Australian Government
Geoscience Australia

GINAN

Coding standard for C++

Ginan documentation

March, 2022

Document approvals

Role	Name	Signature	Date
Prepared:	Aaron Hammond	–	18-3-2022
Approved:	Simon McClusky	–	18-3-2022

Document history

Version	Dated	Author	Notes
Beta	18-3-2022	Aaron Hammond	Ginan Beta release
–	–	–	–

Contents

1 Introduction

1.1 The Positioning Australia Program

The Australian Government is making a significant investment in the Positioning Australia program through Geoscience Australia. The program contains three major projects:

- The commercial procurement and operation of a Satellite Based Augmentation System (SBAS) called SouthPAN which will enhance positioning across the region through the provision of extra GNSS signals and data delivered from a geostationary satellite.
- The enhancement of the National Positioning Infrastructure Capability (NPIC) which will see upgrades to and an expansion of the Global Navigation Satellite System (GNSS) Continuously Operating Reference Station (CORS) network across the South Pacific and Antarctica.
- Ginan is an open source Precise Point Positioning (PPP) toolkit. It can produce PPP position correction products and, operating in another mode, use GNSS observations and those correction products to determine positions with an accuracy in the centimetre range.

The program is summarised in Figure 1.1 below.

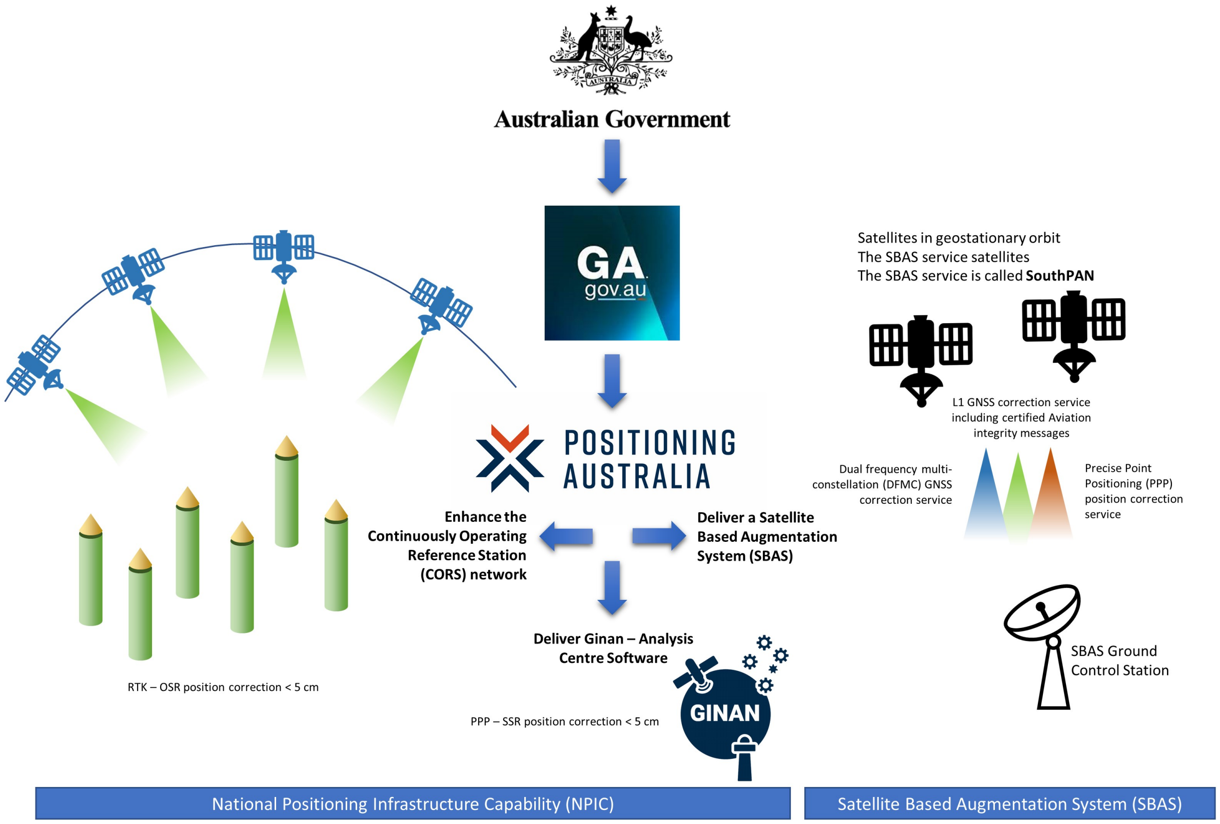
Australian Government
GA gov.au
POSITIONING AUSTRALIA
Enhance the Continuously Operating Reference Station (CORS) network
Deliver a Satellite Based Augmentation System (SBAS)
Deliver Ginan – Analysis Centre Software
GINAN
RTK – OSR position correction < 5 cm
PPP – SSR position correction < 5 cm
Satellites in geostationary orbit
The SBAS service satellites
The SBAS service is called SouthPAN
L1 GNSS correction service including certified Aviation integrity messages
Dual frequency multi-constellation (DFMC) GNSS correction service
Precise Point Positioning (PPP) position correction service
SBAS Ground Control Station
National Positioning Infrastructure Capability (NPIC)
Satellite Based Augmentation System (SBAS)

Figure 1.1: The three main projects in the Positioning Australia program.

1.2 Ginan - Analysis Centre Software

Ginan, is being rolled out in a phased approach and will offer products in four distinct categories:

- The software itself. Ginan is open-source software that GA has hosted on a GitHub repository.
- Standard precise point positioning (PPP) product files. An operational version of Ginan, maintained by Geoscience Australia (GA), will produce on a 24 X 7 basis, a range of standard PPP product files including, for example, a precise orbits and clocks file in SP3 format.
- Precise point positioning correction messages. An operational version of Ginan, maintained by GA, will stream over the internet on a 24 X 7 basis, a range of PPP correction messages in the RTCM3 message format.
- New PPP products and applications yet to be defined. The Ginan toolkit gives GA the ability to offer new PPP products, yet to be defined, but which, in collaboration with users, may spawn new applications and commercial opportunities.

Ginan is summarised in Figure 1.2 below.

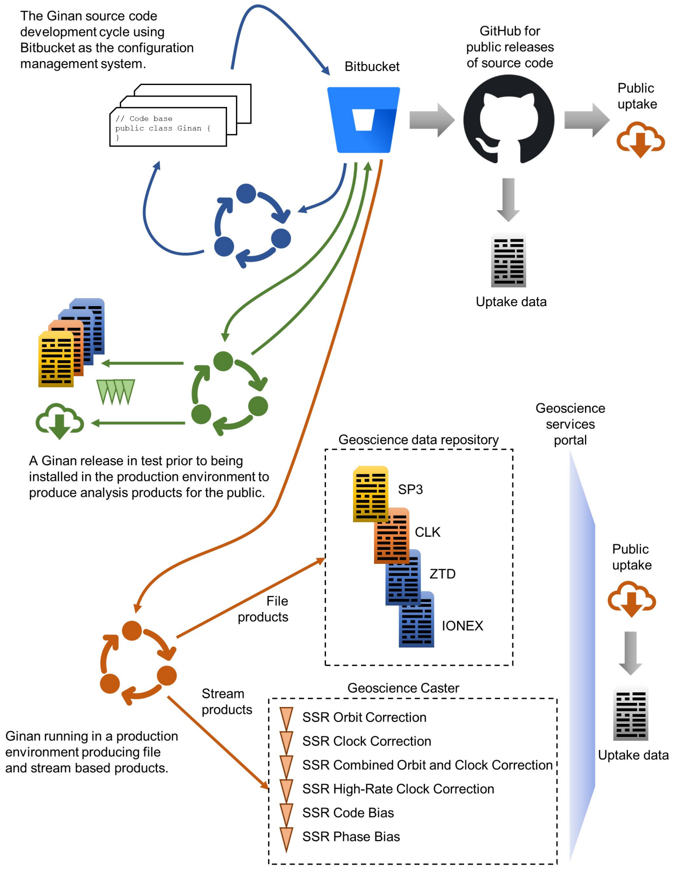
The Ginan source code development cycle using Bitbucket as the configuration management system.
// Code base
public class Ginan {
}
Bitbucket
GitHub for public releases of source code
Public uptake
Uptake data
A Ginan release in test prior to being installed in the production environment to produce analysis products for the public.
Geoscience data repository
SP3
CLK
ZTD
IONEX
Geoscience services portal
Public uptake
Uptake data
File products
Stream products
Ginan running in a production environment producing file and stream based products.
Geoscience Caster
SSR Orbit Correction
SSR Clock Correction
SSR Combined Orbit and Clock Correction
SSR High-Rate Clock Correction
SSR Code Bias
SSR Phase Bias

Figure 1.2: The Ginan product offering.

1.3 This document

This document forms part of the Ginan documentation suite. Geoscience Australia has made Ginan open source to encourage researchers and developers to use and adapt the software for their own purposes, and to contribute new ideas and innovations back to the code base. To make this exchange as smooth as possible, Geoscience Australia recommends that developers follow this coding standard to ensure a consistent code presentation and look and feel.

2 Coding Standards for C++

2.1 Code style

Decades of experience has shown that codebases that are built with concise, clean code have fewer issues and are easier to maintain. If submitting a pull request for a patch to the software, please ensure your code meets the following standards.
Overall we are aiming to

- Write for clarity
- Write for clarity
- Write what you mean, not what is implied
- Write things once
- Use short, descriptive variable names
- Use aliases to reduce clutter.

Inconcise code - Not recommended

```
//check first letter of satellite type against something

if (obs.Sat.id().c_str()[0]) == 'G')
    doSomething();
else if (obs.Sat.id().c_str()[0]) == 'R')
    doSomething();
else if (obs.Sat.id().c_str()[0]) == 'E')
    doSomething();
else if (obs.Sat.id().c_str()[0]) == 'I')
    doSomething();
```

Clear Code - Good

```
char& sysChar = obs.Sat.id().c_str()[0];

switch (sysChar)
{
    case 'G':   doSomething();   break;
    case 'R':   doSomething();   break;
    case 'E':   doSomething();   break;
    case 'I':   doSomething();   break;
}
```

2.2 Spacing, Indentation, and layout

- Use tabs, with tab spacing set to 4.
- Use space or tabs before and after any

$$+ \; - \; * \; / \; = \; < \; > \; == \; != \; \%$$

etc..

- Use space, tab or new line after any , ;
- Use a new line after if statements.
- Use tabs to keep things tidy - If the same function is called multiple times with different parameters, the parameters should line up.

Scattered Parameters - Bad

```
trySetFromYaml(mongo_metadata,output_files,{"mongo_metadata" });
trySetFromYaml(mongo_output_measurements,output_files,{"mongo_output_measurements" });
trySetFromYaml(mongo_states,output_files,{"mongo_states" });
```

Aligned Parameters - Good

```
trySetFromYaml(mongo_metadata,              output_files, {"mongo_metadata"              });
trySetFromYaml(mongo_output_measurements,   output_files, {"mongo_output_measurements"   });
trySetFromYaml(mongo_states,                output_files, {"mongo_states"                });
```

2.3 Statements

One statement per line - * unless you have a very good reason

Multiple Statements per Line - Bad

```
z[k]=ROUND(zb[k]); y=zb[k]-z[k]; step[k]=SGN(y);
```

Single Statement per Line - Good

```
z[k]    = ROUND(zb[k]);
y       = zb[k]-z[k];
step[k] = SGN(y);
```

Example of a good reason:

* Multiple statements per line sometimes shows repetitive code more clearly, but put some spaces so the separation is clear.

Normal

```
switch (sysChar)
{
    case ' ':
    case 'G':
        *sys = E_Sys::GPS;
        *tsys = TSYS_GPS;
        break;
    case 'R':
        *sys = E_Sys::GLO;
        *tsys = TSYS_UTC;
        break;
    case 'E':
        *sys = E_Sys::GAL;
        *tsys = TSYS_GAL;
        break;
    //...continues
}
```

Ok

```
if      (sys == SYS_GLO)    fact = EFACT_GLO;
else if (sys == SYS_CMP)    fact = EFACT_CMP;
else if (sys == SYS_GAL)    fact = EFACT_GAL;
else if (sys == SYS_SBS)    fact = EFACT_SBS;
else                        fact = EFACT_GPS;
```

Ok

```
switch (sysChar)
{
    case ' ':
    case 'G':   *sys = E_Sys::GPS;      *tsys = TSYS_GPS;    break;
    case 'R':   *sys = E_Sys::GLO;      *tsys = TSYS_UTC;    break;
    case 'E':   *sys = E_Sys::GAL;      *tsys = TSYS_GAL;    break;
    case 'S':   *sys = E_Sys::SBS;      *tsys = TSYS_GPS;    break;
    case 'J':   *sys = E_Sys::QZS;      *tsys = TSYS_QZS;    break;
//...continues
}
```

2.4 Braces

New line for braces.

```
if (pass)
{
    doSomething();
}
```

2.5 Comments

- Prefer `//` for comments within functions
- Use `/* */` only for temporary removal of blocks of code.
- Use `/** */` and `///<` for automatic documentation

2.6 Conditional checks

- Put `&&` and `||` at the beginning of lines when using multiple conditionals.
- Always use curly braces when using multiple conditionals.

```
if  ( ( testA     > 10)
    &&( testB   == false
      ||testC   == false))
{
    //do something
}
```

- Use variables to name return values rather than using functions directly

Bad

```
if (doSomeParsing(someObject))
{
    //code contingent on parsing success? failure?
}
```

Good

```
bool fail = doSomeParsing(someObject);
if (fail)
{
    //This code is clearly a response to a failure
}
```

2.7 Variable declaration

- Declare variables as late as possible - at point of first use.
- One declaration per line.
- Declare loop counters in loops where possible.
- Always initialise variables at declaration.

```
int  type  = 0;
bool found = false;         //these have to be declared early so they can be used after the for
    loop

for (int i = 0; i < 10; i++)
{
    bool pass = someTestFunction();    //this pass variable isnt declared until it's used - good
    if (pass)
    {
        type  = typeMap[i];
        found = true;
        break;
    }
}

if (found)
{
    //...
}
```

2.8 Function parameters

- One per line.
- Add doxygen compatible documentation after parameters in the cpp file.
- Prefer references rather than pointers unless unavoidable.

```
void function(
        bool         runTests,          ///< Run unit test while processing
        MyStruct&    myStruct,          ///< Structure to modify
        OtherStr*    otherStr = nullptr) ///< Optional structure object to populate (cant use
    reference because its optional)
{
    //...
}
```

2.9 Naming and Structure

- For structs or classes, use ‘CamelCase‘ with capital start
- For member variables, use ‘camelCase‘ with lowercase start
- For config parameters, use ‘lowercase_with_underscores‘
- Use suffixes (‘_ptr‘, ‘_arr‘, ‘Map‘, ‘List‘ etc.) to describe the type of container for complex types.
- Be sure to provide default values for member variables.
- Use heirarchical objects where applicable.

```
struct SubStruct
{
    int    type = 0;
    double val  = 0;
};

struct MyStruct
{
    bool          memberVariable = false;
    double        precision      = 0.1;

    double                      offset_arr[10]  = {};
    OtherStruct*                refStruct_ptr   = nullptr;

    map<string, double>         offsetMap;
    list<map<string, double>>   variationMapList;
    map<int, SubStruct>         subStructMap;
};

//...

MyStruct myStruct = {};

if (acsConfig.some_parameter)
{
    //..
}
```

2.10 Testing

- Use TestStack objects at top of each function that requires automatic unit testing.
- Use TestStack objects with descriptive strings in loops that wrap functions that require automatic unit testing.

```
void function()
{
    TestStack ts(__FUNCTION__);

    //...

    for (auto& obs : obsList)
    {
        TestStack ts(obs.Sat.id());

        //...
    }
}
```

2.11 Documentation

- Use doxygen style documentation for function and struct headers and parameters
- `/**` for headers.
- `///<` for parameters

```
/** Struct to demonstrate documentation.
* The first line automatically gets parsed as a brief description, but more detailed
    descriptions are possible too.
*/
struct MyStruct
{
    bool    dummyBool;                  ///< The thing to the left is documented here
};

/** Function to demonstrate documentation
*/
void function(
        bool        runTests,           ///< Run unit test while processing
        MyStruct&   myStruct,           ///< Structure to modify
        OtherStr*   otherStr = nullptr) ///< Optional string to populate
{
    //...
}
```

2.12 STL Templates

- Prefer maps rather than fixed arrays.
- Prefer range-based loops rather than iterators or ‘i‘ loops, unless unavoidable.

Bad

```
double double_arr[10] = {};

//..(Populate array)

for (int i = 0; i < 10; i++)    //Magic number 10 - bad.
{

}
```

```
map<string, double> doubleMap;

//..(Populate Map)

for (auto iter = doubleMap.begin(); iter != doubleMap.end(); iter++)   //long, undescriptive - bad
{
    if (iter->first == someVar)     //'first' is undescriptive - bad
    {
        //..
    }
}
```

Good - Iterating Maps

```
map<string, double> offsetMap;

//..(Populate Map)

for (auto& [siteName, offset] : doubleMap)  //give readable names to map keys and values
{
    if (siteName.empty() == false)
    {

    }
}
```

Good - Iterating Lists

```
list<Obs> obsList;

//..(Populate list)

for (auto& obs : obsList)         //give readable names to list elements
{
    doSomethingWithObs(obs);
}
```

Special Case - Deleting from maps/lists

Use iterators when you need to delete from STL containers:

```
for (auto it = someMap.begin(); it != someMap.end();  )
{
    KFKey key = it->first;              //give some alias to the key/value so they're readable

    if (measuredStates[key] == false)
    {
        it = someMap.erase(it);
    }
    else
    {
        ++it;
    }
}
```

2.13 Namespaces

Commonly used std containers may be included with ‘using‘

```
#include <string>
#include <map>
#include <list>
#include <unordered_map>

using std::string;
using std::map;
using std::list
using std::unordered_map;
```

3 Attribution

Ginan - Analysis Centre Software
A project funded as Part of the Positioning Australia program.
Geoscience Australia
https://www.ga.gov.au/scientific-topics/positioning-navigation/positioning-australia
clientservices@ga.gov.au
Cnr Jerrabomberra Ave and Hindmarsh Drive
Symonston ACT 2609
Australia