), z(100)

- Fortran 90 "derived type":
  - type Point
    - real :: x, y, z
  - end type Point

  - defines a new type called Point
  - can declare variables of type Point, use them in subroutine calls & returns, etc.
    - type (Point), dimension(100) :: pt
      - pt(i)%x = pt(j)%y + pt(j)%z

- is it better to have built-ins or a way to make them?
  - e.g., complex
    - a built-in type in Fortran and C99
    - user-defined in C++
Interfaces and abstract data types

- interface: detailed boundary between code that provides a service and code that uses it
- abstract data type (ADT): a data type described in terms of the operations it supports -- its interface -- not by how it is implemented
- examples:
  - math library, BLAS, LAPACK
  - stdio, iostreams, sockets
  - Unix system calls, Windows API
- why use ADT's?
  - can localize implementation: cleaner code, only one place to fix bugs and make improvements
  - can change implementation as necessary without affecting the rest of the program
  - can have multiple implementations behind a single interface
- language mechanisms:
  - C: opaque types
  - C++: classes and objects
  - Fortran: modules

Interface issues

- functionality
  - features and operations provided
  - inputs and return values
- information hiding
  - what parts of implementation are visible
  - what parts are hidden
- resource management
  - creation and initialization
  - maintaining state
  - ownership: sharing and copying
  - memory management
  - cleanup
- error handling
  - what errors are detected?
  - how are they handled or reported?
- other issues
  - efficiency
  - portability
  - convenience, simplicity, generality, consistency, regularity, orthogonality, motherhood, apple pie, ...
Matrix example

• suppose you want a Matrix type

• first look for a library; it may already exist
  - how do you decide whether to use it?
  - “make or buy?”

• interface issues specifically for this type
  - functionality: what operations are provided?
    - what does the user see?
  - hiding the representation
    - what is the implementation that the user doesn’t see?
  - resource management
    - how is memory allocated and freed?
  - error handling
    - what can go wrong and how is it handled?
  - efficiency / performance
    - what are the important operations and how fast are they?
  - ease of use, notation
    - how are the most common operations expressed?
  - etc., etc.

Who manages what memory when?

• a fundamental interface issue
  - getting it wrong or inconsistent is a major problem
  - making it hard for users is a major problem

• who allocates space for a matrix?
  • static or dynamic?
  • can it grow? without limit?
  • who grows it?
  • who complains if it gets too big? how?
  • who owns it?
  • who can change its contents? how?
  • what about assignment and copy?
  • who sees the changes? is it re-entrant?
  • what is its lifetime?
    - when are pointers into the data structure invalidated?
  • who frees it?

• these issues are not all solved by garbage collection
A matrix type in C

• essential idea: hide the representation so user code
doesn't depend on it
  – representation can be changed without affecting user code
  – not much choice about how to present it to the user

• opaque type:
  – a pointer to a structure that is private to the implementation
  – access to structure only through functions that are private to
    the implementation

• user code uses a single typedef
  
  typedef struct Matrix Matrix

• and calls only functions in the interface

  Matrix *M_new(int r, int c);
  void M_put(Matrix *m, int r, int c, double d);
  double M_get(Matrix *m, int r, int c);
  void M_add(Matrix *m3, Matrix *m1, Matrix *m2);
  void M_free(Matrix *m);

Using the Matrix type

typedef struct Matrix Matrix;

int main(int argc, char *argv[]) {
  Matrix *m1, *m2, *m3;
  int i, j;
  double v, v1, v2;
  m1 = M_new(10, 20);
  m2 = M_new(100, 200);
  m3 = M_new(30, 40);
  v1 = v2 = 0;
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 20; j++) {
      M_put(m1, i, j, 1.0 * (i+j));
      M_put(m2, i, j, 2.0 * (i+j));
      v1 -= i + j;
    }
  }
  M_add(m3, m1, m2);
  for (i = 0; i < 10; i++) {
    for (j = 0; j < 20; j++) {
      v = M_get(m3, i, j);
      if (v != 3.0 * (i+j))
        printf("%5d %5d\n", i, j);
      v2 += v;
    }
  }
}
C implementation

struct Matrix {
    int rows, cols;
    double **mp;
};

Matrix *M_new(int r, int c) {
    Matrix *m;
    int i;
    /* BUG: no error checking */
    m = (Matrix *) malloc(sizeof(struct Matrix));
    m->rows = r;
    m->cols = c;
    m->mp = (double **) malloc(r * sizeof(double *));
    for (i = 0; i < r; i++)
        m->mp[i] = (double*) malloc(c * sizeof(double));
    return m;
}

rest of implementation

double M_get(Matrix *m, int r, int c) {
    return m->mp[r][c];
}

void M_put(Matrix *m, int r, int c, double v) {
    m->mp[r][c] = v;
}

void M_add(Matrix *m3, Matrix *m1, Matrix *m2) {
    int i, j;
    for (i = 0; i < m1->rows; i++)
        for (j = 0; j < m1->cols; j++)
            m3->mp[i][j] = m1->mp[i][j] + m2->mp[i][j];
}

void M_free(Matrix *m) {
    int i;
    for (i = 0; i < m->rows; i++)
        free(m->mp[i]);
    free(m->mp);
    free(m);
}
Classes, objects and all that

- data abstraction and protection mechanism
- originally from Simula 67

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

- defines a new data type "thing"
  - can declare variables and arrays of this type, pass to functions, return them, etc.
- object: an instance of a class variable
- method: a function defined within the class
  - (and visible outside)
- private variables and functions are not accessible from outside the class
- not possible to determine HOW the operations are implemented, only WHAT they do
```

A C++ matrix class

- a class is a user-defined type
  - almost the same as a built-in type
  - complete control over life cycle
    - construction, initialization, assignment, copying, destruction
    - operator overloading
  - can hide all aspects of representation

```cpp
class Matrix {
    public:  // methods visible to user
        double get(int r, int c);
        void put(int r, int c, double v);
        Matrix(int r, int c);  // constructor
        ~Matrix();            // destructor

    private:  // user can't see implementation
        int rows, cols;
        double **mp;
        friend Matrix operator +(Matrix &m1,
                                  Matrix &m2);
    }
```
Using the C++ Matrix type

```cpp
int main(int argc, char *argv[]) {
    Matrix m1(10,20), m2(100,200), m3(30,40);
    int i, j;
    double v, v1, v2;

    v1 = v2 = 0;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 20; j++) {
            m1.put(i, j, 1.0 * (i+j));
            m2.put(i, j, 2.0 * (i+j));
            v1 += i + j;
        }
    }

    m3 = m1 + m2;
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 20; j++) {
            v = m3.get(i, j);
            if (v != 3.0 * (i+j))
                printf("%5d %5d\n", i, j);
            v2 += v;
        }
    }
}
```

C++ implementation

```cpp
Matrix::Matrix(int r, int c) {
    /* BUG: no error checking */
    rows = r;
    cols = c;
    mp = new double*[r];
    for (int i = 0; i < r; i++)
        mp[i] = new double[c];
}

double Matrix::get(int r, int c) {
    return mp[r][c];
}

void Matrix::put(int r, int c, double v) {
    mp[r][c] = v;
}

Matrix operator +(Matrix &m1, Matrix &m2) {
    int i, j;
    Matrix m3(m1.rows, m1.cols);
    for (i = 0; i < m1.rows; i++)
        for (j = 0; j < m1.cols; j++)
            m3.mp[i][j] = m1.mp[i][j] + m2.mp[i][j];
    return m3;
}

Matrix::~Matrix() {
    for (int i = 0; i < rows; i++)
        delete[] mp[i];
    delete[] mp;
}
```
there's (lots) more to C++

- operator overloading
- exceptions
- inheritance
- virtual functions, runtime polymorphism
- templates, Standard Template Library
- namespaces

Modules

- module: a Fortran 90 mechanism to collect derived-type declarations, subroutines and functions, parameters, etc., in one place

```fortran
module M
  types
  parameters
  variables
  interfaces
  contains
    subroutines & functions
end module M
```

- module imported into other parts of program as needed
  ```fortran
  use M
  ```

- replaces Common and Include
- permits overloaded function names
- permits some overloaded operators
- permits some hiding of implementation
Fortran version

program main
    use Matrix_module
    type(Matrix) :: m1, m2, m3
    integer :: i, j
    call M_new(m1, 10, 20) ! allocate(m%mp(10,20))
    call M_new(m2, 100, 200)
    do i = 1,10
        do j = 1,20
            call M_put(m1, i, j, 1.0 * (i+j))
            call M_put(m2, i, j, 2.0 * (i+j))
        end do
    end do
    m3 = m1 + m2
    do i = 1,10
        do j = 1,20
            v = M_get(m3, i, j)
            if (v /= 3.0 * (i+j)) then
                print '(I5, I5)', i, j
            end if
        end do
    end do
    call M_free(m1)
    stop
end program main

module Matrix_module
    type Matrix
        integer :: rows, cols
        real, dimension (::,:), pointer :: mp
    end type Matrix
    interface operator (+)  ! matrix+matrix
        module procedure add
    end interface
    contains
    subroutine M_new(m, r, c)
        type (Matrix), intent(out) :: m
        integer, intent(in) :: r, c
        allocate(m%mp(r, c))
        m%rows = r
        m%cols = c
    end subroutine M_new
    function M_get(m, r, c)
        type (Matrix) :: m
        integer, intent(in) :: r, c
        real :: M_get
        M_get = m%mp(r, c)
    end function M_get
end module Matrix_module

Fortran implementation

module Matrix_module
    type Matrix
        integer :: rows, cols
        real, dimension (::,:), pointer :: mp
    end type Matrix
    interface operator (+)  ! matrix+matrix
        module procedure add
    end interface
    contains
    subroutine M_new(m, r, c)
        type (Matrix), intent(out) :: m
        integer, intent(in) :: r, c
        allocate(m%mp(r, c))
        m%rows = r
        m%cols = c
    end subroutine M_new
    function M_get(m, r, c)
        type (Matrix) :: m
        integer, intent(in) :: r, c
        real :: M_get
        M_get = m%mp(r, c)
    end function M_get
end module Matrix_module
Fortran version (continued)

subroutine M_put(m, r, c, v)
type (Matrix) :: m
integer, intent(in) :: r, c
real, intent(in) :: v
m%mp(r, c) = v
end subroutine M_put

function add(m1, m2) result (m3)
type (Matrix), intent(in) :: m1, m2
type (Matrix) :: m3
integer :: i, j

call M_new(m3, m1%rows, m1%cols)
do i = 1, m1%rows
  do j = 1, m1%cols
    m3%mp(i, j) = m1%mp(i, j) + m2%mp(i, j)
  end do
end do
end function add

subroutine M_free(m)
type (Matrix) :: m
deallocate(m%mp)
end subroutine M_free

end module Matrix_module

Things to have noticed

- dimension(:,:)
- pointer
- interface
- operator (+)
- intent(in), intent(out)
- allocate, deallocate
What else?

- what else can you do once the representation is hidden?
- check consistency of sizes, ranges, etc.
- check allocation
  ```fortran
  allocate(mp, stat=astatus)
  if (astatus /= 0) then
    didn't work
  end if
  ```
- re-claim space
- handle special cases like sparse, symmetric, etc., with a uniform interface
  - by overloading functions and operators

Modules vs. classes

- Module:
  - a set of functions, data, etc., to group related pieces
  - interfaces to outside
  - known to each other internally
  - only one instance of a module
  - to create and initialize several objects, do it inside module
    ```fortran
    type(Matrix) :: m
    call M_new(m, r, c)  ! create & init
    ```
- Class
  - set of functions, data, etc., to implement a data type
  - methods provide interface to rest of program
  - known to each other internally
  - constructors permit creating new instances
    - each instance is called an object
    - destructors delete an object
      ```fortran
      Matrix m(r, c): // create & init
      ```
  - a module is (sort of) like a class with only one instance
Inheritance and derived classes

• a way to create or describe one class in terms of another
  – “a D is like a B, with these extra properties...”
  – “a D is a B, plus...”
  – B is the base class or superclass
  – D is the derived class or subclass
    • Perl & C++ use base/derived; Java uses super/sub
• inheritance is used for classes that model strongly related concepts
  – objects share some properties, behaviors, etc.
  – and have some properties and behaviors that are different

• base class has aspects common to all matrices:
  • rows, cols, rank, ...
• derived classes for aspects different for different kinds of widgets:
  • sparse/dense, symmetric, block, diagonal, ...
• one class is a natural extension of the other
  – sometimes you care about the difference
    • inverting, transposing
  – sometimes you don’t
    • rows and columns

Derived classes

```cpp
class Matrix {
    int row, col;
    // other vars common to all Matrices
};
class Sparse : public Matrix {
    float density;
    // other vars specific to Sparse matrices
};
class Dense : public Matrix {
    bool symmetric;
    // other vars specific to Dense matrices
};
```

• a Sparse is a derived class (a kind of) Matrix
  – inherits all members of Matrix
  – adds its own members
• a Dense is also a derived class of Matrix
More derived classes

- derived classes can add their own data members
- can add their own member functions
- can override base class functions with functions of same name and argument types

```cpp
class Sparse : public Matrix {
    float density;
    public void invert() {...} // overrides
    public float dens(int) {...}
}

class Dense : public Matrix {
    bool symmetric;
    public void invert() {...} // overrides
    public bool is_symmetric() {...}
}

Dense d;
Sparse s;

d.invert(); // call Dense.invert
s.invert(); // call Sparse.invert

Matrix m[100];
for (i = 0; i < 100; i++)
    m[i].invert(); // inverts any kind
```

Inheritance principles

- classes are supposed to match the natural objects in the application
- derive specific types from a general type
  - collect common properties in the base class
  - add special properties in the derived classes
- distinctions are not always clear
  - factor as dense/sparse then symmetric/asymmetric?
  - how do we get a symmetric sparse matrix?

Diagram:
```
          Matrix
         /   \
Sparse  Tridiag  Dense
    /   \
Symmm  Banded
```
Summary of inheritance

- a way to describe a family of types
- by collecting similarities (base class)
- and separating differences (derived classes)

- polymorphism: which member function to call is determined at run time

- not every class needs inheritance
  - may complicate without compensating benefit

- use composition instead of inheritance?
  - an object contains (has) an object rather than inheriting from it

- "is-a" versus "has-a"
  - inheritance describes "is-a" relationships
  - composition describes "has-a" relationships

Static vs dynamic memory

- static allocation
  - Fortran COMMON
  - C external arrays

- dynamic allocation
  - malloc/free / pointers
  - new/delete
  - allocate/deallocate

- garbage collection
  - explicit in C and Fortran
  - implicit in Java & scripting languages
  - programmable in C++
    - reference counts, handle classes
Conclusions

• use the best language you can
  – Fortran programmers: use Fortran 90/95 if at all possible
  – C programmers: think about C++

• write it well
  – good code is easier to work with

• plan your interfaces
  – each interface should hide a design decision
  – or a place where a different implementation could be used
  – effort at the beginning pays off at the end