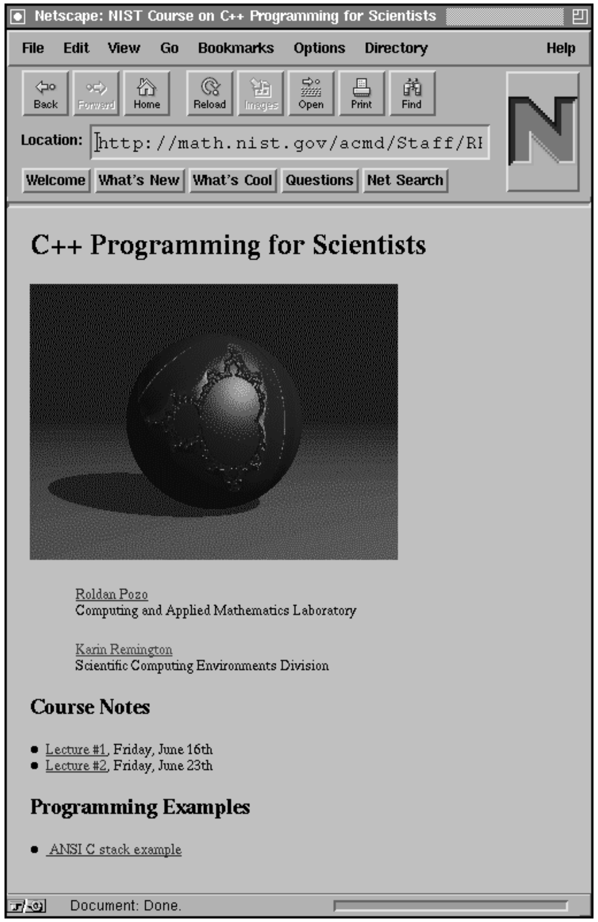
C++ Programming for Scientists

Lecture # 2

C++ Course Web Page

http://math.nist.gov/acmd/Staff/RPozo/class.html



3-minute review on C structures

A C `struct` defines a grouping of one or more related variables. For example,

```
typedef struct
{
    char *title;
    char *author;
    char *publisher;
    float price;
    int num_pages;
} Book;

typedef struct
{
    double real;
    double img;
} Complex;
```

declares the new data types `Complex` and `Book`. This essentially *extends* the language by providing new types which look just like a basic data element (e.g. `float`, `char`)

- Declarations of complex numbers and books now look like

```
Complex u, v, z;
Book B, C;
```

- Accessing elements of a `struct` are performed via the dot (".") operator, as in

```
u.real = 3.0;
u.img = 1.1;

B.name = "Generation X";
B.author = "Douglas Coupland";
B.price = 6.99;
B.num_pages = 123;
```


C structures, cont'd.

- `Complex` and `Books` types can now be used in functions:

```
float add_prices(const Book B[], int N)
{
    int i;
    float sum = 0.0;

    for (i=0; i<N; i++)
        sum += B[i].price;

    return sum;
}

double norm(Complex u)
{
    return sqrt( u.real*u.real + u.img*u.img );
}
```

- for added efficiency, structures can be passed by address:

```
Complex C_add(const Complex *x, const Complex *y)
{
    Complex t;
    t.real = x->real + y->real;
    t.img =  x->img + y->img;

    return t;
}
```

This avoids copying `x` and `y`. Not a big deal for complex numbers (16 bytes), but useful for larger structures (> 100 bytes). Note that `const` specifier is used to denote that the values of `x` and `y` are not changed by `C_add()`.

C++: comments

New "//" symbol can occur anywhere and signifies a comment until the end of line.

```
float r, theta;     // this is a comment.
                    // so is this.

//  theta = 3.0;        a bad way to place a comment...
```

Of course, there are the still two other ways to denote comments:

- with the familiar `/* */` pair

  ```
  /* nothing new here...     */
  ```

  but careful, these do not nest on some compilers (i.e. not a great way to comment-out sections of code which may already contain comments.)

- using the preprocessor via `#ifdef`, `#endif`. This is the best method for commenting-out large sections of code.

  ```
  #if 0
      a = b + c;
      x = u * v;
  #endif
  ```

  Remember that the `#` must occur in the first column.

C++: Movable Declarations

PROBLEM SOLVED: variable declarations aren't close to code.

Local variable declarations in C++ need NOT be congregated at the beginning of functions:

```
double sum(const double x[], int N)
{
   printf("entered function sum().\n");

   double s;                    // Note the declaration here...

   for (int i=0; i<N; i++)      // also notice the declaration
      s += x[i];                // of loop variable "i" used
                                // the "for" expression
   return s;
 }
```

The idea is to declare variables close to the code where they are used. Particularly useful to large subroutines with many local varaibles. This improves code readability and helps avoid type-mismatch errors.

Caveat: the scope of the "`int i`" declaration in the for-loop above is still global to the subroutine. Thus

```
for (int i=0; i<N; i++)
   s += x[i];

for (int i=10; i<M; i++)      // error, "int i" already used.
   s *= y[i];
```

will cause a compile-error, since "i" is used in redeclared in the second for-loop.

C++: functions with default arguments

PROBLEM SOLVED: specifying default arguments to user-defined functions.

Arguments to C++ functions that are often called with the same value can be specified a default value and removed from the parameter list. The default values can only be specificed ONCE. By convention, this is put in the header file (i.e. where it's declared) rather than the .cc file (i.e. where it's implemented.)

- For example,

  ```
  void VectorInit(double x[], int N, double val=0.0);
  ```

  specifies to use a default value of 0.0 unless a value for `val` is passed explicitly. This is determined by the *position* of the argument list, not it's name.

  ```
  VectorInit(X, 10);          // intialize X with zeros

  VectorInit(Y, 20, 3.14);    // override default and initialize
                              // Y[0], Y[1], ... Y[19] with 3.14

  VectorInit(Y, 20, "hello"); // compiler error: 3rd arg mismatch
  ```

- Another example,

  ```
  double my_log(double x, double base=2.7182818284590452354);
  ```

  defaults to compute $\log_e$ as a default. To compute $\log_{10}$, use

  ```
    y = my_log(x);          // defaults to log_e
    y = my_log(x, 10);      // computes log_10
  ```

- PROS: reduces number of function parameters
- CONS: unless default values are intuitively obvious, need good documentation and/or access to header files.

C++: reference parameters

PROBLEM SOLVED: removes proliferations of pesky "`->`" clutter from code.

Since structures are often passed by reference for efficency, application codes often ends up with lots of `->` operators, when referencing their components.
Remember our complex number example?

```
Complex C_add(const Complex *x, const Complex *y)
{
    Complex t;
    t.real = x->real + y->real;
    t.img =  x->img + y->img;

    return t;
}
```

This occurs so often that people sought a better solution:

```
Complex C_add(const Complex &x, const Complex &y)
{
    Complex t;
    t.real = x.real + y.real;
    t.img =  x.img + y.img;

    return t;
}
```

The `&` before each function argument means that it is being passed by *reference* (i.e. by address), but that inside the function it can be treated like any other local variable.

C++: function overloading

PROBLEM SOLVED: baroque function names.

In C++, function *arguments*, as well as the function name, as used for identification. This means it is possible to define the same with different argument *types*. For example,

```
void swap(int *i, int *j)
{
    int t;
    t = a;
    a = b;
    b = t;
}

void swap(Complex *u, Complex *v)
{
    Complex t;
    t = a;
    a = b;
    b = t;
}

void swap(double x, double y)
{
    printf("This swap() routine does nothing.\n");
}
```

How does the C++ compiler know which one to call? By the *arguments* being passed to them:

```
Complex u, v;
int     i, j;

swap(&u, &v);       // swaps complex values of u and v

swap(&i, &j);       // swaps integer values of i and j

swap(i,j);          // does nothing -- promotes i and j to double
                    // and calls double swap(double, double);
```


C++: operator overloading

PROBLEM SOLVED: make user-defined types look like a natural part of the language.

Functions aren't the only thing that can be overloaded in C++; operators (such as +, *, %, etc.) are fair game too.

Given complex numbers, `u`, `v`, `w`, and `z`, which is easier to read?

```
w =  z*(u + v);
```

or

```
Complex t;
C_add(&t, u, v);
C_mult(&w, z, t);
```

How did we do it?

```
Complex operator+(const Complex& a, const Complex &b)
{
    Complex t;

    t.real = a.real + b.real;
    t.img =  a.img + b.img;

    return t;
}

Complex operator*(const Complex &a, const Complex &b)
{
    Complex t;

    t.real = a.real * b.real - a.img * b.img;
    t.img = a.real * b.img + a.img * b.real;

    return t;
}
```

Almost all C++ operators can be overloaded, including `->`, `+=` , and `()`. *More on that later...*

C++: inline functions

PROBLEM SOLVED: excessive function call overheads for small routines.

The aim of inline functions is to reduce the usual overhead associated with a function call. The effect is to substitute *inline* each occurrence of the function call with the text of the function body (similar to *function statements* in Fortran).

```
inline double max(double a, double b)
{
    return  (a > b ? a : b);
}

inline int ten_fold(int i)
{
    return 10*i;
}
```

They're used like regular functions.

- PROS: removes function call overhead by replacing with text of function; provides better opportunities for further compiler optimizations.
- CONS: increases code size, can reduce efficiency if abused (e.g. expanding loop code out of cache). The `inline` modifier is simply a *hint*, not a mandate, to the C++ compiler.
- Caveats: only *one* declaration of the inlined function is allowed; some compilers won't inline functions with loop or complicated case expressions.

Intro to Stream I/0

PROBLEM SOLVED: extends I/O facilities for derived types.

C++ introduces a new concept for handling I/O: file streams. These are abstractions of the conventional C I/O of `stdin`, and `stdout`.

- Output

  ```
  #include <iostream.h>

  int main()
  {
  const int this_year = 1995;
      int birth_year = 1642;
      char *name = "Issac Newton";

      cout << name << " was born " << this_year-birth_year ;
  cout << " years ago in " << birth_year << ". \n";
  }
  ```

  Will print out

  ```
  Issac Newton was born 353 years ago in 1642.
  ```

  The stream `cout` denotes the usual standard output (`stdout`) device (e.g. terminal screen, or piped file). To use streams for output, do the following:

  - include the file `<iostream.h>`. this has the declarations for the basic I/O stream classes.
  - use the `<<` operator to insert data into the stream

- Input

  ```
  cout << "Number of subintervals: ";
  cin >> N;
  ```

  is equivalent to

  ```
  printf("Number of subintervals: ");
  scanf("%d", &N);
  ```

Stream I/O, cont'd.

- stream I/O biggest advantage over `printf()` and `scanf()` is that it can *extended* for user-defined types. For example,

```
ostream& operator<<( ostream &s, const Complex &x)
    {
        s << "(" << x.real << " + " << x.img << "i)";
        return s;
    }
```

creates a new way to print `Complex` numbers in C++:

```
#include <iostream.h>

Complex u,v;

u.real = 2.1;  u.img = 3.6;
v.real = 6.5;  v.img = 7.8;

cout << "The roots are " << u << " and " << v << ".\n";
```

will print out

```
The roots are (2.1 + 3.6i) and (6.5 + 7.8i).
```

Remember: complex numbers are NOT part of the C++ language. We just added them ourselves![1]

[1]Well, OK, the `complex` data type was finally added to ANSI C++, but only as a class library, not as part of the language...

C++: dynamic memory (new and delete)

PROBLEM SOLVED: safer version of `malloc()` and `free()`.

The old way to request a vector of N `double`'s:

```
double *A = (double *) malloc(sizeof(double)*N);   // allocate...
char *c = (char *) malloc(sizeof(char));

free(A);     // deallocate..
free(c);
```

The new way:

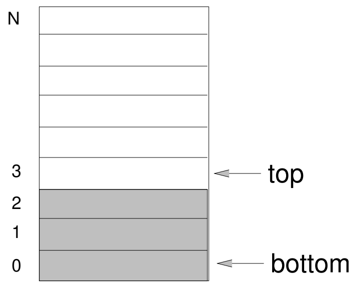
```
double *A = new double[N];      // allocate...
char *c = new char;

delete [] A;                    // deallocate...
delete c;
```

- Benefits:
  - type-safe
  - no type cast is needed
  - less error prone than `malloc()`
  - can be extended for user-defined types
  - deletion of arrays is more explicit via `[ ]`
- Caveats:
  - don't `free()` pointers that were initalized by `new`, or `delete` pointers initalized with `malloc()`.

Homework #2

1. Write an ANSI C program implementing a simple dynamic stack using pointers to the bottom and top of the stack, and the size of the stack as structure elements. (Use the "static" example stack on the WWW page as a guide, if needed.)

```
typedef struct
{
    float *bottom;
    float *top;
    int size;
} DStack;
```



2. Incorporate the `new` and `delete` features of C++ into the generation of your dynamic stack.

3. Experiment with the stream I/O functions applied to components of your stack (for example, write a StackPrint function which uses `cout` rather than printf to print out the elements in a given stack).

4. Overload the `<<` operator so that your stack can be printed with the line:

```
cout << S;
```