C++ CORE GUIDELINES PHILOSOPHY

BJARNE STROUSTRUP & HERB SUTTER

https://github.com/ufoym/cpp-core-guidelines-cheatsheet

P.1: EXPRESS IDEAS DIRECTLY IN CODE

```
class Date {
public:
    Month month() const;
    int month();
    // ...
};
```

O explicit about returning a Month

O explicit about not modifying the state of the Date object

X Leaves the reader guessing and opens more possibilities for uncaught bugs

```
void f(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    int index = -1;
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] == val) {
            index = i;
            break;
        }
    }
    // ...
    auto p = find(begin(v), end(v), val);
    // ...
}
```

X bad, plus should use gsl::index

O A much clearer expression of intent. A C++ programmer should know the basics of the standard library, and use it where appropriate.

```
change_speed(double s);
// ...
change_speed(2.3);

change_speed(Speed s);
// ...
change_speed(2.3);          // error: no unit
change_speed(23_m / 10s); // meters per second
```

X bad: what does s signify?

O better: the meaning of s is specified

P.2: WRITE IN ISO STANDARD C++

There are environments where extensions are necessary, e.g., to access system resources. In such cases, Localize the use of necessary extensions and control their use with non-core Coding Guidelines. If possible, build interfaces that encapsulate the extensions so they can be turned off or compiled away on systems that do not support those extensions.

Extensions often do not have rigorously defined semantics. Even extensions that are common and implemented by multiple compilers might have slightly different behaviors and edge case behavior as a direct result of not having a rigorous standard definition. With sufficient use of any such extension, expected portability will be impacted.

P.3: EXPRESS INTENT

X The implementation detail of an index is exposed (so that it might be misused), and i outlives the scope of the loop, which might or might not be intended.

X The intent of "just" Looping over the elements of v is not expressed here.

```
gsl::index i = 0;
while (i < v.size()) {
    // ... do something with v[i] ...
}
```

O the loop operates on a reference to const elements so that accidental modification cannot happen

O there is no explicit mention of the iteration mechanism

```
for (const auto& x : v) {
    // do something with the value of x
}
```

O if modification is desired

```
for (auto& x : v) {
    // modify x
}
```

Sometimes better still, use a named algorithm.

O This example uses the for_each from the Ranges TS because it directly expresses the intent

```
for_each(v, [](int x) {
    // do something with the value of x
});
```

O makes it clear that we are not interested in the order in which the elements of v are handled.

```
for_each(par, v, [](int x) {
    // do something with the value of x
});
```

X obscure

```
draw_line(int, int, int, int);
draw_line(Point, Point);
```

O clearer, if two ints are meant to be the coordinates of a 2D point.

P.4: IDEALLY, A PROGRAM SHOULD BE STATICALLY TYPE SAFE

- unions -- use **variant** (in C++17)
- casts -- minimize their use; templates can help
- array decay -- use **span** (from the GSL)
- range errors -- use **span**
- narrowing conversions -- minimize their use and use **narrow** or **narrow_cast** (from the GSL) where they are necessary

P.5: PREFER COMPILE-TIME CHECKING TO RUN-TIME CHECKING

```
// read max n integers into *p
void read(int* p, int n);
int a[100];
read(a, 1000);
```

X bad, off the end

```
// read into the range of integers r
void read(span<int> r);
int a[100];
read(a);
```

O better: let the compiler figure out the number of elements

P.6: WHAT CANNOT BE CHECKED AT COMPILE TIME SHOULD BE CHECKABLE AT RUN TIME

```
extern void f1(int* p);
void g1(int n) { f1(new int[n]); }
```

X the number of elements is not passed to f1()

```
extern void f2(int* p, int n);
void g2(int n) { f2(new int[n], m); }
```

X a wrong number of elements can be passed to f2()

```
extern void f3(unique_ptr<int[]>, int n);
void g3(int n)
{
    f3(make_unique<int[]>(n), m);
}
```

X pass ownership and size separately

```
extern void f4(vector<int>&);
extern void f4(span<int>);
void g4(int n)
{
    vector<int> v(n);
    f4(v);              // pass a reference
    f4(span<int>{v}); // pass a view
}
```

O This design carries the number of elements along as an integral part of an object, so that errors are unlikely and dynamic (run-time) checking is always feasible, if not always affordable.

P.7: CATCH RUN-TIME ERRORS EARLY

```
void increment1(int* p, int n)
{
    for (int i = 0; i < n; ++i) ++p[i];
}
void use1(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment1(a, m);
    // ...
}
```

X error-prone

maybe typo, assume that m == 20. The (pointer, count)-style interface leaves increment1() with no realistic way of defending itself against out-of-range errors. If we could check subscripts for out of range access, then the error would not be discovered until p[10] was accessed.

```
void increment2(span<int> p)
{
    for (int& x : p) ++x;
}
void use2(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2({a, m});
    // ...
}
```

O Now, m <= n can be checked at the point of call (early) rather than later.

```
void use3(int m)
{
    const int n = 10;
    int a[n] = {};
    // ...
    increment2(a);
    // ...
}
```

O Further simplified (eliminating the possibility of an error) if all we had was a typo so that we meant to use n as the bound

P.8: DON'T LEAK ANY RESOURCES

```
void f(char* name)
{
    FILE* input = fopen(name, "r");
    if (something) return;
    fclose(input);
}
```

X bad: if something == true, a file handle is leaked

```
void f(char* name)
{
    ifstream input {name};
    if (something) return;
}
```

O OK: no Leak

P.9: DON'T WASTE TIME OR SPACE

will be evaluated on every iteration of the loop, it's better to cache the length outside the loop

```
void lower(zstring s)
{
    for (int i = 0; i < strlen(s); ++i)
        s[i] = tolower(s[i]);
}
```

X

P.10: PREFER IMMUTABLE DATA TO MUTABLE DATA

It is easier to reason about constants than about variables. Something immutable cannot change unexpectedly. Sometimes immutability enables better optimization. You can't have a data race on a constant.

P.11: ENCAPSULATE MESSY CONSTRUCTS, RATHER THAN SPREADING THROUGH THE CODE

```
int sz = 100;
int* p = (int*) malloc(sizeof(int) * sz);
int count = 0;
// ...
for (;;) {
    // ... read an int into x, exit
    // loop if end of file is reached ...
    // ... check that x is valid ...
    if (count == sz)
        p = (int*) realloc(p,
            sizeof(int) * sz * 2);
    p[count++] = x;
    // ...
}
```

X Low-level, verbose, and error-prone

```
vector<int> v;
v.reserve(100);
// ...
for (int x; cin >> x; ) {
    // ... check that x is valid ...
    v.push_back(x);
}
```

P.12: USE SUPPORTING TOOLS AS APPROPRIATE

There are many things that are done better "by machine". Computers don't tire or get bored by repetitive tasks. We typically have better things to do than repeatedly do routine tasks.

- Static analysis tools
- Concurrency tools
- Testing tools

P.13: USE SUPPORT LIBRARIES AS APPROPRIATE

O Unless you are an expert in sorting algorithms and have plenty of time, this is more likely to be correct and to run faster than anything you write for a specific application. You need a reason not to use the standard library (or whatever foundational libraries your application uses) rather than a reason to use it.

```
std::sort(begin(v), end(v), std::greater<>());
```