

Re: how to store list of varying types

Source: <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2008-06/msg01271.html>

- *From:* Joseph M. Newcomer <newcomer@xxxxxxxxxxxx>
 - *Date:* Mon, 30 Jun 2008 17:03:33 -0400
-

When there's a variable-length string, you can do it as

```
typedef struct {
    DWORD length;
    DWORD offset;
} StringValue;

struct {
    StringValue data;
    StringValue parameter;

    DWORD thing;
    DWORD somethingelse;
    StringValue whatever;
    BYTE strings[1];
};
```

So you put all the data in the front, where it has fixed lengths, and all the variable-length data follows, starting at strings[0];

then your strings all follow; they may or may not be NUL-terminated, your choice. This is another common representation.

joe

On Mon, 30 Jun 2008 13:05:38 -0700, "Nick Schultz" <nick.schultz@xxxxxxxx> wrote:

There are some messages that contain variable length strings. How do I represent that with those in the struct definitions?

I suppose I could just give the message struct a pointer to a CString and send out two messages. One being the message struct with the its CString pointer null, and the second one the CString object. The front end will then have to finish constructing the packet by copying the two data objects into its local heap and then set the message struct's CString pointer to CString object's local heap location.

this should work right? Is there an easier solution to this problem? I wouldn't want to put a whole String in the struct's union declaration, since

Re: how to store list of varying types

that would make the smallest packet(3 bytes) take up the same size of a maxed out string-bearing packet (~255 bytes)

Thanks

Nick

"Joseph M. Newcomer" <newcomer@xxxxxxxxxxxx> wrote in message news:nm6i64dq4uqo1lg2707k3gj2gejqbt47u8@xxxxxxxxxx

See below...

On Mon, 30 Jun 2008 09:50:34 -0700, "Nick Schultz"

<nick.schultz@xxxxxxxx>

wrote:

I need to pass a class that will contain about 30 to 100 bytes of information. The class also has 2 vectors, one that holds the raw packets (CAN bus supports 8 byte max packets) that make up the protocol packet , and the other vector holds descriptions of the data fields in the payload (byte position, length, name). Original implementation had the vectors storing pointers to the objects, however since we're passing data between processes, those pointers won't be valid to the receiving process, correct?

Correct. What you could do is store it as a data structure

```
<packettype> <totallength> <rawbytetype>  
<totallength><rawbyte0>...<rawbyten>  
<type0><offset0><type1offset1>...<typen><offsetn>
```

Just send these out. There are some interesting questions, such as why you need the data prepared (couldn't each receiver just call a parsing subroutine?); you could just send the raw data out. Since the packet type implicitly indicates what the offsets would be, you could just define a union member

```
typedef struct {  
    WORD count;
```

Re: how to store list of varying types

```
BYTE flags; } TYPE0;
typedef struct {
    DWORD count;
    WORD thing;
    BYTE flags;
    WORD whatever; } TYPE1;
```

```
typedef struct {
    BYTE header;
    union {
        TYPE0 type0;
        TYPE1 type1;
        ...
        TYPEn typen;
    } t;
} data;
```

then, when you receive the packet, you just map the data union onto it, e.g.,

```
LRESULT CMyWhatever::OnCopyData(WPARAM wParam, LPARAM lParam)
{
    data * stuff = (data *)...address of byte in data packet...;
    switch(stuff->header)
    {
        case TYPE0:
            HandleType0(&stuff->t.type0);
            break;
        case TYPE1:
            HandleType1(&stuff->t.type1);
            break;
        ...
        default: // unknown type
            return 0;
    }
}
```

```
void CMyWhatever::HandleType0(TYPE0 * info)
{
    ...do stuff
}
```

```
void CMyWhatever::HandleType1(TYPE1 * info)
{
    ...do stuff
}
```

etc. I used neutral names like type0, type1, etc. but for one of my embedded systems the types might have been

Re: how to store list of varying types

RAWDATAPACKET, CONTROLPACKET, VALUEPACKET,
TIMERPACKET, SWITCHPACKET, and
similar
meaningful names.

I don't know what your specific protocol is, but in the embedded protocols
I've worked in,
there is ALWAYS a structure, so there should be no reason to create a
vector that
represents pieces that are "preparsed". You know the type, which tells
you the structure,
and you interpret the data relative to that structure, nothing else fancy
required.

etc.

Essentially, you are talking about trivial amounts of data, so the notion
of copying
becomes irrelevant for performance.

Also, I was told our systems use approximately 40% of the 1
mbit/s bus
speed. According to the protocol, there are some (small)
messages that
are
anticipated to be issued 500–800Hz. others range from
200–267 Hz and
some
1 to 60 Hz.

This is my first real world, nontrivial application (fresh out
of
college),
so I don't really have a feel where or when optimizations.
Thanks for
your
help!

800 messages/sec is about 1.25ms/msg. Since an 2.8GHz x86 can peak out at
6instructions/ns, you have time to issue over 7 million instructions
between each message.
That's a lot of headroom.

But note that Windows makes no pretensions about being a realtime system.
With a
messaging system vastly less efficient than the one you are proposing, I
could handle

Re: how to store list of varying types

1400messages/sec each of which involved complex processing

Since this is an early project, one question is: is your background Unix/linux? It is a natural decomposition in Unix/linux to think of "processes", but have you simply considered "threads"?

Also, this backend, "routing" process should be running at all times.

Would making it a windows service be an appropriate solution? Are there any precautions I should take?

It might not be a good idea. A Windows service cannot easily communicate with applications. You would have to export named pipes from the service and send data out the named pipes to get the information distributed. That's the only effective way to communicate with a service. It cannot use SendMessage/PostMessage to applications running as the logged-in user.

I would suggest the on-the-fly pipe allocation mechanism where a new pipe is created each time a connection is established.

joe

Thanks

Nick

"Joseph M. Newcomer" <newcomer@xxxxxxxxxxxx> wrote in message news:p39b6492okq1gmg1t6d8u3t6e0n1hqom7q@xxxxxxxxxxx

One way to handle this is to create a "router process" that handles all communication. A process that wishes to receive messages posts a message to the router process that tells what kind of messages it wants to receive. When the router process

Re: how to store list of varying types

receives a message of type "A" it passes a copy of it to all the registered processes. For example, it could use PostMessage if the content is small (two pointer-sized values), or it could sequentially send WM_COPYDATA to each process. Or, because SendMessage is synchronous, you could consider starting a new thread for each process, creating a UI thread. The main data thread will do a PostThreadMessage to each thread based on the desired registry of elements, and each thread does a dequeue-and-SendMessage(WM_COPYDATA) of the data.

It makes no sense whatsoever to consider shared_ptr in this context because there is nothing to share, or share it with. How big are your packets, for example? I'd just copy the entire packet, and not worry about overheads of making a copy. This would be a pointless waste of time most of the time. How long does it take to copy 20 bytes? MEASURE it. Use the high-resolution timer. How many tens of nanoseconds does it take?

It is a common error to try to optimize code that never required optimization.

Example: I have a system that uses PostMessage for interprocess communication. A string is sent by putting the connection id and a byte count in WPARAM, and 0 to 4 bytes of text

Re: how to store list of varying types

in LPARAM. Typical messages were 20 to 100 bytes, so it could take 6–21 messages to pass it (a message with a 0 byte count was the "end of message" terminator).

This was a quick hack to get the program running. However, some years later, we had a client that required "400 messages per minute" performance. This was the

Moment of Truth:

I was going to have to rewrite this interface. But FIRST, I decided to measure it. I cranked up the input data generator on four machines all connected with 100–base–T Ethernet. I peaked out at 1400 messages/minute. So efficiency didn't matter; I had beat the desired goal by better than a factor of 3. That's good enough.

Premature optimization is usually a mistake. In the absence of performance data, attempts at optimization are usually misdirected, resulting in overly complex code that is harder to create, debug, and maintain than the simple code, but which has no noticeable impact on the performance.
joe

On Fri, 27 Jun 2008 15:30:28 –0700, "Nick Schultz"
<nick.schultz@xxxxxxx>
wrote:

Hmm...

What would you recommend of a way of sending multiple copies of the same packet from one process to

Re: how to store list of varying types

potentially multiple
processes? Also keep in
mind
that not every process will
always receive every
packet, for example
process 1 & 2 only care
about packet-type A and
process 2 & 3 only care
about packet-type B

What I want is a backend
process (perhaps a service)
that manages a
connection to the bus,
performs protocol parsing,
etc.

Applications will hook into
the backend by registering
and requesting
what
type of messages it wants to
receive. The backend then
uses filters to
distribute packets to the
applications. Original intent
was to use
shared_ptrs to the packet
objects so we don't have to
waste memory and
time
copying multiple objects,
however it now sounds like
that is not an
option...

Thanks Joe for the input,

Nick

"Joseph M. Newcomer"
<newcomer@xxxxxxxxxxxx>
wrote in message
news:4jma64poa58k345o72igegcm6sbrarokcl@xxxxxxxxxx

This will
work in all
kinds of

Re: how to store list of varying types

contexts,
but not for
multiple
applications.
joe

On Fri, 27
Jun 2008
11:51:07
-0700,
"Nick
Schultz"
<nick.schultz@xxxxxxx>
wrote:

The
main
use
for
this
application
is
that
there
can
be
multiple
applications
interested
in
the
same
packet.
instead
of
making
multiple
copies
the
same
packet,
I
can
just
create
multiple
shared_ptrs
that
point
to

Re: how to store list of varying types

one
packet,
and
when
the
last
application
is
done
with
the
packet,
it
will
delete
itself.

"Joseph
M.
Newcomer"
<newcomer@xxxxxxxxxxxx>
wrote
in
message
news:vbvt54p0cvlvsheu97igbige2hbo3qa14d@xxxxxxxxxxx

But
what
good
does
a
shared_ptr
do
here?
It
is
overkill.
joe
On
Thu,
19
Jun
2008
09:37:36
-0700,
"Nick
Schultz"
<nick.schultz@xxxxxxxx>
wrote:

Re: how to store list of varying types

MFC
Feature
Pack
includes
TR1
which
has
shared_ptrs.

"Giovanni
Dicano"
<giovanni.dicano@xxxxxxxxxxx>
wrote
in
message
[news:Oym\\$PJe0IHA.2384@xxxxxxxxxxxxxxxx](mailto:news:Oym$PJe0IHA.2384@xxxxxxxxxxxxxxxx)

"Nick
Schultz"
<nick.schultz@xxxxxxx>
ha
scritto
nel
messaggio
news:ekB5f1Y0IHA.4500@xxxxxxxx

I
planned
on
creating
a
ProtocolPacket
class
that
represents
an
entire
packet,
and
contains
a
vector
of
dataElements.
dataElement
is
a
class
that

Re: how to store list of varying types

contains
a
pointer
to
the
data,
its
size(in
bytes)
and
a
char*
that
stores
its
field
name.

I
would
need
more
details,
but
in
general
I
would
say
that
in
C++,
I
prefer
using
std::vector
as
container
(instead
of
raw
pointer),
and
std::wstring
or
some
other
string
class
instead

Re: how to store list of varying types

of
char*.

Moreover,
there
is
a
usual
naming
convention
in
C++,
that
class
names
start
with
an
upper-case
letter
(so,
I
would
use
DataElement
instead
of
dataElement).
Lower-case
tends
to
be
used
for
other
cases,
like
class
instances.
e.g.

```
//  
Instantiate  
a  
DataElement  
DataElement  
dataElement;
```

So,
I
would

Re: how to store list of varying types

```
define
a
class
or
a
struct
like
this:

class
DataElement
{
public:

std::vector<
BYTE
>
Data;

//
You
don't
need
a
size-in-bytes
field
here,
//
because
vector
has
a
size()
method
for
//
that
purpose.
//
So
Data.size()
gives
you
that
size.

//
I
assume
that
your
```

Re: how to store list of varying types

"field
names"
here
are
ANSI
only.
//
For
Unicode,
you
may
use
std::wstring.
std::string
Name;
};

Then
I
would
store
all
these
DataElement's
in
a
vector
like
this:

```
typedef  
std::vector<  
DataElement  
*  
>  
DataElementList;
```

```
DataElementList  
myDataElements;
```

Note
that
the
vector
stores
pointers
to
DataElement
instances.
If
these

Re: how to store list of varying types

pointers
have
a
shared
ownership
semantic,
I
would
wrap
them
in
a
smart
pointer
like
shared_ptr.
e.g.

```
typedef  
boost::shared_ptr<  
DataElement  
>  
DataElementSP;  
typedef  
std::vector<  
DataElementSP  
>  
DataElementList;
```

In
that
way,
you
don't
have
to
pay
attention
to
DataElement
destruction
(the
shared_ptr
smart
pointer
stores
a
reference
count,
and
when

Re: how to store list of varying types

it gets 0, the object is automatically deleted).

My original implementation called for malloc'ing the necessary space on the heap,

In C++, you would use new[] instead of malloc(), or a robust container like std::vector.

```
SomeType  
*  
p  
=  
new  
SomeType[  
count  
];  
  
std::vector<
```

Re: how to store list of varying types

SomeType

>

```
v[  
count  
];
```

From
vector,
you
can
have
the
pointer
to
the
first
element
using:

SomeType

*

```
pFirst  
=  
&v[0];
```

If
you
use
new[],
you
must
also
delete
(sooner
or
later)
your
data,
using
delete[].
Instead,
vector
has
a
destructor
that
does
cleanup.

Moreover,
vector

Re: how to store list of varying types

can
safely
grow
its
size
if
necessary
(e.g.
after
a
.push_back(
<new
data>
);
),
and
it's
guarded
against
buffer
overruns
(which
are
security
enemy
#1).
Instead,
using
raw
new[],
you
may
have
lots
of
problems
like
off-by-one
index,
or
index
completely
out-of-range,
corrupting
nearby
memory,
etc.
It's
not
that
you

Re: how to store list of varying types

must
not
use
new[]:
you
may
use
new[],
but
you
(or
those
who
will
maintain
your
code)
must
pay
lots
more
attention,
and
the
code
is
less
robust,
more
fragile,
than
using
a
robust
C++
container
class
like
std::vector.

Note
that
there
are
also
MFC
versions
of
the
classes
I

Re: how to store list of varying types

mentioned
in
this
post:
you
can
use
CString
to
store
strings,
and
CArray
template
instead
of
std::vector.
(AFAIK,
MFC
has
no
equivalent
of
smart
pointer
like
shared_ptr...).

HTH,
Giovanni

Joseph
M.
Newcomer
[MVP]
email:
newcomer@xxxxxxxxxxxxx
Web:
<http://www.flounder.com>
MVP
Tips:
http://www.flounder.com/mvp_tips.htm

Joseph M.
Newcomer

Re: how to store list of varying types

[MVP]
email:
newcomer@xxxxxxxxxxxxx
Web:
<http://www.flounder.com>
MVP Tips:
http://www.flounder.com/mvp_tips.htm

Joseph M. Newcomer [MVP]
email: newcomer@xxxxxxxxxxxxx
Web: <http://www.flounder.com>
MVP Tips:
http://www.flounder.com/mvp_tips.htm

Joseph M. Newcomer [MVP]
email: newcomer@xxxxxxxxxxxxx
Web: <http://www.flounder.com>
MVP Tips: http://www.flounder.com/mvp_tips.htm

Joseph M. Newcomer [MVP]
email: newcomer@xxxxxxxxxxxxx
Web: <http://www.flounder.com>
MVP Tips: http://www.flounder.com/mvp_tips.htm

.