Interfacing BOS to C++

Benno List

Computer and Software Meeting

11.6.97

- General Considerations
- Bank Access with C++
- Conclusions

ps-file: `http://www-h1.desy.de/~blist/c++/transp.cas.ps`

example page: `http://www-h1.desy.de/~blist/c++/`

Why C++?

- **It's strongly typed.**
  You can tell the compiler what you want to do
  $\Rightarrow$ The compiler can (sometimes) tell you when you are doing something else.
- **It faciliates modularization and encapsulation.**
  You don't have to invent long and silly names that also fit into 8 characters to avoid name clashes.
- **It has a very nice syntax.**
  You'll never have to sum the components of a 4–vector by hand again.
- **It comes with an extremely powerful library.**
  You'll never have to implement lists or a sort algorithm again yourself.
- **Motivation.**
  FORTRAN programming is dull. Students (we all) want to use a modern programming language.
- **Jobs.**
  Diploma and PhD students need jobs after their work in HEP. Being an experienced FORTRAN programmer is a great way not to get one.
- **C++ is the future.**
  At least it's closer to the future than FORTRAN.
- (And, of course, there's `root`.)

Is Interfacing BOS to C++ the Right Way?

NO.

The data model is at the heart of object oriented programming and design. To start with a FORTRAN–like data model means to throw away many of the advantages C++ has to offer.

BUT...

BUT

- **A thorough approach takes years.**
  Diploma and PhD students don't have years.
- **A thorough approach needs proficient C++ users.**
  We don't have them. We have to train some to get them.
- **Banks are not such a bad data model.**
  They just contain what people think is needed (and available) for analysis.
- **Tens of (wo)manyears have been invested in existing H1 software.**
  Nobody will re–write H1SIM and H1REC in C++. This leaves us with lots of BOS and FPACK–based software.
- **Structured bank access is better than unstructured access.**
  Lots of M–Jobs fail every day because of wrong use of pointers to banks, confusing `IBTAB` and `RBTAB`, by circumvention of miniheaders, and so on. In the past this has cost months of (re)processing time.

Remember:

H1 software is written by amateurs, and used by amateurs.

Some of them (us) may be gifted amateurs, but still amateurs.

By the way

A really radical, thorough approach to object oriented design would probably not even use C++, but a really dyed–in–the–wool OOP language like Java.

C++ was designed to allow incremental transition from old–fashioned (C) to modern (OO) programming.

We have not even managed the transition to C so far...

What Bjarne S. Thinks About It

Introducing new ways of doing things into an organization can be painful. [...] In particular, an abrupt change that overnight turns productive and proficient members of "the old school" into ineffective novices in "the new school" is typically unacceptable. However, it is rare to achieve major gains without changes, and significant changes typically involve risks.

C++ was designed to minimize such risks by allowing a gradual adoption of techniques. Although it is clear that the largest benefits from using C++ are achieved through data abstraction, object–oriented programming, and object–oriented design, it is not clear that the fastest way to achieve these gains is a radical break with the past. [...]

The idea of learning object–oriented design, object–oriented programming, and C++ in stages is supported by facilities for mixing C++ code with code written in languages that do not support C++'s notion of data abstraction and object–oriented programming. [...]

This allows designs for C++ to evolve to use data abstraction and class hierarchies in the presence of code written in languages in which these concepts are missing and even under the constraint that the resulting code must be callable from procedural languages.

Bjarne Stroustrup,
The C++ programming language (3rd ed.), pp. 718f.

What's Coming Now?

This talk is about ideas.

I want to show how one could access BOS banks from C++.

The emphasis is on syntax.

I have implemented these things just to show that it's possible, and to gain experince by using the stuff in toy analysis jobs.

Many issues are not adressed yet:

- Multiple BOS commons
- Work banks
- Garbage collection
- Banks that have changed their format (DMIS)

  V.B.: "Never change a bank format!"

Let's Start

For the code, look at
`http://www-h1.desy.de/~blist/c++/`.

A Bank DDL is very close to the definition of a structure.

⇒ Use structures (or classes) to tell C++ what a bank contains.

Example: A steering bank for a track selection.

A Steering Bank for a Track Selection

DDL:

```
!       BANKname BANKtype      ! Comments
!
 BANK   QTRA                   ! Steering parameters for
!                                track selection
!   ATTributes:
!   -----------
!COL ATT-name FMT Min    Max    ! Comments
!
! Conditions for good central track:
  1  TH_MIN_C  F   0.    180.  ! min. theta [degrees]
  2  TH_MAX_C  F   0.    180.  ! max. theta [degrees]
  3  RS_MAX_C  F   0.    200.  ! max. start radius [cm]
  4  RE_MIN_C  F   0.    200.  ! min. end radius [cm]
  5  RL_MIN_C  F   0.    200.  ! min. radial length [cm]
  6  PT_MIN_C  F   0.    INF   ! min. p_T [GeV]
! and much more...
 END BANK
```

This can be translated directly into a structure in C++:

```
struct qtrarow {
// Conditions for good central track:
  float th_min_c;  // min. theta [degrees]
  float th_max_c;  // max. theta [degrees]
  float rs_max_c;  // max. start radius [cm]
  float re_min_c;  // min. end radius [cm]
  float rl_min_c;  // min. radial length [cm]
  float pt_min_c;  // min. p_T [GeV]
// and much more...
}
```

Using the Bank

- Open it:

```
StrBank<qtrarow> qtra ("QTRA", 0);
```

  This opens the bank, i.e. performs `nlink ("QTRA", 0)`. `qtra` now represents the bank. Only the bank index is stored in `qtra`.

- Access it:

```
float thet_min_cen = qtra.th_min_c;
```

  So `qtra` acts like a structure of type `qtrarow` itself.

- Or copy its contents completely:

```
qtrarow cuts = *qtra;
```

  i.e. `qtra` can also be viewed as pointer to a `qtrarow`.

- If the bank is not on the BOS common, these accesses would lead to exceptions. Two functions (methods) are, however, always allowed:

```
int open   = qtra.is_open();  // 1 if bank was found,
                              // 0 else
int length = qtra.length();   // length if bank was,
                              // found 0 else
```

How is that done?

Class Bank: Base class for all bank accesses.

```
class Bank {
protected:
  int base;                                // only index is
                                           // stored
public:
  inline Bank (char name [4], int no=0)   // constructor
    { base = nlink (name, no); }           // gets bank index
  inline ~Bank () {}

  inline int length() const
    { return base ?  bcs.iw [base-1] : 0; }
  inline int is_open() const
    { return base > 0; }
  inline char *name() const
    { return base > 0 ? (char*)bcs.iw [base-4] : 0; }
  inline int number() const
    { return base > 0 ? bcs.iw [base-3] : -1; }

  inline int i (int n) const {             // for access to
    if (base) {                            // integers
      if (n > 0 && n <= length())
        return bcs.iw [base-1+n];
      else
        throw BankIndexWrong (name(), number(), n);
   } else {
      throw BankNotOpen();
   }
  }
  inline float r (int n) const {            // for access to
    if (base) {                             // reals
      if (n > 0 && n <= length())
        return bcs.rw [base-1+n];
      else
        throw BankIndexWrong (name(), number(), n);
    } else {
      throw BankNotOpen();
    }
  }
};
```

Class StrBank

For "structured" banks, a template StrBank has been created:

```
template <class row>
class StrBank: public Bank {

// No additional information is stored compared to Bank

public:
  inline StrBank (char name [4], int no=0): Bank (name, no) {
//   Check that bank is at least as long as its structure demands
    if (base && length() < sizeof(row) / sizeof (int)) {
      throw StrBankLengthError(name, no);
    }
  }

  inline ~StrBank () {}

//   Overloading the dereferencing operator "->" allows
//   "bank.member" notation

  row *operator-> () const {
    return base ? (row*)(bcs.iw+base) : 0;
  }

//   Overloading the dereferencing operator "*" allows
//   to copy complete bank contents

  row& operator* () const {
    if (base)
      return *(row*)(bcs.iw+base);
    else
      throw BankNotOpen();
  }
};
```

Constructors and StrBank

A constructor can be used to define default values for steering banks:

```
struct qtrarow {
// Conditions for good central track:
  float th_min_c;  // min. theta [degrees]
  float th_max_c;  // max. theta [degrees]
  float rs_max_c;  // max. start radius [cm]
  float re_min_c;  // min. end radius [cm]
  float rl_min_c;  // min. radial length [cm]
  float pt_min_c;  // min. p_T [GeV]
// and much more...

// Constructor defines default values;
// zero default values can be omitted.
  qtrarow () {
    th_min_c = 20;
    th_max_c = 160;
    rs_max_c = 45;
    rl_min_c = 10;
    pt_min_c = 0.15;
  }
}

void trksel() {
  static int first = 1;
  static qtrarow cuts();      // "cuts" is initialized
                              // to default values
  if (first) {
    first = 0;
    try {
      StrBank<qtrarow> qtra ("QTRA");
      if (qtra.is_open())      // Bank is there,
        cuts = *qtra;          // get cuts from bank
    }
    catch (StrBankLengthError) // Steering bank is too short,
    { }                        // use default values
  }
// Now do the real track selection...
}
```

Next Step: Tables

DDL of the DTRA bank:

```
!       BANKname BANKtype      ! Comments
!
  TABLE  DTRA                   ! DST vertex fitted tracks
!    ATTributes:
!    -----------
! COL ATT-name FMT Min    Max   ! Comments
!
   1  PTINV_TR F    0.   200.   ! 1./pt (GeV**-1) ; B-Field correction
                                ! is already applied for this quantity
   2  PHI_TR   F  -PI     +PI   ! phi angle at vertex
   3  THET_TR  F    0.    +PI   ! theta angle at vertex
!     and much more...
  END TABLE
```

And the C++ definition of one row:

```
class dtrarow {
public:
  float ptinv_tr;
  float phi_tr;
  float thet_tr;
// and much more...

// plus access functions:
  inline float getpt () {return 1/ptinv_tr;}
  inline float getp ()  {return 1/(sin(thet_tr)*ptinv_tr);}
// and much more...
}
```

Such a translation could be done automatically, e. g. by an awk script!

Usage of Tables

```
void trksel() {

// initialization stuff you saw already...

// here comes the bank access:

  Table<dtrarow> dtra ("DTRA", 0);
  for (int i = 1; i <= dtra.rows(); i++) { // rows() returns 0 if
                                           // bank was not found
    float   theta =  dtra[i].thet_tr;      // get theta
    float   pt =     dtra[i].getpt();      // get p_t
    dtrarow &track = dtra[i];              // get reference to
                                           // whole track
    float   p  =     track.getp();         // get p of track
  }
```

The overloaded subscripting operator [] allows access to rows as if the bank were an array of rows.

The miniheader is always used to locate the position of a row in the bank.

When the table is opened, the consistency of the miniheader is checked, and it is made sure that the number of columns is at least as big as the number of elements of one row.

Implementation of `Table`

```
template <class row>
class Table:
public Bank {

protected:
  inline int rows_() const {return bcs.iw [base+1];}
  inline int columns_() const {return bcs.iw [base];}

public:
  inline Table (char name [4], int no=0): Bank (name, no) {
//  Check consistency of miniheader
    int c = columns();
    if (base && length() != rows()*c + 2)
      throw TableStructureError (name, no, c, rows(), length());
//  Check number of columns
    if (base && c < sizeof (row)/sizeof(int))
      throw TableWidthError (name, no, c, sizeof(row)/sizeof(int));
  }

  inline ~Table () {}

  inline int rows() const {return base ?  rows_() : 0;}
  inline int columns() const {return base ?  columns_() : 0;}

  row& operator[] (int i) const;
};

template <class row>
row& Table<row>::operator[] (int i) const {
  if (base) {
    if (i >= 1 && i <= rows())
      return *(row *)(bcs.iw+base+2+(i-1)*columns_());
    else
      throw BankIndexWrong (name(), number(), i);
  } else {
    throw BankNotOpen();
  }
}
```

And now: Relations

A relation in a table is just a rownumber of another bank, i. e. an integer, albeit with a special meaning.

One obvious implementation of relations is this:

```
class dtrarow {
public:
  float ptinv_tr;
  float phi_tr;
  float thet_tr;
// and much more...
  int l_ver_tr;    // Link to vertex at begin of track:  L_VER_TR
  int l_tcn_tr;    // Link to DTNV tracks central L_TNC_TR
  int l_tnf_tr;    // Link to DTNV tracks forward L_TNF_TR
  int l_nxn_tr;    // Link to next fit hypothesis L_NXT_TR

// plus access functions, not shown here...
// and much more...
}
```

Usage would be (definition of `dverrow` is left out):

```
  int     link_ver = dtra[i].l_ver_tr;                // link to vertex
  dverrow vertex   = dver[dtra[i].l_ver_tr];          // vertex
  int     ver_typ  = dver[dtra[i].l_ver_tr].ityp_ve; // vertex type
```

Possible (and common) bugs:

```
  float   ver_x    = dver[dtra[i].l_tcn_tr].x_ve;    // oops!
//                              ^^^^^^^^ should be l_ver_tr.
//                    Caught at runtime only if l_tcn_tr > dver.rows().

  float   ver_y    = dtnv[dtra[i].l_ver_tr].y_ve;    // oops!
//                    ^^^^ should be dver.
//                    Caught by compiler: dtnv has no y_ve member
```

Better: Smart Pointers

A template `TablePointer` allows this syntax:

```
class dtrarow {
public:
  float ptinv_tr;
  float phi_tr;
  float thet_tr;
// and much more...
  TablePointer<dverrow> l_ver_tr; // Link to vertex at begin of track:  L_VER_TR
  TablePointer<dtnvrow> l_tcn_tr; // Link to DTNV tracks central L_TNC_TR
  TablePointer<dtnvrow> l_tnf_tr; // Link to DTNV tracks forward L_TNF_TR
  TablePointer<dtrarow> l_nxn_tr; // Link to next fit hypothesis L_NXT_TR

// plus access functions, not shown here...
// and much more...
}
```

Objects of type `TablePointer` contain just one integer, and can be used like integers, i. e. all examples of the previous page remain valid. But we can now have

```
  int     link_ver = dtra[i].l_ver_tr;          // link to vertex, auto-
                                                // matic type conversion
  dverrow vertex   = *dtra[i].l_ver_tr;         // vertex
  int     ver_typ  = dtra[i].l_ver_tr->ityp_ve; // vertex type

  float   ver_x    = dtra[i]->l_tcn_tr.x_ve;    // oops!
//                             ^^^^^^^^ should be l_ver_tr.
//             Caught at compile time because l_tcn_tr points to DTNV
```

This use of pointers involves some overhead, because an `nlink` is performed every time the link is used.

The template has to know to which bank a pointer points. Therefore one needs an initialization:

```
char TablePointer<dverrow>::name [5] = "DVER";
int  TablePointer<dverrow>::no       = 0;
```

This can be done in a separate file.

Implementation of TablePointer

```
template <class row>
class TablePointer {

protected:
  static char name [5];     // Needs to be initialized!
  static int no;            // Needs to be initialized!

public:
  int rownumber;

  TablePointer (int i=0) {rownumber = i;}

  operator int () const {return rownumber;} // defines cast to int
  row& operator* () const;
  row *operator-> () const;
};

template <class row>
row &TablePointer<row>::operator* () const {
  if (rownumber >= 1) {
    Table<row> T (name, no);
    return T [rownumber];
  } else {
    throw BankIndexWrong (name, no, rownumber);
  }
}

template <class row>
row *TablePointer<row>::operator-> () const {
  if (rownumber >= 1) {
    Table<row> T (name, no);
    if (T.is_open())
      return &T [rownumber];
    else
      throw BankIndexWrong (name, no, rownumber);
  } else {
    throw BankIndexWrong (name, no, rownumber);
  }
}
```

This Is a Prototype

At the start of an ambitious development project, we do not know the best way to structure the system. [...] How—short of building the complete system—do we get the information necessary to understand what design decisions are significant and to estimate their ramifications?

We conduct experiments. [...]

Prototyping is a way of experimenting. The desired results from building a prototype are the insights that building it brings, not the prototype itself. [...]

After use, a prototype should be thrown away.

Bjarne Stroustrup,
The C++ programming language (3rd ed.), pp. 710f.

Conclusions

The software presented here is just a first try to make BOS data available to C++ programming.

Many things could be improved, or done differently. Maybe one should start at a completely different point, e. g. build an FPACK–to–C++ interface that fills C++ objects directly from FPACK records.

The scheme presented here allows to have an access to BOS banks which is

- structured,
- range–checked, but still
- fast, and
- avoids data integrity problems which would arise from copying data.

It allows to transfer existing FORTRAN programs into C++ in a reasonable time, which could make them much more reliable and better maintenable.

One last word:
I'll leave DESY in 10 days.
If anybody likes to continue this business, (s)he is cordially invited!