Boas práticas de programação e padrões para C++

Dr. Rodrigo Mologni Gonçalves dos Santos

Conteúdo

Conteúdo

Conteúdo

Regras aritméticas

Regras

1. Não misture aritmética com sinal e sem sinal
2. Use tipos sem sinal para manipulação de *bits*
3. Use tipos com sinal para aritmética
4. Não faça *overflow*
5. Não faça *underflow*
6. Não divida pelo zero inteiro
7. Não tente evitar valores negativos usando **`unsigned`**

Regras

8. Não use **`unsigned`** para subscritos, prefira **`gsl::index`**

Não misture aritmética com sinal e sem sinal

Razão

- Evite resultados errados.

Exemplo

```
int x = -3;
unsigned int y = 7;
cout << x - y << '\n'; // unsigned result, possibly 4294967286
cout << x + y << '\n'; // unsigned result: 4
cout << x * y << '\n'; // unsigned result, possibly 4294967275
```

Use tipos sem sinal para manipulação de *bits*

Razão

- Os tipos sem sinal suportam a manipulação de *bits* sem surpresas dos *bits* de sinal.

Exemplo

```
unsigned char x = 0b1010'1010;
unsigned char y = ~x; // y == 0b0101'0101;
```

Observação

- Tipos sem sinal também podem ser úteis para aritmética de módulo. No entanto, adicione comentários, pois esse código pode ser novidade para muitos programadores.

Use tipos com sinal para aritmética

Razão

- A maior parte da aritmética faz uso de sinal.

Exemplo

```
template<typename T, typename T2> T
subtract(T x, T2 y) {
    return x - y;
}

void test() {
    int s = 5;
    unsigned int us = 5;
    cout << subtract(s, 7) << '\n'; // -2
    cout << subtract(us, 7u) << '\n'; // 4294967294
    cout << subtract(s, 7u) << '\n'; // -2
    cout << subtract(us, 7) << '\n'; // 4294967294
    cout << subtract(s, us + 2) << '\n'; // -2
    cout << subtract(us, s + 2) << '\n'; // 4294967294
}
```

Não transborde

Razão

- O estouro geralmente torna seu algoritmo numérico sem sentido.
- Aumentar um valor além de um valor máximo pode causar corrupção de memória e comportamento indefinido.

Exemplo

```
int a[10];
a[10] = 7; // bad, array bounds overflow

for (int n = 0; n <= 10; ++n)
    a[n] = 9; // bad, array bounds overflow
```

Exemplo

```
int n = numeric_limits<int>::max();
int m = n + 1; // bad, numeric overflow
```

Não faça *underflow*

Razão

- Diminuir um valor além de um valor mínimo pode causar corrupção de memória e comportamento indefinido.

Exemplo

```
int a[10];
a[-2] = 7; // bad
int n = 101;

while (n--)
    a[n - 1] = 9; // bad (twice)
```

Não divida pelo inteiro zero

Razão

- O resultado é indefinido e provavelmente um travamento.

Exemplo

```
int divide(int a, int b) { // BAD, should be checked (e.g., in a precondition)
    return a / b;
}
```

Exemplo

```
int divide(int a, int b) { // good, address via precondition (and replace with contracts
                           // once C++ gets them)
    Expects(b != 0);
    return a / b;
}

double divide(double a, double b) { // good, address via using double instead
    return a / b;
}
```

Não tente evitar valores negativos usando *unsigned*

Razão

- A escolha **`unsigned`** implica em muitas mudanças no comportamento usual de números inteiros. Ela pode suspender avisos relacionados a *overflow* e facilitar o ocorrência de erros relacionados a mistura de valores com e sem sinais.

Exemplo

```
unsigned int u1 = -2; // Valid: the value of u1 is 4294967294
int i1 = -2;
unsigned int u2 = i1; // Valid: the value of u2 is 4294967294
int i2 = u2; // Valid: the value of i2 is -2
```

Não use unsigned para subscritos, prefira *gsl::index*

Razão

- Para evitar confusão assinada/não assinada.
- Para permitir uma melhor otimização.
- Para permitir uma melhor detecção de erros.
- Para evitar as armadilhas com **`auto`** e **`int`**.

Exemplo

```
vector<int> vec = /*...*/;

for (int i = 0; i < vec.size(); i += 2) // might not be big enough
    cout << vec[i] << '\n';
for (unsigned i = 0; i < vec.size(); i += 2) // risk wraparound
    cout << vec[i] << '\n';
for (auto i = 0; i < vec.size(); i += 2) // might not be big enough
    cout << vec[i] << '\n';
for (vector<int>::size_type i = 0; i < vec.size(); i += 2) // verbose
    cout << vec[i] << '\n';
for (auto i = vec.size()-1; i >= 0; i -= 2) // bug
    cout << vec[i] << '\n';
for (int i = vec.size()-1; i >= 0; i -= 2) // might not be big enough
    cout << vec[i] << '\n';
```

Exemplo

```
vector<int> vec = /*...*/;

for (gsl::index i = 0; i < vec.size(); i += 2) // ok
    cout << vec[i] << '\n';
for (gsl::index i = vec.size()-1; i >= 0; i -= 2) // ok
    cout << vec[i] << '\n';
```

Regras de classes

Regras

1. Organizar dados relacionados em estruturas (**`struct`**s ou **`class`**es)
2. Use **`class`** se a classe tiver uma invariante; use **`struct`** se os membros dos dados puderem variar independentemente
3. Representar a distinção entre uma interface e uma implementação usando uma classe
4. Torne uma função membro apenas se ela precisar de acesso direto à representação de uma classe
5. Coloque funções auxiliares no mesmo namespace da classe que elas suportam
6. Não defina uma classe ou **`enum`** e declare uma variável de seu tipo na mesma instrução

Regras

7. Use **`class`** em vez de **`struct`** se algum membro não for público
8. Minimizar a exposição dos membros

Organizar dados relacionados em estruturas (*structs* ou *classes*)

Razão

- Facilidade de compreensão.
- Se os dados estiverem relacionados (por razões fundamentais), esse fato deverá ser refletido no código.

Exemplo

```
void draw(int x, int y, int x2, int y2); // BAD: unnecessary implicit relationships
void draw(Point from, Point to); // better
```

Use *class* se a classe tiver uma invariante; use *struct* se os membros dos dados puderem variar independentemente

Razão

- Legibilidade.
- Facilidade de compreensão.
- O uso de **`class`** alerta o programador sobre a necessidade de uma invariante.
- Esta é uma convenção útil.

Exemplo

```
struct Pair { // the members can vary
                independently
    string name;
    int volume;
};
```

```
class Date {
public:
    // validate that {yy, mm, dd} is a
       valid date and initialize
    Date(int yy, Month mm, char dd);
    // ...
private:
    int y;
    Month m;
    char d; // day
};
```

Representar a distinção entre uma interface e uma implementação usando uma classe

Razão

- Uma distinção explícita entre interface e implementação melhora a legibilidade e simplifica a manutenção.

Exemplo

```
class Date {
public:
    Date();
    // validate that {yy, mm, dd} is a valid date and initialize
    Date(int yy, Month mm, char dd);

    int day() const;
    Month month() const;
    // ...
private:
    // ... some representation ...
};
```

Torne uma função membro apenas se ela precisar de acesso direto à representação de uma classe

Razão

- Menos acoplamento com funções-membro.
- Menos funções que podem causar problemas ao modificar o estado do objeto.
- Redução do número de funções que precisam ser modificadas após uma mudança na representação.

Exemplo

```
class Date {
    // ... relatively small interface ...
};

// helper functions:
Date next_weekday(Date);
bool operator==(Date, Date);
```

Exceções

- C++ exige que funções **`virtual`** sejam membros e nem todas as funções **`virtual`** acessam diretamente os dados.
- C++ requer que os operadores **=**, **( )**, **[ ]** e **->** sejam membros.
- Um conjunto de funções poderia ser projetado para ser usado em uma cadeia:

```
x.scale(0.5).rotate(45).set_color(Color::red);
```

Exceções

- Um conjunto de sobrecarga pode ter alguns membros que não acessam dados privados diretamente.

```
class Foobar {
public:
    void foo(long x) { /* manipulate private data */ }
    void foo(double x) { foo(std::lround(x)); }
    // ...
private:
    // ...
};
```

Coloque funções auxiliares no mesmo *namespace* da classe que elas suportam

Razão

- Uma função auxiliar é uma função que não precisa de acesso direto à representação da classe, mas é vista como parte da interface útil para a classe. Colocá-los no mesmo **`namespace`** da classe torna óbvio seu relacionamento com a classe e permite que eles sejam encontrados por pesquisa dependente de argumento.

Exemplo

```
namespace Chrono { // here we keep time-related services

    class Time { /* ... */ };
    class Date { /* ... */ };

    // helper functions:
    bool operator==(Date, Date);
    Date next_weekday(Date);
    // ...
}
```

Não defina uma classe ou *enum* e declare uma variável de seu tipo na mesma instrução

Razão

- Misturar uma definição de tipo e a definição de outra entidade na mesma declaração é confuso e desnecessário.

Exemplo

```
struct Data { /*...*/ } data{ /*...*/ };
```

```
struct Data { /*...*/ };
Data data{ /*...*/ };
```

Use *class* em vez de *struct* se algum membro não for público

Razão

- Legibilidade.
- Para deixar claro que algo está sendo oculto/abstraído.
- Esta é uma convenção útil.

Exemplo

```
struct Date {
    int d, m;
    Date(int i, Month m);
    // ... lots of functions ...
private:
    int y; // year
};
```

Minimizar a exposição dos membros

Razão

- Encapsulamento.
- Ocultação de informações.
- Minimizar a chance de acesso não intencional.
- Simplificar a manutenção.

Exemplo

```
template<typename T, typename U>
struct pair {
    T a;
    U b;
    // ...
};
```

```
class Distance {
public:
    // ...
    double meters() const { return magnitude*unit; }
    void set_unit(double u)
    {
        // ... check that u is a factor of 10 ...
        // ... change magnitude appropriately ...
        unit = u;
    }
    // ...
private:
    double magnitude;
    double unit; // 1 is meters, 1000 is kilometers,
                 0.001 is millimeters, etc.
};
```

```
class Foo {
public:
    int bar(int x) { check(x); return do_bar(x); }
    // ...
protected:
    int do_bar(int x); // do some operation on the data
    // ...
private:
    // ... data ...
};

class Dir : public Foo {
    //...
    int mem(int x, int y) {
        /* ... do something ... */
        return do_bar(x + y); // OK: derived class can bypass check
    }
};

void user(Foo& x) {
    int r1 = x.bar(1); // OK, will check
    int r2 = x.do_bar(2); // error: would bypass check // ...
}
```

Regras de constantes

Regras

1. Por padrão, torne objetos imutáveis
2. Por padrão, crie funções de membro **`const`**
3. Por padrão, passe ponteiros e referências para **`const`**s
4. Use **`const`** para definir objetos com valores que não mudam após a construção
5. Use **`constexpr`** para valores que podem ser calculados em tempo de compilação

Por padrão, torne objetos imutáveis

Razão

- Objetos imutáveis são mais fáceis para raciocinar. Portanto, torne os objetos não-**`const`** apenas quando houver necessidade de alterar seu valor. Evita alterações de valor acidentais ou difíceis de perceber.

Exemplo

```
for (const int i : c) cout << i << '\n'; // just reading: const
for (int i : c) cout << i << '\n'; // BAD: just reading
```

Exceção

```
void f(const char* const p); // pedantic
void g(const int i) { ... } // pedantic
```

Por padrão, crie funções de membro *const*

Razão

- Uma função membro deve ser marcada **`const`** ao menos que altere o estado de um objeto. Isso fornece uma declaração mais precisa da intenção do projeto, melhor legibilidade, mais erros detectados pelo compilador e, às vezes, mais oportunidades de otimização.

Exemplo

```
class Point {
    int x, y;
public:
    int getx() { return x; } // BAD, should be const as it doesn't modify the object's
                                state
    // ...
};

void f(const Point& pt)
{
    int x = pt.getx(); // ERROR, doesn't compile because getx was not marked const
}
```

Por padrão, passe ponteiros e referências para *consts*

Razão

- Para evitar que uma função chamada altere inesperadamente o valor. É muito mais fácil raciocinar sobre programas quando funções chamadas não modificam o estado.

Exemplo

```
void f(char* p); // does f modify *p? (assume it does)
void g(const char* p); // g does not modify *p
```

Use *const* para definir objetos com valores que não mudam após a construção

Razão

- Evite surpresas causadas por valores de objetos alterados inesperadamente.

Exemplo

```
void f()
{
    int x = 7;
    const int y = 9;

    for (;;) {
        // ...
    }
    // ...
}
```

Use *constexpr* para valores que podem ser calculados em tempo de compilação

Razão

- Melhor desempenho, melhor verificação em tempo de compilação, avaliação garantida em tempo de compilação, sem possibilidade de condições de corrida.

Exemplo

```
double x = f(2); // possible run-time evaluation
const double y = f(2); // possible run-time evaluation
constexpr double z = f(2); // error unless f(2) can be evaluated at compile time
```

Regras de declarações

Regras

1. Mantenha os escopos pequenos
2. Declare nomes em inicializadores de instrução **`for`** e condições para limitar o escopo
3. Mantenha os nomes comuns e locais curtos e mantenha os nomes incomuns e não locais mais longos
4. Evite nomes com aparência semelhante
5. Evite nomes **`ALL_CAPS`**
6. Use auto para evitar repetição redundante de nomes de tipos

Regras

7. Não reutilize nomes em escopos aninhados
8. Sempre inicialize um objeto
9. Não introduza uma variável (ou constante) antes de precisar usá-la
10. Não declare uma variável até que você tenha um valor para inicializá-la
11. Prefira a sintaxe de inicialização **`{}`**
12. Use **`unique_ptr<T>`** para segurar ponteiros
13. Declare um objeto **`const`** ou **`constexpr`** a menos que queira modificar seu valor posteriormente

Regras

14. Não use uma variável para dois propósitos não relacionados
15. Use **`std::array`** ou **`stack_array`** para **`array`**s na pilha
16. Use **`lambda`**s para inicialização complexa, especialmente de variáveis **`const`**
17. Não use macros para constantes ou funções

Mantenha os escopos pequenos

Razão

- Legibilidade.
- Minimize a retenção de recursos.
- Evite o uso indevido de valor.

Exemplo

```
void use() {
    int i; // bad: i is needlessly accessible after loop
    for (i = 0; i < 20; ++i) { /* ... */ }
    // no intended use of i here
    for (int i = 0; i < 20; ++i) { /* ... */ } // good: i is local to for-loop

    if (auto pc = dynamic_cast<Circle*>(ps)) { // good: pc is local to if-statement
        // ... deal with Circle ...
    } else {
        // ... handle error ...
    }
}
```

Exemplo

```
void use(const string& name) {
    string fn = name + ".txt";
    ifstream is {fn};
    Record r;
    is >> r;
    // ... 200 lines of code without
       intended use of fn or is ...
}
```

```
Record load_record(const string& name) {
    string fn = name + ".txt";
    ifstream is {fn};
    Record r;
    is >> r;
    return r;
}

void use(const string& name) {
    Record r = load_record(name);
    // ... 200 lines of code ...
}
```

Declare nomes em inicializadores de instrução *for* e condições para limitar o escopo

Razão

- Legibilidade.
- Limite a visibilidade da variável do *loop* ao escopo do *loop*.
- Evite usar a variável do *loop* para outros fins após o *loop*.
- Minimize a retenção de recursos.

Exemplo

```
void use() {
    for (string s; cin >> s;)
        v.push_back(s);

    for (int i = 0; i < 20; ++i) { // good: i is local to for-loop
        // ...
    }

    if (auto pc = dynamic_cast<Circle*>(ps)) { // good: pc is local to if-statement
        // ... deal with Circle ...
    } else {
        // ... handle error ...
    }
}
```

Mantenha os nomes comuns e locais curtos e mantenha os nomes incomuns e não locais mais longos

Razão

- Legibilidade.
- Reduzir a chance de conflitos entre nomes não locais e não relacionados.

Exemplo

```
template<typename T> // good
void print(ostream& os, const vector<T>& v) {
    for (gsl::index i = 0; i < v.size(); ++i)
        os << v[i] << '\n';
}
```

Exemplo

```
template<typename Element_type> // bad: verbose, hard to read
void print(ostream& target_stream, const vector<Element_type>& current_vector) {
    for (gsl::index current_element_index = 0; current_element_index <
         current_vector.size(); ++current_element_index )
        target_stream << current_vector[current_element_index] << '\n';
}
```

Exemplo

```
void complicated_algorithm(vector<Record>& vr, const vector<int>& vi, map<string, int>& out)
// read from events in vr (marking used Records) for the indices in
// vi placing (name, index) pairs into out
{
    // ... 500 lines of code using vr, vi, and out ...
}
```

Evite nomes com aparência semelhante

Razão

- Clareza e legibilidade do código.
- Nomes muito semelhantes retardam a compreensão e aumentam a probabilidade de erro.

Exemplo

```
if (readable(i1 + l1 + ol + o1 + o0 + ol + o1 + I0 + l0)) surprise();
```

Exemplo

```
struct foo { int n; };
struct foo foo(); // BAD, foo is a type already in scope
struct foo x = foo(); // requires disambiguation
```

Evite nomes *ALL_CAPS*

Razão

- Esses nomes são comumente usados para macros.
- Os nomes **`ALL_CAPS`** são vulneráveis à substituição não intencional de macros.

Exemplo

```
// somewhere in some header:
#define NE !=

// somewhere else in some other header:
enum Coord { N, NE, NW, S, SE, SW, E, W };

// somewhere third in some poor programmer's .cpp:
switch (direction) {
case N:
    // ...
case NE:
    // ...
// ...
}
```

Use *auto* para evitar repetição redundante de nomes de tipos

Razão

- A repetição simples é tediosa e propensa a erros.
- Quando você usa **`auto`**, o nome da entidade declarada fica em uma posição fixa na declaração, aumentando a legibilidade.

Exemplo

```
auto p = v.begin(); // vector<DataRecord>::iterator
auto z1 = v[3]; // makes copy of DataRecord
auto& z2 = v[3]; // avoids copy
const auto& z3 = v[3]; // const and avoids copy
auto h = t.future();
auto q = make_unique<int[]>(s);
auto f = [](int x) { return x + 10; };
```

Exceção

```
auto lst = { 1, 2, 3 }; // lst is an initializer list
auto x{1}; // x is an int (in C++17; initializer_list in C++11)
```

Não reutilize nomes em escopos aninhados

Razão

- É fácil ficar confuso sobre qual variável é usada.
- Pode causar problemas de manutenção.

Exemplo

```
int d = 0; // ...

if (cond) {
    // ...
    d = 9;
    // ...
} else {
    // ...
    int d = 7;
    // ...
    d = value_to_be_returned;
    // ...
}

return d;
```

Sempre inicialize um objeto

Razão

- Evite erros de “usado antes de definir” e comportamento indefinido associado.
- Evite problemas com a compreensão de inicialização complexa.
- Simplifique a refatoração.

Exemplo

```
void use(int arg) {
    int i; // bad: uninitialized variable // ...
    i = 7; // initialize i
}
```

Exemplo

```
widget i; // "widget" a type that's expensive to
             initialize, possibly a large POD
widget j;

if (cond) { // bad: i and j are initialized "late"
    i = f1();
    j = f2();
} else {
    i = f3();
    j = f4();
}
```

```
pair<widget, widget> make_related_widgets(bool x)
{
    return (x) ? {f1(), f2()} : {f3(), f4()};
}
auto [i, j] = make_related_widgets(cond); // C++17
```

Não introduza uma variável (ou constante) antes de precisar usá-la

Razão

- Legibilidade.
- Para limitar o escopo em que a variável pode ser usada.

Exemplo

```
int x = 7;
// ... no use of x here ...
++x;
```

Não declare uma variável até que você tenha um valor para inicializá-la

Razão

- Legibilidade.
- Limite o escopo em que uma variável pode ser usada.
- Não arrisque “usado antes de definir”.
- A inicialização costuma ser mais eficiente que a atribuição.

Exemplo

```
string s;
// ... no use of s here ...
s = "what a waste";
```

Exemplo

```
SomeLargeType var; // Hard-to-read CaMeLcAsEvArIaBlE

if (cond) // some non-trivial condition
    Set(&var);
else if (cond2 || !cond3) {
    var = Set2(3.14);
} else {
    var = 0;
    for (auto& e : something)
        var += e;
}

// use var; that this isn't done too early can be enforced statically with only control flow
```

Prefira a sintaxe de inicialização {}

Razão

- Prefira **{}**.
- As regras para inicialização **{}** são mais simples, mais gerais, menos ambíguas e mais seguras do que para outras formas de inicialização.
- Use **=** somente quando tiver certeza de que não haverá redução de conversões.
- Para tipos aritméticos integrados, use **=** somente com auto.

Exemplo

```
int x {f(99)};
int y = x;
vector<int> v = {1, 2, 3, 4, 5, 6};
```

Exemplo

```
vector<int> v1(10); // vector of 10 elements with the default value 0
vector<int> v2{10}; // vector of 1 element with the value 10
vector<int> v3(1, 2); // vector of 1 element with the value 2
vector<int> v4{1, 2}; // vector of 2 elements with the values 1 and 2
```

Exemplo

```
int x {7.9}; // error: narrowing
int y = 7.9; // OK: y becomes 7. Hope for a compiler warning
int z = gsl::narrow_cast<int>(7.9); // OK: you asked for it
```

Exemplo

```
auto x1 {7}; // x1 is an int with the value 7
auto x2 = {7}; // x2 is an initializer_list<int> with an element 7
auto x11 {7, 8}; // error: two initializers
auto x22 = {7, 8}; // x22 is an initializer_list<int> with elements 7 and 8
```

Use *unique_ptr<T>* para segurar ponteiros

Razão

- Usar **`std::unique_ptr`** é a maneira mais simples de evitar vazamentos.
- É confiável, faz com que o sistema de tipos faça grande parte do trabalho para validar a segurança da propriedade, aumenta a legibilidade e tem custo de tempo de execução zero ou quase zero.

Exemplo

```
void use(bool leak) {
    auto p1 = make_unique<int>(7); // OK
    int* p2 = new int{7}; // bad: might leak
    // ... no assignment to p2 ...
    if (leak) return;
    // ... no assignment to p2 ...
    vector<int> v(7);
    v.at(7) = 0; // exception thrown
    delete p2; // too late to prevent leaks // ...
}
```

Declare um objeto *const* ou *constexpr* a menos que queira modificar seu valor posteriormente

Razão

- Dessa forma, você não pode alterar o valor por engano.
- Essa forma pode oferecer oportunidades de otimização do compilador.

Exemplo

```
void f(int n) {
    const int bufmax = 2 * n + 2; // good: we can't change bufmax by accident
    int xmax = n; // suspicious: is xmax intended to change?
    // ...
}
```

Não use uma variável para dois propósitos não relacionados

Razão

- Legibilidade e segurança.

Exemplo

```
void use() {
    int i;
    for (i = 0; i < 20; ++i) { /* ... */ }
    for (i = 0; i < 200; ++i) { /* ... */ } // bad: i recycled
}
```

Exemplo

```
void write_to_file()
{
    std::string buffer; // to avoid reallocations on every loop iteration
    for (auto& o : objects) {
        // First part of the work.
         generate_first_string(buffer, o);
         write_to_file(buffer);

         // Second part of the work.
         generate_second_string(buffer, o);
         write_to_file(buffer);

         // etc...
    }
}
```

Use *std::array* ou *stack_array* para *arrays* na pilha

Razão

- Estes tipos de **`array`**s são legíveis e não são convertidos implicitamente em ponteiros.

Exemplo

```
const int n = 7;
int m = 9;

void f() {
    int a1[n];
    int a2[m]; // error: not ISO C++
    // ...
}
```

```
const int n = 7;
int m = 9;

void f() {
    array<int, n> a1;
    stack_array<int> a2(m);
    // ...
}
```

Use *lambdas* para inicialização complexa, especialmente de variáveis *const*

Razão

- A função **`lambda`** encapsula perfeitamente a inicialização local, incluindo a limpeza de variáveis temporárias necessárias apenas para a inicialização, sem a necessidade de criar uma função desnecessária, não local, mas não reutilizável.
- Também funciona para variáveis que devem ser **`const`**, mas somente após algum trabalho de inicialização.

Exemplo

```
widget x; // should be const, but:
for (auto i = 2; i <= N; ++i) { // this could be some
    x += some_obj.do_something_with(i); // arbitrarily long code
} // needed to initialize x
// from here, x should be const, but we can't say so in code in this style
```

Exemplo

```
const widget x = [&] {
    widget val; // assume that widget has a default constructor
    for (auto i = 2; i <= N; ++i) { // this could be some
        val += some_obj.do_something_with(i); // arbitrarily long code
    } // needed to initialize x
    return val;
}();
```

Não use macros para constantes ou funções

Razão

- Macros são uma importante fonte de *bugs*.
- As macros não obedecem às regras usuais de escopo e tipo.
- As macros não obedecem às regras usuais de passagem de argumentos.
- As macros garantem que o leitor humano veja algo diferente do que o compilador vê.
- Macros complicam a construção de ferramentas.

Exemplo

```
#define PI 3.14
#define SQUARE(a, b) (a * b)
```

```
constexpr double pi = 3.14;
template<typename T> T square(T a, T b)
{
    return a * b;
}
```

Regras de enumerações

Regras

1. Prefira enumerações em vez de macros
2. Use enumerações para representar conjuntos de constantes nomeadas relacionadas
3. Prefira **`enum class`**es em vez de **`enum`**s “simples”
4. Defina operações em enumerações para uso simples e seguro
5. Não use **`ALL_CAPS`** para enumeradores
6. Evite enumerações sem nome
7. Especifique o tipo subjacente de uma enumeração somente quando necessário

Regras

8. Especifique os valores do enumerador somente quando necessário

Prefira enumerações em vez de macros

Razão

- As macros não obedecem às regras de escopo e tipo.
- Os nomes das macros são removidos durante o pré-processamento e, portanto, geralmente não aparecem em ferramentas como depuradores.

Exemplo

```
// webcolors.h (third party header)
#define RED 0xFF0000
#define GREEN 0x00FF00
#define BLUE 0x0000FF

// productinfo.h
// The following define product subtypes based on color
#define RED 0
#define PURPLE 1
#define BLUE 2

int webby = BLUE; // webby == 2; probably not what was desired
```

Exemplo

```
enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { red = 0, purple = 1, blue = 2 };

int webby = blue; // error: be specific
Web_color webby = Web_color::blue;
```

Use enumerações para representar conjuntos de constantes nomeadas relacionadas

Razão

- Uma enumeração mostra que os enumeradores estão relacionados e pode ser um tipo nomeado.

Exemplo

```
enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
```

Prefira *enum classes* em vez de *enums* “simples”

Razão

- Para minimizar surpresas: **`enum`** tradicionais são convertidos em **`int`** com muita facilidade.

Exemplo

```
void Print_color(int color);

enum Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;

// Clearly at least one of these calls is buggy.
Print_color(webby);
Print_color(Product_info::blue);
```

Exemplo

```
void Print_color(int color);

enum class Web_color { red = 0xFF0000, green = 0x00FF00, blue = 0x0000FF };
enum class Product_info { red = 0, purple = 1, blue = 2 };

Web_color webby = Web_color::blue;

Print_color(webby); // Error: cannot convert Web_color to int.
Print_color(Product_info::red); // Error: cannot convert Product_info to int.
```

Defina operações em enumerações para uso simples e seguro

Razão

- Conveniência de uso e prevenção de erros.

Exemplo

```
enum class Day { mon, tue, wed, thu, fri, sat, sun };

Day& operator++(Day& d) {
    return d = (d == Day::sun) ? Day::mon : Day{++d}; // error: infinite recursion
}

Day today = Day::sat;
Day tomorrow = ++today; // tomorrow = Day::sun(6)
```

Exemplo

```
enum class Day { mon, tue, wed, thu, fri, sat, sun };

Day& operator++(Day& d) {
    return d = (d == Day::sun) ? Day::mon : static_cast<Day>(static_cast<int>(d)+1);
}

Day today = Day::sat;
Day tomorrow = ++today; // tomorrow = Day::sun(6)
```

Não use *ALL_CAPS* para enumeradores

Razão

- Evite conflitos com macros.

Exemplo

```
// webcolors.h (third party header)
#define RED 0xFF0000
#define GREEN 0x00FF00
#define BLUE 0x0000FF

// productinfo.h
// The following define product subtypes based on color
enum class Product_info { RED, PURPLE, BLUE }; // syntax error
```

Evite enumerações sem nome

Razão

- Se você não consegue nomear uma enumeração, os valores não estão relacionados.

Exemplo

```
// bad
enum { red = 0xFF0000, scale = 4, is_signed = 1 };

// alternative
constexpr int red = 0xFF0000;
constexpr short scale = 4;
constexpr bool is_signed = true;
```

Especifique o tipo subjacente de uma enumeração somente quando necessário

Razão

- O padrão é o mais fácil de ler e escrever.

Exemplo

```
enum class Direction : char { n, s, e, w,
                              ne, nw, se, sw }; // underlying type saves space

enum class Web_color : int32_t { red = 0xFF0000,
                                 green = 0x00FF00,
                                 blue = 0x0000FF }; // underlying type is redundand
```

Exemplo

```
// to forward-declare an enum or enum class

enum Flags : char;

void f(Flags);

// ....

enum Flags : char { /* ... */ };
```

Exemplo

```
// to ensure that values of that type have a specified bit-precision
enum Bitboard : uint64_t { /* ... */ };
```

Especifique o tipo subjacente de uma enumeração somente quando necessário

Razão

- É o mais simples.
- Evita valores duplicados do enumerador.

Exemplo

```
enum class Col1 { red, yellow, blue };
enum class Col2 { red = 1, yellow = 2, blue = 2 }; // typo
enum class Month { jan = 1, feb, mar, apr, may, jun,
                   jul, august, sep, oct, nov, dec }; // starting with 1 is conventional
enum class Base_flag { dec = 1, oct = dec << 1, hex = dec << 2 }; // set of bits
```

Regras de estilos

Regras

1. Não diga nos comentários o que pode ser claramente indicado no código
2. Declare a intenção nos comentários
3. Mantenha os comentários claros
4. Mantenha um estilo de indentação consistente
5. Evite codificar informações de tipo em nomes
6. Torne o comprimento de um nome aproximadamente proporcional ao comprimento do seu escopo

Regras

7. Use um estilo de nomenclatura consistente
8. Use **`ALL_CAPS`** apenas para nomes de macro
9. Prefira nomes **`underscore_style`**
10. Torne os literais legíveis
11. Use espaços com moderação
12. Use uma ordem de declaração de membro de classe convencional
13. Use o *layout* derivado de K&R

Regras

14. Use o *layout* de declarador estilo C++
15. Evite nomes que sejam facilmente mal interpretados
16. Não coloque duas declarações na mesma linha
17. Declare (apenas) um nome por declaração
18. Não use **`void`** como tipo de argumento
19. Use a notação convencional de **`const`**
20. Use o sufixo **`.cpp`** para arquivos de código e **`.h`** para arquivos de interface

Não diga nos comentários o que pode ser claramente indicado no código

Razão

- Os compiladores não leem comentários.
- Os comentários são menos precisos que o código.
- Os comentários não são atualizados tão consistentemente quanto o código.

Exemplo

```
auto x = m * v1 + vv;   // multiply m with v1 and add the result to vv
```

Declare a intenção nos comentários

Razão

- O código diz o que é feito, não o que deveria ser feito.
- Muitas vezes a intenção pode ser declarada de forma mais clara e concisa do que a implementação.

Exemplo

```
void stable_sort(Sortable& c)
    // sort c in the order determined by <, keep equal elements (as defined by ==) in
    // their original relative order
{
    // ... quite a few lines of non-trivial code ...
}
```

Mantenha os comentários claros

Razão

- Seja claro e conciso. O excesso de comentários pode dificultar a compreensão do código.
- Use inglês, para maior manutenibilidade do código.

Mantenha um estilo de indentação consistente

Razão

- Legibilidade.
- Evitar “erros bobos”.

Exemplo

```
if (i < 0) error("negative argument"); // bad

if (i < 0) // good
    error("negative argument");
```

Evite codificar informações de tipo em nomes

Razão

- Se os nomes refletirem tipos em vez de funcionalidade, será difícil alterar os tipos usados para fornecer essa funcionalidade.
- Se o tipo de uma variável for alterado, o código que a utiliza terá que ser modificado.
- Minimize conversões não intencionais.

Exemplo

```
void print_int(int i);
void print_string(const char*);

print_int(1); // repetitive, manual type
                 matching
print_string("xyzzy"); // repetitive,
                          manual type
                          matching
```

```
void print(int i);
void print(string_view); // also works on
                            any string-like
                            sequence

print(1); // clear, automatic type matching
print("xyzzy"); // clear, automatic type
                    matching
```

Torne o comprimento de um nome aproximadamente proporcional ao comprimento do seu escopo

Razão

- Quanto maior for o escopo, maior será a chance de confusão e de conflito de nomes.

Exemplo

```
double sqrt(double x); // return the square root of x; x must be non-negative

int length(const char* p); // return the number of characters in a zero-terminated C-style
                           string

int length_of_string(const char zero_terminated_array_of_char[]) // bad: verbose

int g; // bad: global variable with a cryptic name

int open; // bad: global variable with a short, popular name
```

Use um estilo de nomenclatura consistente

Razão

- A consistência na nomenclatura e no estilo de nomenclatura aumenta a legibilidade.

Exemplo

```
// ISO standard
int my_map(double x);

// ISO standard by Stroustrup
int My_map(double x);

// CamelCase
int MyMap(double x);
```

Use *ALL_CAPS* apenas para nomes de macros

Razão

- Para evitar confundir macros com nomes que obedecem a regras de escopo e tipo.

Exemplo

```
void f()
{
    const int SIZE{1000}; // Bad, use 'size' instead
    int v[SIZE];
}

enum bad { BAD, WORSE, HORRIBLE }; // BAD
```

Prefira nomes *underscore_style*

Razão

- O uso de sublinhados para separar partes de um nome é o estilo C e C++ original e usado na Biblioteca Padrão C++.

Observação

- Esta regra é um padrão para ser usado somente se você tiver escolha. Muitas vezes, você não tem escolha e deve seguir um estilo estabelecido para obter consistência. A necessidade de consistência supera o gosto pessoal.

Torne os literais legíveis

Razão

- Legibilidade.

Exemplo

```
auto c = 299'792'458; // m/s2
auto q2 = 0b0000'1111'0000'0000;
auto ss_number = 123'456'7890;

auto hello = "Hello!"s; // a std::string
auto world = "world"; // a C-style string
auto interval = 100ms; // using <chrono>
```

Use espaços com moderação

Razão

- Muito espaço torna o texto maior e distrai.

Exemplo

```
#include < map >
int main(int argc, char * argv [ ])
{
    // ...
}
```

```
#include <map>
int main(int argc, char* argv[])
{
    // ...
}
```

Use uma ordem de declaração de membro de classe convencional

Razão

- Uma ordem convencional de membros melhora a legibilidade.

Exemplo

```
class X {
public:
    // interface
protected:
    // unchecked function for use by derived class implementations
private:
   // implementation details
};
```

Use o *layout* derivado de K&R

Razão

- Este é o layout C e C++ original. Preserva bem o espaço vertical. Ele distingue bem diferentes construções de linguagem (como funções e classes).

Exemplo

```
struct Cable {
    int x;
    // ...
};

double foo(int x) {
    if (0 < x) {
        // ...
    }

    switch (x) {
    case 0:
        // ...
        break;
    case amazing:
        // ...
        break;
    default:
        // ...
        break;
    }

    if (0 < x)
        ++x;

    if (x < 0)
        something();
    else
        something_else();

    return some_value;
}
```

Use o *layout* de declarador estilo C++

Razão

- O *layout* no estilo C enfatiza o uso em expressões e gramática, enquanto o estilo C++ enfatiza os tipos.

Exemplo

```
T& operator[](size_t); // OK
T &operator[](size_t); // just strange
T & operator[](size_t); // undecided
```

Evite nomes que sejam facilmente mal interpretados

Razão

- Legibilidade. Nem todo mundo possui telas e impressoras que facilitam a distinção de todos os personagens. Confundimos facilmente palavras com grafia semelhante e palavras com grafia ligeiramente incorreta.

Exemplo

```
int oO01lL = 6; // bad

int splunk = 7;
int splonk = 8; // bad: splunk and splonk are easily confused
```

Não coloque duas declarações na mesma linha

Razão

- Legibilidade. É realmente fácil ignorar uma declaração quando há mais em uma linha.

Exemplo

```
int x = 7; char* p = 29; // don't
int x = 7; f(x); ++x; // don't
```

Declare (apenas) um nome por declaração

Razão

- Uma declaração por linha aumenta a legibilidade e evita erros relacionados à gramática C/C++.
- Também deixa espaço para um comentário mais descritivo no final da linha.

Exemplo

```
char *p, c, a[7], *pp[7], **aa[10]; // don't
```

Exceção

```
auto [iter, inserted] = m.insert_or_assign(k, val);
if (inserted) { /* new entry was inserted */ }
```

Exemplo

```c
double scalbn(double x, int n);   // OK: x * pow(FLT_RADIX, n); FLT_RADIX is usually 2

double scalbn( // better: x * pow(FLT_RADIX, n); FLT_RADIX is usually 2
    double x, // base value
    int n // exponent
);

// better: base * pow(FLT_RADIX, exponent); FLT_RADIX is usually 2
double scalbn(double base, int exponent);
```

Exemplo

```
int a = 10, b = 11, c = 12, d, e = 14, f = 15;
```

Não use *void* como tipo de argumento

Razão

- É verboso e necessário apenas para compatibilidade com C.

Exemplo

```
void f(void); // bad
void g(); // better
```

Use a notação convencional de *const*

Razão

- A notação convencional é familiar aos programadores
- Consistência em grandes bases de código.

Exemplo

```
const int x = 7; // OK
int const y = 9; // bad

const int *const p = nullptr; // OK, constant pointer to constant int
int const *const p = nullptr; // bad, constant pointer to constant int
```

Use o sufixo *.cpp* para arquivos de código e *.h* para arquivos de interface

Razão

- É uma convenção de longa data.
- Mas a consistência é mais importante, então se o seu projeto usa outra coisa, siga isso.

Regras de funções

Consultar

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-functions

Regras de limites

Consultar

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-bounds

Regras de tempo de vida

Consultar

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-lifetime

Regras de tipos

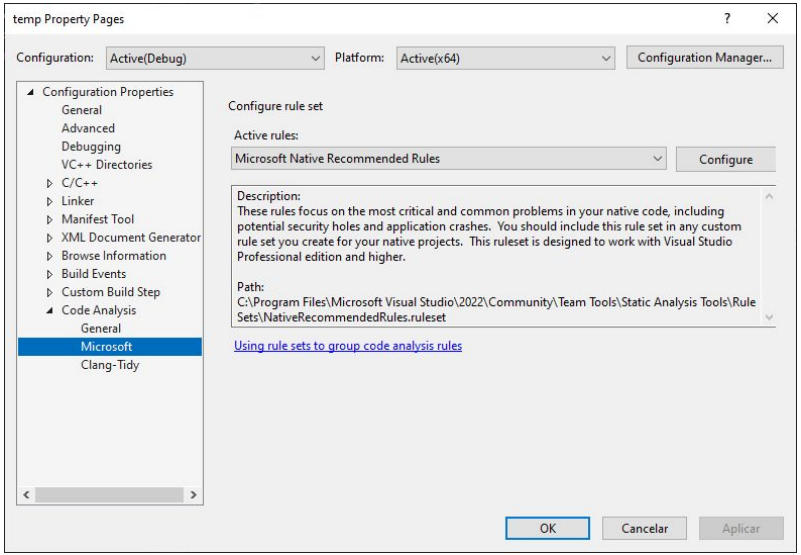
Consultar

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-concrete

Ferramentas de apoio

CppCoreChecker

- A análise de código do compilador C++ da Microsoft contém um conjunto de regras destinadas especificamente à aplicação das Regras Básicas do C++.



Referências

C++ Core Guidelines

https://isocpp.org/guidelines