Capítulo 9

Programação avançada

Se você chegou até aqui fazendo todos os exercícios que encontrou no caminho então está em condições de dar o salto final que este capítulo lhe vai propor. Esta lição é uma espécie de *diploma* junto com um convite para continuar... afinal, todo diplomado deve continuar estudando.
Vamos construir dois programas:

- O menu de opções de um programa para fazer gráficos de figuras geométricas (esperamos que você o termine e nos envie uma cópia),
- Um programa para fazer *aritmética* com números complexos.

Os dois programas ficarão imcompletos, mas, já fizemos isto nos capítulos anteriores para envolvê-lo diretamente no trabalho. Vamos deixá-lo *a um passo* de prosseguir para programação avançada que deve ser feita com $\mathcal{C}$++. Não se esqueça, $\mathcal{C}$++ contém $\mathcal{C}$.

9.1 Continuar se aprofundando em $\mathcal{C}$

Dissemos que que continuar se aprofundando em $\mathcal{C}$ certamente seria estudar $\mathcal{C}++$. Vamos discutir isto aqui. Observe que qualquer livro sobre $\mathcal{C}++$, tem pelo menos uma 200 páginas, portanto seria uma desonestidade querer lhe dizer que ao final deste capítulo você saberá programar nesta nova linguagem. Apenas esperamos lhe mostrar que vale a pena considerá-la.

A linguagem de programação $\mathcal{C}++$, é, em tese, uma outra linguagem de programação, apenas ela *entende* $\mathcal{C}$ ou ainda, com mais precisão, $\mathcal{C}++$ *extende* $\mathcal{C}$, contendo todos os comandos de $\mathcal{C}$ e além disto tem novas estruturas de programação.

Os exercícios abaixo vão explorar estes fatos.

Exercícios: 51 $\mathcal{C}++$ *é uma extensão* $\mathcal{C}$

1. *Execute na linha de comandos:*

   ```
   c++ -Wall -oprog1 integral.c
   ```

   e rode `prog1`. *Este é o executável criado com o compilador do* $\mathcal{C}++$ *a partir do código* `integral.c` *que é um programa escrito em* $\mathcal{C}$.

2. *Execute na linha de comandos:*

```
gcc -Wall -oprog2 integral.c
```

e rode `prog2`*. Este é o executável criado com o compilador do* `gcc` *a partir do código* `integral.c`*.*

3. *Execute na linha de comandos:*

```
c++ -Wall -oprog3 integral.cc
```

e rode `prog3`*. Este é o executável criado com o compilador do* $\mathcal{C}++$ *a partir do código* `integral.cc` *que é um programa "escrito" em* $\mathcal{C}++$*.*

4. *Executando (em Linux)*

```
ls -la prog* | more
```

você vai poder verificar que

$$\texttt{prog2} < \texttt{prog1} < \texttt{prog3}$$

em que "`prog2` $<$ `prog1`*" significa que o código do* `prog2` *é menor do que o código do* `prog1`*, e procure encontrar uma justificativa para este resultado.*

5. *Leia o programa* `integral.cc` *e veja quais são as notáveis diferenças entre um programa em* $\mathcal{C}++$ *ou em* $\mathcal{C}$*.*

O último exercício sugere que a substituição de `printf()` por `cout` *transformou* um programa em $\mathcal{C}$ num programa em $\mathcal{C}++$ o que seria uma grande simplificação.

Na verdade o que se costuma fazer é criar pequenas funções em $\mathcal{C}$ para usá-las em programas escritos em $\mathcal{C}++$.

Na última parte deste capítulo você vai ver dois exemplos de como isto se faz.

A justificativa para os tamanhos dos programas é a seguinte: `prog2` é o menor porque, naturalmente, é um programa em $\mathcal{C}$ e o compilador `gcc` está otimizado para compilar os programas nesta linguagem.

Consequentemente `prog1` ficou um pouco maior uma vez que o compilador do $\mathcal{C}++$ foi chamado para compilar um programa escrito em $\mathcal{C}$.

Finalmente usamos o compilador do $\mathcal{C}++$ para compilar uma *mistura* e o resultado foi um executável um pouco maior, mas que também pode ser executado.

Lição: evite as misturas...

9.1.1 A linguagem $\mathcal{C}++$

Você talvez já tenha ouvido falar de programação orientada a objeto. Esta é uma característica fundamental de $\mathcal{C}++$ e de algumas linguagens de programação como

- `Python`,
- `Java`.

Estas são algumas das linguagens de programação voltadas para *programação orientada a objetos*. Existem dezenas delas.

$\mathcal{C}++$ é uma linguagem bastante popular mas que perdeu um pouco de terreno para `Java` que é uma linguagem mais simples e sob alguns aspectos mais segura que $\mathcal{C}++$. Entretanto $\mathcal{C}++$ é extremamente rápida, de 20 a 30 vezes mais rápida que `Java` segundo a maioria dos analistas.

Tres pontos levam `Java` a se sobrepor a $\mathcal{C}++$,

- simplicidade,
- segurança,
- e o poder econômico de uma grande firma.

A simplicidade tem aspectos negativos, e a consequêcia disto é que `Java` está crescendo em complexidade de modo que este aspecto (*positivo*) tende a desaparecer.

A segurança dentro de $\mathcal{C}++$ pode ser alcançada se você aprender a programar bem. Lembre-se do que já dissemos, repetindo o que dizem outros autores: *um programa mal escrito pode ser tão ruim que é melhor você re-aprender a programar e fazer outro programa em vez de tentar corrigir o programa.* Se você aprender a programar bem, a segurança em $\mathcal{C}++$ pode ser facilmente atingida e seus programas serão rápidos e confiáveis como há muitos rodando por aí.

O poder da grande firma aos poucos vai se diluindo frente a beleza, a coerência do software livre e de domínio público de onde vêm $\mathcal{C}++$ e `Python`. Embora o Java também já tenha sido tomado pelo software livre... não é a tôa que uma grande multinacional chama o *software livre* de cancer, porque vai terminar por comê-la... é, a *liberdade* é um *cancer*, para o poder.

Mas, é claro, se você optar por `Java`, não estará mal servido e o que aprender aqui sobre $\mathcal{C}++$ lhe vai ser útil para programar bem em `Java`.

Seguindo o esquema que nos guiou até aqui, construiremos um exemplo. Vamos retomar o programa `menu.c` e construir, a partir dele, um novo programa `menu.cc`. Nesta construção, e nos comentários, lhe mostraremos um pouco de $\mathcal{C}++$ e da leveza que é programar *orientado a objeto*.

Existe uma dificuldade inicial, devido a abstração deste novo estilo de programação, mas vamos lhe mostrar que vale a pena o salto.

9.1.2 Programação orientada a objeto

Primeiro uma advertência: *leia esta seção sem levá-la muito a sério, rapidamente. Passe logo para a seguinte em que vamos construir a prática. Depois de ler a próxima seção volte para uma leitura mais detalhada desta, e verá que ela ficou mais clara e, inclusive, lhe explicou melhor o que aconteceu na próxima.*

Observação: 38 *Abstração em programação*

A palavra abstração *é usada com o seu sentido literário exato, consiste em produzir não exatamente* `programas` *mas* `modelos de programação`*. Volte posteriormente para ler este texto, quando tiver lido e rodado o primeiro programa* orientado a objeto*.*

Há muitas "rotinas"que se repetem dentro de uma "tarefa", então a idéia de programar orientado a objeto *consiste em captar tais rotinas e os seus objetivos e colocar tudo isto numa única* `cápsula` *que pode ser chamada de qualquer lugar de dentro da "tarefa".*

A palavra `encapsular` *faz parte do jargão de* programação orientada a objeto *e significa isto,* construir o modelo constituido *de* `variáveis`*, agora chamadas de* `objetos`*, e as funções que as alteram, agora chamadas de* `métodos` *.*

Em $\mathcal{C}++$ `encapsulamos` *os* `objetos` *e os* `métodos` *em* `classes` *que são os modelos que serão depois copiados, como a natureza faz com as moléculas de DNA...*

Leia os exemplos e depois volte a ler este texto.

Programar *orientado a objeto* exige algum aprofundamento na concepção do que é um programa. Os programas ficam mais *abstratos* no sentido de que, em vez de fazermos um programa, criamos uma *classe de programas.* Nosso comportamento nos capítulos anteriores conduziu a isto aos poucos quando insistentemente falamos de modularização dos programas. Um programa *orientado a objeto* em qualquer linguagem tem a seguinte estrutura geral:

```
# include sistema.h (1)
class sistema programa; (2)
int main() (3)
{
programa.init() (4)
programa.abre(programa.dados) (5)
programa.leia(programa.coisa) (5)
programa.faça(programa.coisa) (5)
programa.escreva(programa.coisa) (5)
programa.fecha(programa.dados) (5)
return 0; (6)
}
```

Explicações

1. #include sistema.h é o *método* em $\mathcal{C}$ para incluir a leitura de bibliotecas. Aqui estamos supondo que no arquivo `sistema.h` esteja a classe que precisamos.

   Esta linha traz para a memória todo o conteúdo da "classe" mencionada. Na classe `sistema` estão definidas todas as funções necessárias para comunicação com o sistema e os tipos de dados relacionados com os periféricos.

   Por exemplo, `cout` é um método definido na biblioteca `iostream.h` que se ocupa da saída de dados. Lá também está definida `cin`, que é um método para leitura de dados.

Na linguagem de *orientação a objetos* as funções agora recebem o nome de *métodos*. Quando você constrói uma classe, que é o protótipo de um *objeto*, você também define os métodos que devem alterar as variáveis desta classe. Então `cout, cin` são dois métodos definidos em `iostream.h`.

O sistema fica assim bastante protegido e claramente concebido. Tudo de um determinado *objeto* fica junto com ele, o que torna fácil a manutenção e a reutilização.

2. `class sistema programa`

   é equivalente a uma definição de *tipo de dado*, aqui muito mais poderosa porque cria um *objeto* a imagem da classe `sistema`, quer dizer, tendo todos métodos e variáveis definidos no *protótipo* `sistema`. É o que se chama tecnicamente de *herança*.

   `programa` herda as propriedades de `sistema`. .

   Usa-se também a expressão: `programa` é uma *instância* de `sistema`. Instância é um sinônimo de exemplo ou de representação.

   Nem tudo de `sistema` será herdado por `programa`. O código pode estabelecer restrições declarando partes que sejam `públicas` ou `privadas`. Isto é um detalhe a ser visto mais a frente. Então `programa` herda tudo que for declarado como público da classe `sistema`

3. Como em $\mathcal{C}$ sempre há uma funçãoprincipal.

4. O método `init()`, abreviado do inglês, `intitialization` tem a função de criar as variáveis que vão ser herdadas com valores apropriados. Da mesma forma como se "*inicializam* variáveis, há também métodos para destruí-las, que não vamos discutir aqui.

   Em geral `programa.init()` é o primeiro item de `main()` e que serve para inicializar as variáveis herdadas. Isto em geral é muito importante, mas pode não ser necessário em um determinado programa.

5. Aqui estamos supondo que em sistema foram definidos os métodos

   `abre(), leia(), escreva(). fecha()`

   e agora `programa` herdou estes métodos. A forma como eles devem ser usados é com o operador "ponto"que hoje é universalmente usado

   - em redações de textos técnicos. As rubricas dos planejamentos indicam a precedência de rubricas com sufixos indicados pelo "ponto";
   - na definição dos nós da Internet;

   Assim `programa.abre()` é um método herdado por `programa` definido em `sistema`.

   O método `abre` definido em `sistema` está possivelmente associado a um arquivo em disco e agora este arquivo se torna a entrada ou a saída padrão. Ao final ele é fechado sem erro de fechar outro arquivo que não interessava.

Pode haver até vários arquivos abertos sendo manipulados por outros ramos do programa num sistema multi-tarefa em uso em vários computadores, *tudo sob controle*, será fechado quando terminar de ser utilizado pela `instância` de sistema que foi chamada.

```
programa.leia(programa.coisa)
```

quer dizer que existe um *método* em `sistema` com o nome `leia()` e que vai ser aplicado a uma variável que em `sistema` tem o identificador `coisa`.

6. E, como é imperativo, toda função termina com um `return`, e "sempre"`main() produz um inteiro ao final de suas operações.`

Usamos verbos e frases que não pertencem a nenhuma linguagem de programação, mas que se parecem com $\mathcal{C}$, que traduzem o que acontece em qualquer uma delas. Quando você ver um programa em $\mathcal{C}++$ irá logo reconhecer o que fizemos acima.

Veja as vantangens desta *complicação...*

- Os "métodos" definidos em `sistema` se ocupam da segurança e dos detalhes.

- Quando você quiser atualizar um sistema, tudo deve ser feito na `classe sistema`. Se ela for *bem feita* o restante simplesmente herda as modificações sem grandes traumas, algumas vezes sem que seja necessário mexer em coisa alguma dos programas que herdam *métodos* e *objetos* das classes. Outras vezes pequenas modificações são necessárias.

  Vou lhe dar um exemplo errado (que novidade)!

  Observe a função `limpa_janela()`, definda em `ambiente.h`. Depende de que `ambiente.h` você vai ler... se for no diretório do `DOS` ela usa uma função apropriada para este sistema. Se for em `Linux`, vai usar outra.

  Alteramos apenas a biblioteca `ambiente.h` e todos os programas, usando `limpa_janela()` funcionam, podem ser usados nos dois sistemas.

  Por que o exemplo está errado ? porque `ambiente.h` não é uma *classe* ! Mas a idéia é a mesma. Basta transformar `ambiente.h` em uma classe e o exemplo fica perfeito.

- Compare o código acima com o conteúdo de `menu.cc` e com o código de `menu.c` e se convença que `menu.cc` é mais poderoso. Com o tempo você vai se convencer que também é mais simples.

- Aparentemente uma complicação nova apareceu, aumentou a quantidade de texto para ser digitada: `programa.qualqueroutracoisa()`. Sempre temos que digitar "`programa`". Isto não representa problema: simplesmente não digite "`programa`". Escreva sua ideia. Quando compilar, o *compilador* vai lhe dizer que `qualqueroutracoisa()` não existe... Ai você digita um "sinal qualquer" antes de todos os métodos e variáveis e faz uma troca automática deste sinal por "`programa.`".

  Você evitou de digitar "`programa.`"!

- Finalmente, quando for executado

  ```
  programa.fecha(programa.dados)
  ```

  não haverá mais nada zanzando pelos sistema ligado ao disco e e nem ao arquivo "dados". Segurança total.

Se você estiver achando este texto difícil, *muito abstrato*, lembre-se da observação feito ao começo. Era para fazer uma leitura rápida e passar às seções seguintes em que exemplos vão ser construídos, e depois voltar para uma segunda leitura.

9.2 O programa `menu.cc`

Vamos reproduzir a idéia do programa `menu.c` que é um programa (incompleto, verdade!) escrito em $\mathcal{C}$ com os novos conceitos de *programação orientada a objetos*.

9.2.1 Construção da idéia

Leia o programa `menu.c` e rode este programa para ver onde queremos chegar. Primeiro rode e depois leia.

Primeiro um pouco de propaganda. Leia os dois programas

- `menu.cc` e
- `menu.c`

e vamos compará-los sem entrar nos detalhes técnicos da comparação.

- Veja que a função `executa()` nos dois programas é a mesma.
- Veja que a função `main()` em um dos programas é mais simples. É quase uma lista de frases que dizem o que vão fazer. Costumo chamar esta função `main()` de `script` contrariando um pouco o linguajar técnico de computação em que esta palavra tem um significado especial, aqui usamos no espirito com que se usa, no do teatro:

  `main()` em `menu.cc` é um `script` que chama os atores, na ordem certa, para executarem o espetáculo. Os atores já sabem o que fazer.

Ainda dentro do espirito de propaganda, no bom sentido, (se é que propaganda pode ter um sentido bom...), compare os programas

`menu.cc` e `menu_mil.cc`.

Obviamente, veja o nome, `menu_mil.cc` é uma[1] versão muito melhorada do anterior. Rode primeiro para se convencer deste detalhe:

c++ -Wall -oprog menu.cc

e rode `./prog`.

Depois

[1] mil é abreviação de milenium...

c++ -Wall -oprog menu_mil.cc

e rode `./prog`.

Agora leia os dois programas, `menu.cc` e `menu_mil.cc`.

A classe que está definida no arquivo `ambi.h`, que é a versão para $\mathcal{C}++$ de da "velha" `ambiente.h`.

Foi fácil melhorar o programa `menu.cc` e dar-lhe a cara de `menu_mil.cc`, apenas incluindo a bilioteca `ambi.h` onde estão definidas as funções

`limpa_janela(), apetecof(), copyleft()`

como `ambi.h` é uma classe pode ser utilizada com as modificações descritas anteriormente, usando o "ponto" para caracterizar a origem dos métodos e neste caso chamariamos

`amb.limpa_janela(), amb.apetecof(), amb.copyleft()`

em que `amb` seria uma `instância` da classe `ambi.h`.

Vamos aos passos na transformação de `menu.c`, em `menu.cc` usando a classe `informacao`. Leia o arquivo `ambi.h` em que a classe está gravada.

- Observe a estrutura de uma classe, não entre nos detalhes agora.
  - Começa com a palavra chave `class`

    seguida do nome da classe `informacao.`
  - A palavra-chave "`public:`", terminada com dois pontos, marca uma área em que estão definidas *variáveis* e *métodos* que poderão ser utilizados por qualquer *instância* (exemplo) da classe, os objetos.
  - Segue a lista do que é público e depois poderia haver uma lista de variáveis ou métodos privativos da classe numa área etiquetada com a palavra-chave "`private:`".
  - Observe que tudo isto está entre chaves e ao final um ponto e vírgula. Nesta área os métodos se encontram apenas anunciados, a implementação vem depois.
  - A implmentação dos métodos vem logo abaixo, cada método iniciado com a palavra-chave `inline`. Veja a sintaxe:

    ```
    inline void informacao::init()
    ```

    que copiamos do método `init()` e que podemos repetir de forma mais abstrata (geral):

    ```
    inline tipo informacao::nome()
    ```

    em que `informacao` é o nome da classe.

Agora é usar a classe como em `menu_mil`.

9.3 Números complexos

Vamos mostrar como criar uma `classe` chamada `complexo` que define um número complexo e depois fazer uso desta classe num programa.

9.3.1 Que é número complexo

Os números complexos são aqueles números que aparecem com as equações do segundo grau quando o *delta* for negativo, as equações que *não tem raizes reais*.

Isto dá origem a números do tipo

$$z = 3 + 2i = a + bi; \tag{9.1}$$
$$\Re(z) = 3 = a; \tag{9.2}$$
$$\Im(z) = 2 = b; \tag{9.3}$$

Ou seja, um número complexo é um par de números reais $(a, b) \equiv a + bi$.

Quando somamos dois números complexos, usamos a regra da álgebra "soma de termos semelhantes". Poristo somamos *parte real* com *parte real* e *parte imaginária* com *parte imaginária*.

Quando multiplicamos dois números complexos, usamos também a mesma regra da álgebra "soma de termos semelhantes", mas antes, multiplicamos todos os termos entre si (usando a propriedade distributiva), depois somamos o que for semelhante.

Veja na figura (fig. 9.1) página 200,

Para criar esta *classe de objetos* temos que ter "pares ordenados" e pelo menos os seis métodos:

- Método para extrair a parte real
- Método para extrair a parte imaginária
- Método da soma
- Método do produto
- Método do produto por um escalar
- Método para calcular o módulo

Em vez de diretamente criar uma classe, vamos primeiro fazer um programa em $\mathcal{C}$ que execute estas tarefas todas, depois vamos transformá-lo em classe de modo que nunca mais precisemos que repetir a tarefa inicial. Quando precisarmos de números complexos faremos uma *herança*.

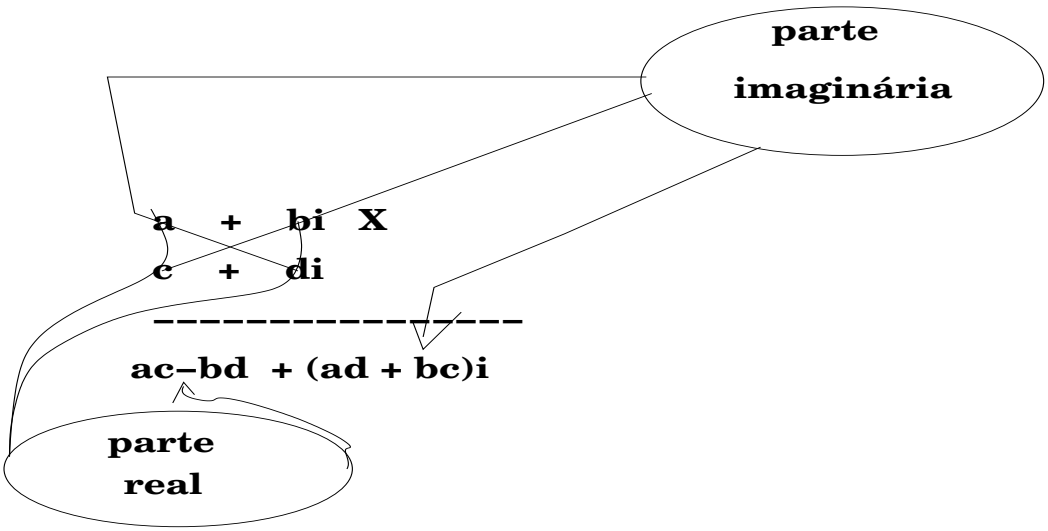

Figura 9.1: O produto de números complexos: parte imaginária se obtem em cruz

9.3.2 O programa em $\mathcal{C}$

O programa em $\mathcal{C}$ é o trabalho experimental em que se baseia a construção da classe, e aqui está uma das razões pelas quais você deve aprender $\mathcal{C}$...

A lista de exercícios seguinte o conduz a construção do programa `complexos01.c`. Quer dizer que este programa contém as soluções dos exercícios.

Exercícios: 52 *Construção da álgebra dos complexos*

1. *Faça um programa que entenda o que é um número complexo e possa explicitar sua* parte real *e sua* parte imaginária*. Solução* `complex01.c`

2. *Extenda o programa* `complex01.c` *para que ele faça a soma de dois números complexos. Solução* `complex02.c`

3. *Construa um programa que calcule a soma e o produto de dois números complexos. Solução* `complex03.c`

4. *Transfira todas as funções de* `complex03.c`*, exceto* `main()` *para a biblioteca* `complex.h`*, coloque a diretiva de compilação adequada para leitura desta biblioteca e roda o programa. Solução* `complex04.c, complex04.h`

5. *Complete a biblioteca complex.h para que sejam feitas as operações de multiplicação por um escalar e cálculo do módulo de um número complexo. Altere a função* `entrada()` *e roda o programa. Solução* `complex.c, complex.h`

6. *Verifique que os programas* `complexos01.c` *e* `complexos.c` *são apenas versões de* `complex.c` *sem uso da bibliteca* `complex.h`*. Tome uma posição sobre qual é o melhor dos métodos.*

9.3.3 Construção de `complexo_milenium_plus.cc`

Neste momento não precisamos mais buscar um longo atalho para chegar no nosso objetivo. Talvez já tenha lido umas duas vezes a seção inicial deste capítulo, você já deve ter compreendido claramente que, programar orientado a objeto consiste em criar

- o `objeto` um conjunto de dados, ou de tipos de dados, que descrevem o `objeto`;
- os `métodos` um conjunto de funções que, `métodos`, que manipulam o `objeto` ou partes do `objeto`

`encapsulados` numa `classe` que poderá ser re-utilizada em diversas *instâncias*.

O que faremos agora é simplesmente transformar a biblioteca `complex.h` em `complexO.h` e o programa `complex.c` em `complexO.cc`.

Leia os dois árquivos `complexO.h`, `complexO.cc`, e depois rode

c++ -Wall -oprog complexO.cc
./prog

para ver que o efeito é exatamente o mesmo que

gcc -Wall -oprog complexO.cc
./prog

mas que agora você pode reaproveitar o conteúdo da estrutura `Complexo` com mais facilidade. Os exercícios lhe mostrarão como fazê-lo.

Exercícios: 53 *Números complexos orientados a objeto*
Construção de uma classe derivada

1. *Quando definimos os números complexos em* `complexO.h`*, nos esquecemos de definir módulo. Defina uma nova classe,* `algebra`*, que* `herde` *o contúdo de* `Complexo` *e tenha o método val_abs_comp() para calcular o módulo dos complexos. Solução* `algebra.h`
2. *Altere o menu do programa* `complexO.cc` *para nele incluir a nova possibilidade do cálculo de módulos de números complexos, redija um novo arquivo, não altere o existente. Solução* `complexo_milenium.cc`.
3. *Sem solução neste livro... Construa uma classe menu retirando de*
   `complexo_milenium`
   todas as funções de maneiras que fique apenas a `main()`*. Fica como desafio para o leitor, construir*
   `complexo_milenium_platinum.cc`
   a versão definitiva.

Neste momento você pode ver a construção de `algebra.h` como uma *correção* feita à classe `complexo` definida em `complexO.h`.

Mas não é bem assim o nosso objetivo. Queremos que veja que você pode ter classes com menos propriedades, *mais gerais*, e com elas construir *classes derivadas* que tem mais propriedades. Se convença: *quanto maior for o*

número de propriedades de uma classe, menos probabilidade vai haver para sua re-utilização.

O desafio colocado como último exercício, no bloco anterior, mostra como podemos chegar a uma situação limite: o programa `complexo_milenium_plainum.cc` definitivo que nunca será alterado. Todas as alterações serão feitas apenas em `classes` específicas:

- Novas propriedades dos números complexos, serão incluídas na classe complexo (observe que a classe `algebra` é desnecessária, ela foi construída apenas para dar um exemplo de classe derivada). As alterações, as novas propriedades que queiramos incluir nos números complexo, podem ser feitas na classe `complexo`.

- Alterações no menu comentários, enfim, comunicação com o usuário, serão feitas na `classe menu`

- Obviamente o trabalho fica departamentalizado, mas é bom não esquecer que toda alteração na classe `complexo` implica em possível alteração na classe `menu`, por exemplo, o acréscimo da possibilidade do cálculo do módulo de números complexo nos obrigou a alterar o menu.

`complexo_gold` Claro, quando precisarmos vender a próxima versão *super melhorada* do nosso produto, ainda temos referências como `gold` e `plus` para garantir ao cliente que vale a pena investir no novíssimo produto... Pensamos na próxima edição deste livro lançar `complexo_gold_XGL` em que o desafio estará resolvido. Aguarde...