C++ templates for abstract groups

John Kerl

October 9, 2005

Abstract

Many group-theoretic properties from elementary abstract algebra lend themselves to simple, easily automated algorithms for small finite groups. A basic software system which implements such algoriths is presented. As well, we attempt to tantalize the reader into further explorations into the C++ language.

Contents

1 History

The software described in this document is called SACK. It is a simple algebra calculator; the K is autoeponymous and serves to form a complete English word.

SACK was written by the author during and immediately after a senior-level undergraduate course in abstract algebra. The feature list is intentionally short: SACK is not a powerful research-level computer-algebra package (e.g. GAP xxx cite). This means that is is short and simple, its features are elementary, and its algorithms are naive.

The original design goal was to have a simple command-line program for computing in small finite groups, similar to the Unix `bc` command. That is, if one can ask one's computer for the product of the integers 10981765243 and 76452183109, then ought to just as easily as for the composition of the permuations (10 9 8)(1 7)(6 5 2)(4 3) and (7 6)(4 5)(2 1 8 3)(10 9). In particular, one should not need to wait for a monolithic computer-algebra system to start up, nor should need to overcome a steep learning curve.

Another design goal was to automate certain repetitive tasks in elementary abstract algebra, such as brute-force associativity checking.

The third design goal became apparent during development: one learns far more from writing one's own software that from using someone else's. Joseph Joubert (xxx cite) said "To teach is to learn twice." The same is true for computer programming: one's understand of a concept may be tested by one's ability to instruct a machine to work with that concept.

2 C++

The C++ language is an extension of the C language (xrefs here, including K&R). Here, the basic features needed for the current presentation are sketched.

mention "compiler"

2.1 Atomic data types

The C++ language includes a few built-in data types. These include:

- `char`: a single byte of memory, or 8 bits.
- `int`: typically, a 32-bit integer. The arithmetic is taken modulo 2^{32} with zero-centered mod, i.e. numbers from -2^{31} to $2^{31}-1$ are representable. For the purposes of the software described in this document, the integers used are small enough that they never wrap around modulo 2^{32}, and thus the arithmetic may be considered exact.
- `float`: Signed mantissa-and-exponent rational approximation to real numbers. Not used by the software described in this document.

These are used as follows:

```
    int a = 3;
    int b = 4;
    int c;

    c = a * b - 23;
```

Note that `int` determines the type; `a`, `b`, and `c` name instances of that type. The former is a cookie cutter; the latter are cookies[1].

xxx note mod and xor operators.

xxx something about array pointers, and dynamic allocation.

2.2 Classes and operator overloading

Compound data structures may be constructed using the `class` keyword. For example, a modular-arithmetic type may be defined as follows:

```
class integer_mod_t {
    int residue;
    int modulus;

    ...
public:
    integer_mod_t(int res, int mod);
    friend integer_mod_t   operator+(integer_mod_t that);
    friend integer_mod_t   operator-(integer_mod_t that);
    friend integer_mod_t   operator-(void);
    friend integer_mod_t   operator*(integer_mod_t that);
    friend integer_mod_t   operator/(integer_mod_t that);
}

integer_mod_t::integer_mod_t(int res, int mod)
{
    xxx validate modulus
    this->modulus = mod;
    this->residue = res % mod;
    xxx posmod
}
```

[1] This is a metaphor, not to be confused with Web cookies which are unrelated.

```
operator*(integer_mod_t a, integer_mod_t b)
{
    integer_mod_t c;
    if (a.residue != b.residue) {
        fatal error of some sort...
    }
    c.modulus = a.modulus;
    c.residue = (a.residue * b.residue) % c.modulus;
    return c;
}

    ...
    integer_mod_t a(8, 11);
    integer_mod_t b(9, 11);
    integer_mod_t c = a * b;
```

Note several items:

- The `_t` suffix is nothing more than a convention. This means that `integer_mod_t`, when seen elsewhere, will be recognizable on sight as being a data type.
- The `integer_mod_t` may be thought of as a compound data type. It may contain atomic data types (as in this example), elements of other compound data types, arrays of various data types, etc.
- xxx note that each element contains the modulus. This may seem like overkill (doubling the memory) and also error-prone (need to check compatibility on all ops), but experience has shown that overall system design is far more elegant as a result.
- The C++ compiler by default knows only how to do arithmetic on atomic types. The `operator` keyword allows the programmer to instruct the C++ compiler how to do arithmetic on a particular compound data type.
- The `public` keyword is a technical point of C++ which is unimportant to the current discussion; see xxx xref for information.
- The `...` warrants further discussion; the current code snippet is intended to be incomplete but instructive.
- xxx define "member", and say something about the dot and arrow.
- mention `friend` — I don't like it but it makes the examples simpler for an introductory document.

2.3 Arrays

One may define arrays of atomic or compound data types. The syntax is as follows:

```
int v[10];
int i;
for (i = 0; i < 10; i++)
    v[i] = 2*i - 3;
```

The first line of this snippet shows how to declare an array of length n (here, 10). Note that array indices are taken from 0 to $n-1$, rather than from 1 to n. (This is a frequent source of consternation for the mathematically-inclined novice.)

Note that one may also make an array of compound data types:

```
integer_mod_t v[10];
...
```

Also, a compound data type may contain array members. This will be illustrated by example in section xxx xref.

2.4 Templates

As seen, the C++ language supports arrays of arbitrary type. For example, it is just as easy to write

```
int v[10];
...
```

as it is to write

```
integer_mod_t v[10];
...
```

xxx come up with a good motivating example here ...

this is half-assed but how about it:

xxx mention name overloading

xxx mention `int *`.

```
int dot_product(int * a, int * b, int n)
{
    int c = 0;
    for (int i = 0; i < n; i++)
        c = c + a[i] * b[i];
}
```

```
float dot_product(float * a, float * b, int n)
{
    float c = 0;
    for (int i = 0; i < n; i++)
        c = c + a[i] * b[i];
}

integer_mod_t dot_product(integer_mod_t * a, integer_mod_t * b, int n)
{
    integer_mod_t c = 0;
    for (int i = 0; i < n; i++)
        c = c + a[i] * b[i];
}
```

Note that the internal details of each of these three routines is identical. What is desired is a way to give the compiler a way to compute the dot product of any data type that supports addition, multiplication, and assignment from 0.

```
template<class mytype>
mytype dot_product(mytype * a, mytype * b, int n)
{
    mytype c = 0;
    for (int i = 0; i < n; i++)
        c = c + a[i] * b[i];
}
```

The above code snippet is a **template**.

The programmer may later invoke this as follows:

```
    brand_new_type_t a[20];
    brand_new_type_t b[20];
    brand_new_type_t c = dot_product(a, b, 20);
```

This code snippet is a **template instantiation**.

3 Concrete groups

In this section, a few concrete groups are introduced. The key point is that for each concrete type, the operations idiosyncratic to that type are defined by the programmer. Then, in section 4, templates will be used to take care of group-theoretic operations which are independent of data type.

The data types to be discussed here include:

- Klein-four and unit-quaternion groups
- Cyclic groups
- Metacyclic, dihedral, and generalized-quaternion groups
- Symmetric and alternating groups
- Polyhedral symmetries as subgroups of a symmetric group

xxx note that in each case, the `*` must be used for the group operation. The reason is that the template code will always use `*` for the group operation. One would like to use, say, `group<mytype, *>` or `group<mytype, +>` but this is not supported by C++.

xxx first level of data types: elements.

3.1 $\mathcal{V}_4$ and $\mathcal{Q}_8$

For the Klein-four group, the data strucure is as follows:

```
class v4_t {
    int code; // From 0 to 3
    ...
    friend v4_t operator*(v4_t a, v4_t b);
}
```

The operation is as follows:

```
static int V4_cayley_table[4][4] = {
    /*        e  a  b  c */
    /* e */ { 0, 1, 2, 3 },
    /* a */ { 1, 0, 3, 2 },
    /* b */ { 2, 3, 0, 1 },
    /* c */ { 3, 2, 1, 0 },
};

v4_t operator*(v4_t a, v4_t b)
{
    v4_t c;
    c.code = V4_cayley_table[a.code[b.code]];
}
```

xxx elucidate the double brackets: (*) code from 0-3 (checking omitted); (*) find row of table; (*) find column.

The unit quaternions are similar; here too, a table is used to define the arithmetic operation.

3.2 Cyclic, metacyclic, dihedral, and generalized-quaternion groups

Cyclic groups of order n are idential to `integer_mod_t`, except that their multiplication operation implements what we would normally think of as addition modulo n.

Metacyclic and generalized-quaternion groups follow the same plan as the dihedral groups; only the latter will be detailed.

The dihedral group $\mathcal{D}_n$ is here taken to be the symmetry group on a plane n-gon. It has order $2n$ and is given by the following presentation:

$$\mathcal{D}_n = \langle \rho, \phi \mid \rho^n = \phi^2 = 1, \phi\rho = \rho^{n-1}\phi \rangle.$$

From the presentation, the element ρ (*rho* for *rotate*) has order n; the element ϕ (*phi* for *flip*) has order 2. Furthermore, repeated use of the final relation enables any element of $\mathcal{D}_n$ to be put into the form $\rho^i\phi^j$ for $i = 0, 1, 2, \ldots, n-1$ and $j = 0, 1$. Thus, given two elements $\rho^i\phi^j$ and $\rho^k\phi^\ell$ of $\mathcal{D}_n$, we obtain the product

$$\begin{aligned} j = 0: \qquad & \rho^i\rho^k\phi^\ell && = \rho^{i+k}\phi^\ell \\ j = 1: \qquad & \rho^i\phi\rho^k\phi^\ell && = \rho^{i-k}\phi^{\ell+1} \end{aligned}$$

The dihedral data type may then be defined as follows:

```
class dih_t {
    int rot;
    int flip;
    int n;
    ...
```

Note that as in section xxx, the modulus n is carried around in each element of the group. Then the group operation may be implemented as follows:

```
dih_t::operator*(dih_t a, dih_t b)
{
    dih_t c;

    if (a.n != b.n) {
        fatal error ...
    }

    if (a.flip)
        c.rot = a.n + a.rot - b.rot;
    else
        c.rot = a.rot + b.rot;
```

```
    if (c.rot >= a.n) // Reduce mod n
        c.rot -= a.n;
    c.flip = a.flip ^ b.flip;
    c.n = a.n;
    return c;
}
```

xxx note avoidance of the mod operator, as well as the use of the XOR operator. ++ and – operator.

3.3 $\mathcal{S}_n$, $\mathcal{A}_n$, and polyhedral symmetries

Although we often write elements of $\mathcal{S}_n$ in cycle notation, the image-map format is handier for a software implementation. (Nonetheless, I/O routines may be implemented to handle cycle notation — the point is that the internal storage format uses image maps.) That is, we can write, for example,

$$\sigma = (1234)(567) = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 4 & 1 & 6 & 7 & 5 \end{pmatrix}.$$

In the latter form, 1 maps to 2, 2 maps to 3, 5 maps to 6, etc. Note that the top row of the image map always consists of the numbers 1 through n, and thus may be omitted.

A permuation data type may then be implemented as follows:

```
class pmt_t {
    int   n;
    int * images;
    ...
};
```

where the `images` array holds the bottom row of the image map for a given permutation. Using image maps, multiplication (i.e. composition) of permuations is elegant:

```
pmt_t pmt_t::operator* (pmt_t that)
{
    if (a.n != b.n) {
        fatal error ...
    }
    pmt_t c(a.n);
    for (int i = 0; i < a.n; i++)
        c.images[i] = a.images[b.images[i]];
    return c;
}
```

xxx parity operation; count swaps on qsort. later.

Polyhedral symmetries as subgroups of a symmetric group.

xxx note dihedral groups (symmetries of plane polygons) could have been done this way too. In fact, *all* these SFGs could be done this way. Point: Friendly I/O.

3.4 Other routines

xxx I/O not discussed. Show some sample ASCII stuff though.

op `==`, op `!=`, op `>`.

3.5 Fill routines

previous was type stuff. This is group stuff. Separate source file.

xxx second level of data types.

list-all; from-file

4 Algorithms for abstract groups

snippet of instantiation code x 2: header and source

xxx third level of data types. This one is abstract.

4.1 Data structures

xxx array, group

xxx note currently `tarray<element_type>` is it. No `group_t<element_type>`. Doesn't readily lend itself to quotients. But that's where I'm at.

4.2 Group axioms

Now, it is easy to check whether an array of elements is in fact a group:

```
template<class element_type>
int group_is_group(
    tarray<element_type> & G)
{
    if (!group_is_closed(G))
```

```
        return 0;
    if (!group_is_associative(G))
        return 0;
    if (!group_has_identity(G))
        return 0;
    element_type e;
    if (!group_search_for_identity(G, e))
        return 0;
    if (!group_has_inverses(G, e))
        return 0;
    return 1;
}
```

Here is the associativity-checking routine; the others are similar. As well, the is-abelian routine follows the same idea.

```
// ------------------------------------------------------------
template<class element_type>
int group_is_associative(
    tarray<element_type> & G)
{
    int n = G.get_num_elements();
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                if ((G[i] * (G[j]) * G[k]) != (G[i] * (G[j]) * G[k]))
                    return 0;
    return 1;
}
```

Several simple algorithms now follow.

4.3 Exponentiation

Given the group's * operation, it is easy to raise an element to any integer power.

4.4 Inversion

Once a set is known to be a group (here, a small finite group), it is just as easy to invert an element x: simply obtain the number n of elements of the group, then compute $x^{-1} = x^{n-1}$.

```
// Use this if you don't know the set is a group.
```

```
template<class element_type>
int group_element_search_for_inverse(
    tarray<element_type> & G,
    element_type   a,
    element_type   e,
    element_type & rainv)
{
    int n = G.get_num_elements();
    for (int i = 0; i < n; i++) {
        element_type b = G[i];
        if (((a * b) == e) && ((b * a) == e)) {
            rainv = b;
            return 1;
        }
    }
    return 0;
}

// Use this if you know the set is a group.
template<class element_type>
element_type group_element_inverse(
    element_type x,
    int group_order)
{
    return group_element_exp(x, group_order - 1, group_order);
}
```

4.5 Finding the identity

Likewise, one need not mark the identity element of a group in any special way. Rather, when the identity is needed, one may take any element of the group and raise it to the nth power.

```
// Call this if you don't know the set is a group.
template<class element_type>
int group_search_for_identity(
    tarray<element_type> & G,
    element_type & re)
{
    int n = G.get_num_elements();
    for (int i = 0; i < n; i++) {
        element_type e = G[i];
        int all_equal = 1;
        for (int j = 0; j < n; j++) {
            element_type x = G[j];
```

```
            if ((e * x != x) || (x * e != x)) {
                all_equal = 0;
                break;
            }
        }
        if (all_equal) {
            re = e;
            return 1;
        }
    }
    return 0;
}

// Call this if you know the set is a group.
template<class element_type>
element_type group_get_identity(
    tarray<element_type> & G)
{
    int n = G.get_num_elements();
    return group_element_exp(G[0], n, n);
}
```

4.6 Orders

Group order: Once a set is known to be a group, simply take the number of the elements in the set.

Element orders: xxx

```
template<class element_type>
int group_element_order(
    element_type x,
    element_type e,
    int group_order)
{
    for (int i = 1; i <= group_order; i++) {
        if (!divides(i, group_order))
            continue;
        element_type xi = group_element_exp(x, i, group_order);
        if (xi == e)
            return i;
    }
    std::cerr << "group_element_order:  No order found.\n";
    exit(1);
```

```
}
```

Maximal element order: For each element x of the group G, compute the element order of x. Take the maximum of all these.

Cyclicity: Compute the maximal element order. Test it for equality against the group order.

4.7 Cayley table

List of elements: For each element of the set, invoke the I/O routine. Using templates, this results in the specific data type's output routine being invoked.

Cayley table:

```
template<class element_type>
int group_print_cayley_table(
    tarray<element_type> & G,
    std::ostream & os)
{
    int n = G.get_num_elements();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            element_type Gij = G[i] * G[j];
            if (j > 0)
                os << " ";
            os << Gij;
        }
        os << "\n";
    }
    return 1;
}
```

4.8 To do

Torsion

center/centralizer

close

cayley_sn

subgroup, normal_subgroup, normalizer, core

nilpotent, solvable

commsgr, ascendant, descendant

aut group

5 More information

xxx ref for more about C++ in general; mathematical C++ in particular

certainly xref to GAP/MeatAxe/GP/Maple/Magma

References

[**DF**] D.S. Dummit and R.M. Foote. *Abstract Algebra* (2nd ed.). John Wiley and Sons, 1999.

[**Gro**] L.C. Grove. *Algebra* Dover, 1983?.