C++ Programming for Scientists

Roldan Pozo

Applied and Computational Mathematics Division
pozo@cam.nist.gov

Karin A. Remington

Scientific Computing Environments Division
karin@cam.nist.gov

NIST

C++ Course Outline

- **Part I: A Better C**
  - ANSI C subset
  - function overloading
  - default arguments
  - operator overloading
  - reference parameters
  - new and delete
  - I/O streams
- **Part II: "Building Classes"**
  - Classes = Data structures + Functions
  - various examples of scientific classes
  - constructors / destructors
  - explicit type conversions
  - I/O stream overloading
- **Part III: Inheritance and OO Programming**
  - case statement considered harmful
  - Inheritance: derived classes
  - when to derive
  - elegance vs. performance
- **Part IV: Advanced C++**
  - templates
  - exceptions
  - advanced I/O streams (binary files, etc.)
  - compatibility issues

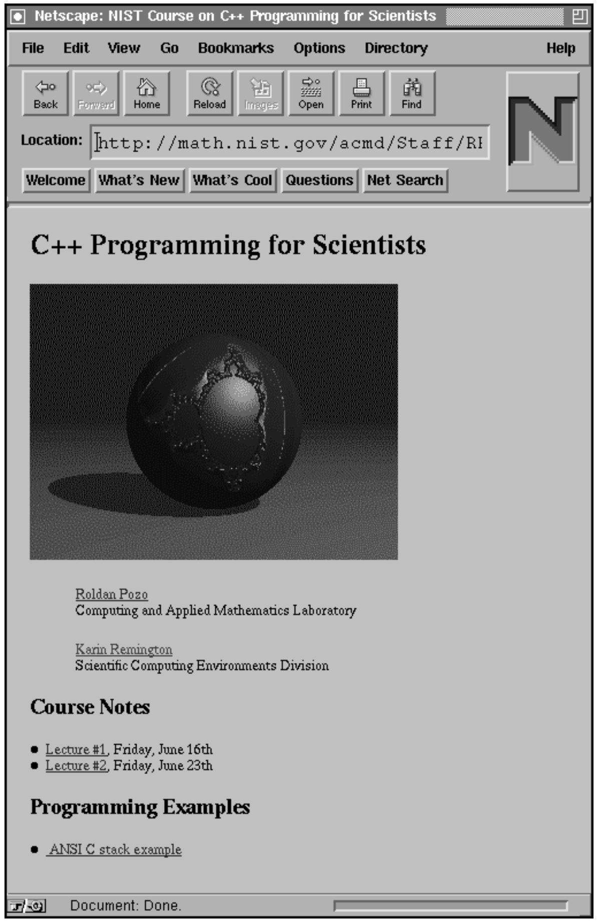
References

- **Introductory**
  - *The Complete C++ Reference*, **2nd ed., H. Schildt, 1995.**
  - *C++ Primer*, **S. Lippman, 1992.**
  - *Effective C++*, **S. Meyers, 1993.**
  - *Scientific and Engineering C++*, **J. Barton, L. Nackman, 1994**
  - *A Book on C++*, **I. Pohl, 1994.**

- **Advanced**
  - *Annotated C++ Reference Manual*, **M. A. Ellis, B. Stroustrup, 1990.**
  - *ANSI C++ Draft Standard*, **ANSI/ISO, 1995.**
  - *C++ Report*
  - *Journal of Object Oriented Programming*

C++ Course Web Page

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



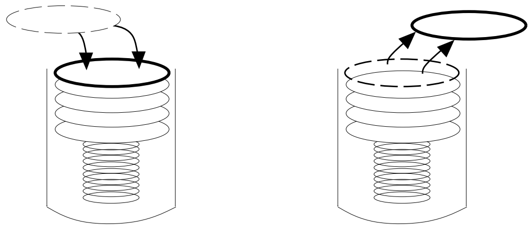
From C to C++

Features of C:

- **a small, simple language (by design).**
- **ideal for short-to-medium size programs and apps.**
- **lots of code and libraries written in it.**
- **good efficiency (a close mapping to machine architecture).**
- **pretty stable (K&R, ANSI C).**
- **designed for systems programming, not numerics.**
- **some (albeit minor) idiosyncrasies.**
- **C preprocessor (cpp) is a good, close friend.**
- **poor type-checking addressed by ANSI C.**

so, what's the problem? Why C++?

A motivating example: implementing stack data structure in C



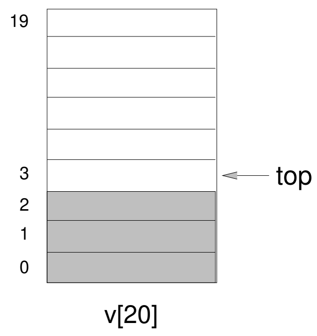
Goal: create some C code to manage a stack of numbers.

- **a *stack* is a simple first-in/last-out data structure resembling a stack of plates:**
  - **elements are removed or added only at the top**
  - **elements are added to the list via a function `push()`.**
  - **elements are removed from the stack via `pop()`.**
- **stacks occur in many software applications: from compilers and language parsing, to numerical algorithms.**
- **one of the simplest container data structures.**

sounds easy enough...

Simple Stack in C

```
typedef struct
{
    float v[20];
    int top;
} Stack;


void push(Stack *S, float val)
{
    S->v[ S->top ] = val;
    (S->top)++;
}


float pop(Stack *S)
{
    return (S->v[--(S->top)]);
}


void init(Stack *S)
{
    S->top = 0;
}


int full(Stack *S)
{
    return (S->top >= 20);
}
```



Using the Stack data structure in C programs

```
Stack S;

init(&S);                          /* initialize              */

push(&S, 2.31);                    /* push a few elements     */
push(&S, 1.19);                    /* on the stack...         */

printf("%g\n", pop(&S));           /* use return value in     */
                                   /* expressions...          */

push(&S, 6.7);
push(&S,  pop(&S) + pop(&S));      /* replace top 2 elements */
                                   /* by their sum            */
```

... so what's wrong with this?

A few gotcha's...

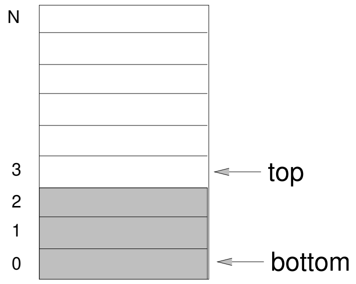
```
Stack A, B;
float x,y;

push(&A, 3.141);        /* core dump: didn't initialize A */

init(&A);

x = pop(&A);            /* error: A is empty!  */
                        /* stack is now in corrupt state; */
                        /* x's value is undefined... */

A.v[3] = 2.13;          /* don't do this! */
A.top = -42;

push(&A, 0.9);          /* OK, assuming A's state is valid.  */
push(&A, 6.1);
init(&B);
B = A;                  /* weird, but legal. */

init(&A);               /* whoops!  just wiped out A and B */


                        /* can you find the bug? */

void MyStackPrint(Stack *A)
{
    if (A->top=0)
        printf("Stack is empty.\n");
    else
        printf("Stack is non-empty.\n");
}
```

Problems with the `Stack` data structure

- **NOT VERY FLEXIBLE:**
  - fixed stack size of 20
  - fixed stack type of `float`
- **NOT VERY PORTABLE:**
  - function names like `full()` and `init()` likely to cause naming conflicts
- **but the biggest problem is it's NOT VERY SAFE:**
  - internal variables of data structure are exposed to outside world
  - their semantics are directly connected to the internal state
  - can be easily be corrupted by external programs, causing difficult-to-track bugs
  - no error handling
    - pushing a full stack
    - popping an empty stack
    - initializing a stack more than once
    - no method to determine if stack is in corrupt state
  - assignment of stacks (`A=B`) leads to reference semantics and dangerous dangling pointers.

Attempt #2: A better stack...

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




```
int DStack_init(DStack *S, int N);  /* return 0 if successful */
                                    /* 1, otherwise */

float DStack_pop(DStack *S);        /* return 0.0 if stack is */
                                    /* empty.                 */

int DStack_empty(DStack *S);        /* return 1 if empty,     */
                                    /*  0 otherwise           */
```

Improvements:

- **dynamic size (uses `malloc()`)**
- **primitive error handling**
- **function names `DStack_init()` less likely to cause naming conflicts**
- **pointer indirection results in somewhat faster access**

Still suffers from:

- **all the data corruption problems described earlier**

BIG PROBLEM:

- **old application code that used `S->v` or `S->top` no longer works!!!**

Everything you didn't want to know about errors...

Structured programming actually *impedes* the management of errors

Basically three common ways to handle them:

1. **existentialist: don't assume errors (i.e. do nothing).**
2. **bureaucratic: encode an error in return value, let someone else worry about it.**

   ```
   f(x, y, z, N, k);
   ```

   becomes

   ```
   errno = f(x, y, z, N, k);
   if (errno ==1) call MyErrorHandler1();
   else
   if (errno >= 2 && errno < 10) call MyErrorHandler2();
   ...
   ```

   - **clutters up application code.**
   - **returned error codes often cryptic.**
   - **programmers often ignore them. (How many times are `malloc()` and `fopen()` used without checking return value?)**
   - **do not "scale" well, particularly in multi-level software components. Have to be hanlded immediately, or become lost.**
3. **fascist: shut program down (i.e. `exit()`)**
   - **OK, *only* in `main()` level of application**
   - **not great for interactive applications (e.g. X apps), control applications (e.g. robotics), compilers, OS, etc.**

Attempt #3: Generic stacks via the preprocessor

```
typedef struct
{
    TYPE *bottom;
    TYPE *top;
    int size;
} GDStack;          /* Generic (?) Dynamic Stack */

int GDStack_init( GDStack *S, int N);
int GDStack_push( GDStack *S, TYPE v);
...
```

How to use in application:

1. **put all source into file GDstack.h**
2. **in application code do**

```
#define TYPE float
#include "GDstack.h"

 GDStack S;             /* a stack of floats!          */

#define TYPE int        /* oops, preprocessor warning!  */
                        /* redefinition of macro TYPE.  */

#include "GDStack.h"    /* compiler error: redefinition */
                        /* of functions!                */

GDStack S2;             /* nice try, but won't work.    */
```

- **Big problem: impossible for subprograms to tell what type a GDStack holds:**

  `???  = GDStack_pop(&S);`

- **works OK if only using *one* type of stack in *one* source file, but really not a good library solution...**

Attempt # 4 : Generic stacks via the editor...

```
typedef struct
{
    TYPE *bottom;
    TYPE *top;
    int size;
} GDStack_TYPE;

int GDStack_TYPE_init( GDStack_TYPE *S, int N);
int GDStack_TYPE_push( GDStack_TYPE *S, TYPE v);
...
```

How to use:

1. **put all source into base files GDstack.h and GDstack.c**
2. **use editor's global search & replace to convert every string of "TYPE" to "float", or "int", or whatever.**
3. **in application code do**

```
#include "GDstack_float.h"
#include "GDstack_int.h"

GDStack_float S;   /* a stack of floats!              */

GDStack_int S2;    /* finally, a stack of ints!       */

GDStack_String T;  /* oops, need to run back to editor */
                   /* and generate more files...      */
```

4. **Must link with** `GDStack_float.o`, `GDStack_int.o`, `GDStack_String.o`,..

- **Works, but *extremely* ugly...**
- **and STILL has all of the previous data corruption problems!**

Reality Check

- software is constantly being modified
  - better ways of doing things
  - bug fixes
  - algorithm improvements
  - platform changes (move from an HP to an RS/6000)
  - environment changes (new random number library)
  - customer or user has new needs and demands
- real applications are very large and complex (i.e. > 100,000 lines of code) typically involving more than one programmer
- you can never anticipate how your data structures and methods will be utilized by application programmers.
- ad-hoc solutions OK for tiny programs, but don't work for large software projects
- horror stories of incredibly simple bugs bringing large software projects to a grinding halt...
- software maintenance and development costs keep rising, and we know it's much cheaper to *reuse* rather *redevelop* code, yet we still keep recoding the same components over and over...

What have we learned from years of software development?

Software engineering points out that

- the major defect of the data-structure problem solving paradigm is the scope and visibility that the key data structures have with respect to the surrounding software system.

So, we'd like...

- DATA HIDING: the inaccessibility of the internal structure of the underlying data type.

- ENCAPSULATION: the binding of an underlying data type with the associated set of procedures and functions that can be used to manipulate the data.

$$\text{Objects} \approx \text{C structures} + \text{member functions}$$

How does C++ help solve these problems?

- **provides a mechanism for packaging C `struct` and corresponding methods together (*classes*)**
- **protects internal data structure variables from the outside world (`private` *keyword*)**
- **provides a mechanism for automatically initalizing and destroying user-defined data structures (*constructors/destructors)***
- **provides a mechanism for generalizing argument *types* in functions and data structures (*templates*)**
- **provides mechanism for gracefully handling program errors and anomalies (*exceptions*)**

Note that inheritance is *not* on this list. That comes later...

Getting Started with C++

- **You've already been writing C++ programs! (Sort of... ANSI C $\subset$ C++, but K&R C $\not\subset$ ANSI C)**
- **source files names typically end in either: .cc, .cpp, .C, .cxx, .c++.**
- **header files names typically end in either .h, .H, .hpp,**
- **some common compilers:**

g++	**Gnu (most Unix workstations)**
CC	**Sun, HP, SGI**
xlC	**IBM RS/6000**
bcc	**Borland C++ (PC)**
cl	**Microsoft C++ (PC)**
wcl	**Watcom C++ (PC)**

- **most any ANSI C program can be compiled with C++.**
- **compiling and linking similar to C, e.g.**

```
g++ -c main.cc

g++ -o main main.cc sum.cc -lm
```

ANSI C: function prototypes

- dramatically reduces argument mismatch errors. This is one of ANSI C's most useful enhancements.
- semantic type checking is performed on all functions [1] .
- Arguments and return types must match function declarations, otherwise compiler generates errors.

```
double y;
int n;
char *name = "foo";

double cos(double x);       /* this can be included    */
int strlen(char *s);        /* a separate header file  */

y = cos(2);             /* integer promoted to double. */
                        /* ... can cause problems      */
                        /* with older K&R compilers.   */

n = strlen(3.0);      /* compiler error: 3.0 cannot be */
                      /* converted to char*.           */


n = strlen(name, 3);   /* compiler error: called with  */
                       /* wrong number of arguments.   */
```

[1] With some few exceptions, like functions explicitly declared with a variable number of arguments (e.g. `printf()`).

ANSI C: `const` identifier

- determines that a variable cannot be modified. (This is ANSI C's second most useful enhancement.)

  ```
  const double h = 6.6256e-34;
  ```

  is better than

  ```
  #define h 6.6256e-34
  ```

  The first has type and scope information; can also be understood by the debugger.

- The big win, however, is in specifying that *variables* passed by address to functions are not modified.

  ```
  char* strcpy(char *s1, const char *s2);    /* modifies s1, but not s2 */
  ```

  This ensures that one can pass large structures efficiently (i.e. by address) and *safely* into external functions.

- also used to denote pointers to constants, and constant pointers:

  ```
  const double pi = 3.1415926535897932;
  const double *x = &pi;
        double  A = 1.0;
        double  B = 2.0;
        double * const dcp = &pi;
        double * dp;

  *x = 2.0;       /* error, can't change pi */

  *dcp = 2.0;     /* OK, change A to 2.0 */

  dcp = &B;       /* error, can't change dcp */

  x = &A;         /* OK, but can't modify A via x */
  ```

ANSI C: `void` and `void*` types

Used to specify generic pointers and the absence of parameters.

```
void foo(void);    /* function f requires no arguments, and  */
                   /* returns no arguments */

void *ptr;        /* a generic pointer (can point to anything) */
                  /* must be explicitly casted when used */

int    i=3;
double x=2.0;

ptr = &i;
 *((int *) ptr) = 7;       /* change the value of i */

ptr = &x;
*((double *) ptr) = 4.0;   /* change the value of x */

*ptr = 8.0;                /* syntax error. */
```

(Note: in K&R C, `char*` often played the role of `void*`. The ANSI C convention is safer.)

Differences between K&R and ANSI C

- **functions must be prototyped in ANSI C**
- **can pass structures by value in ANSI C**
- **support for enumerated types, `const` in ANSI C**
- **K&R defaults return types to `int`**

For example,

- **K&R C:**

```
#define LPP_VERSION "1.2a"

daxpy(y, a, x, N)
double a, *x, *y;
int N;
{
int i;

    for (i=0; i++; i<N)
        y[i] += a * x[i];
}
```

- **ANSI C**

```
const char *LPP_VERSION = "1.2a";

void daxpy(double *y, double a, const double *x, int N)
{
    int i;

    for (i=0; i++; i<N)
        y[i] += a * x[i];
}
```

Homework #1

1. **Locate the C++ compiler on your system.**
   (Try `man g++`, or `man CC`.)

2. **Compile and run the hello-world program.**

```
#include <stdio.h>

int main()
{
    printf("hello world.\n") ;
    return 0;
}
```

   (This is mainly to check that system include files, libraries, and linker are installed correctly.)

3. **Recode one of your last C program assignments in ANSI C. (i.e. use `const` wherever appropriate, prototype external functions, etc.) then recompile it with C++.**