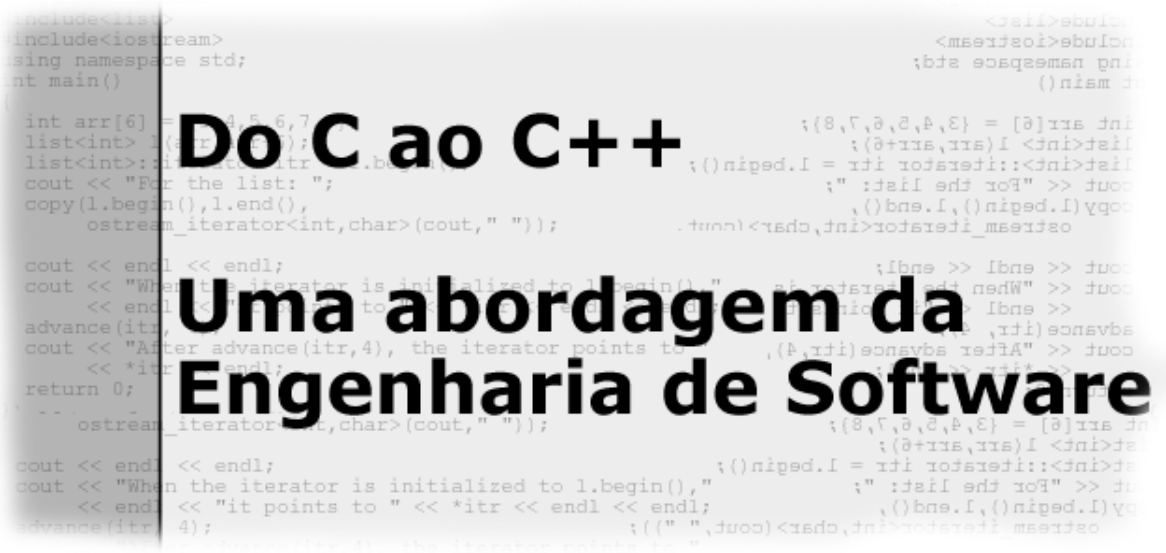


Tiago Barros – 2002

Índice Analítico

Capítulo I

Engenharia de Software Estruturada

Engenharia de Software Estruturada: o uso de funções

Nos primórdios da programação, os programas eram criados seguindo uma única seqüência, ou seja, tínhamos programas completos escritos sem nenhuma estruturação. A idéia de função, código que pode ser chamado várias vezes em diferentes partes do programa e com diferentes parâmetros, trouxe uma grande estruturação à programação. Neste capítulo veremos como utilizar funções, e na seção prática, como estruturar nosso programa em funções. Para isto, começaremos com uma visão geral do básico de C++, para que tenhamos o suporte necessário para utilizar as funções.

O C++ básico

Tipos: estruturas, uniões e tipos enumerados

Como você já deve saber, quando nos referimos a tipo, estamos falando do tipo da variável (porção de memória utilizada para guardar dados). Os tipos diferem no formato como os dados são armazenados e na quantidade de memória utilizada. Também diferem na semântica, pois embora possam ter a mesma quantidade de memória, dois tipos podem guardar informações de significado diferente. O C++ é uma linguagem fortemente tipada, o que significa que as operações sobre variáveis são realizadas apenas entre variáveis do mesmo tipo, ou de tipos compatíveis. A linguagem C++ define vários tipos próprios, chamados tipos básicos. Suas características, que você já deve conhecer, podem ser encontradas no Apêndice A, de forma que não prolongarei aqui uma discussão sobre declaração e utilização de tipos básicos.

Declarando arrays

Array nada mais é do que um conjunto de variáveis do mesmo tipo que são acessadas pelo mesmo nome e um índice.
Para declarar um array de **char**, por exemplo, fazemos:

```
char array_de_char[20];
```

O valor entre parênteses é a quantidade de posições que o array possui, ou seja, `array_de_char` possui 20 variáveis do tipo **char**. Veremos a importância dos arrays e como utilizar arrays multidimensionais e ponteiros com arrays mais adiante, neste capítulo.

Definindo novos tipos

Além dos tipos de dados simples como **int**, **float**, **double**, **char**, ...; podemos definir os nosso próprios tipos de dados. Estes tipos são definidos através das palavras-chave **struct**, **union** e **enum** (e posteriormente, como veremos, **class**).

Estruturas

As estruturas de C++ funcionam como tipos compostos, em que se pode guardar diversas informações agrupadas em um mesmo nome. Por exemplo, para definir o tipo composto bola, utilizamos **struct** de acordo com a sintaxe abaixo:

```
struct TBall
{
    int x;
    int y;
```

```
    int radius;
    TColor color;
};
```

A bola que definimos tem, portanto, uma posição (x, y), um raio e uma cor. Se criarmos uma variável do tipo bola, que acabamos de definir, podemos acessar as características da bola da seguinte forma:

```
TBall bola;
bola.x = 10;
bola.y = 15;
bola.radius = 5;
bola.color = clBlue;
```

Note, que TColor não é um tipo de dados simples, de forma que podemos criar tipos de dados compostos a partir de outros tipos compostos também.

Uniões

Uma união é um tipo composto parecido com uma estrutura. A diferença é que na união, os diversos "membros" compartilham a mesma área de memória, ou seja, alterando um membro da união, na realidade estamos alterando todos. As uniões são utilizadas quando queremos acessar uma mesma informação de diversas formas, sobre a ótica de diversos tipos de dados, ex:

```
union TCharInt
{
    char c;
    short i;
};

int main()
{
    TCharInt var;

    var.c = 'A';
    cout << "O codigo ASCII da letra A e " << var.i;
    cin.get();

    return 0;
}
```

No código acima, nós atribuímos uma constante do tipo **char** à união **var**. Posteriormente, acessamos a união **var** através do seu membro **i**, do tipo **short int**. Como a letra A é representada internamente através do seu código numérico, quando acessamos a variável como um tipo **short int**, teremos como retorno o valor numérico (do código ASCII) da letra A. Portanto, o exemplo acima exibirá:

```
O codigo ASCII da letra A e 65
```

Tipos enumerados

Os tipos de dados enumerados são utilizados para representar tipos conceituais que não estão presentes no C++ padrão. Por exemplo, para representar o tipo de dados cor, que pode assumir os valores vermelho, verde, azul, amarelo, ciano, magenta, branco e preto; poderíamos utilizar um tipo de dados enumerado:

```
enum TColor {clRed, clGreen, clBlue, clYellow, clCian, clMagenta,
clWhite, clBlack };
```

Desta forma, as variáveis do tipo TColor só aceitariam um dos valores acima, que, internamente estão representados por números, mas no código têm uma semântica que é bastante útil para quem for utilizar.

Modificadores de tipos

Em C++, podemos modificar a forma como as variáveis são declaradas e alocadas, seu escopo e a sua ligação (capacidade de um item ser usado em diferentes arquivos de um mesmo programa). Isto é feito com a utilização dos modificadores de tipo. Para entender melhor o seu funcionamento, veremos alguns conceitos básicos necessários, antes de falar dos modificadores de tipos propriamente ditos.

Escopo

Variáveis em C++ podem ter escopo local (ou de bloco) ou global (ou de arquivo). Variáveis globais ficam disponíveis a todo o código do arquivo, enquanto as variáveis locais estão disponíveis apenas dentro do bloco em que foram criadas e nos blocos internos a ele.

Classes de armazenamento

A memória de armazenamento (espaço reservado às variáveis) de um programa em C++ é dividida em três classes:

- Armazenamento automático: onde as variáveis de um bloco são alocadas e armazenadas quando o programa entra no bloco. Seu escopo é local.
- Armazenamento estático: área de armazenamento persistente. Os dados com armazenamento estático estão disponíveis durante toda a execução do programa. Isto significa que os dados estáticos não são reinicializados a cada vez que uma função é chamada, como ocorre com o armazenamento automático.
- Armazenamento livre: área de alocação de memória em tempo de execução. Toda variável alocada em tempo de execução é armazenada nesta área. É de suma importância que as variáveis criadas nesta área sejam desalocadas sempre que não forem mais ser utilizadas.

Ligação

A ligação (lincagem) se refere à capacidade de um item ser utilizado em diversos arquivos de um programa. Variáveis externas (declaradas fora de qualquer função) possuem também ligação externa, o que significa que elas ficam disponíveis a qualquer arquivo do mesmo programa. Já as variáveis de ligação interna só estão disponíveis no arquivo em que foram criadas.

Os conceitos vistos acima estão relacionados de forma não sistemática, sendo às vezes difícil de se entender claramente. Portanto, abaixo segue uma lista das possíveis associações entre classes de armazenamento, escopo e ligação:

Tipo da variável	Forma de declaração	Escopo	Ligação
Variáveis automáticas	Declaradas dentro de um bloco	Local	Interna
Variáveis externas	Declaradas fora de qualquer função	Global	Externa
Variáveis externas estáticas	Variáveis externas declaradas com a palavra-chave **static**	Global	Interna
Variáveis externas constantes	Variáveis externas declaradas com a palavra-chave **const**	Global	Interna
Variáveis estáticas	Variáveis automáticas declaradas com a palavra-chave **static**	Local	Interna

Depois de entendidos estes conceitos, vamos aos modificadores propriamente ditos.

Modificadores:

static: é utilizado para declarar variáveis na classe de armazenamento estático.
const: é utilizado para declarar constantes, que, obviamente, não podem ter o seu conteúdo alterado.
extern: é utilizado para indicar que estamos utilizando uma variável externa, que foi declarada em outro arquivo.
register: é utilizado para "pedir" ao compilador que armazene a variável em um registrador da CPU. Isto acelera o acesso à variável, mas não deve ser utilizado para todas as variáveis pois os computadores possuem geralmente registradores de 32 bits (em alguns computadores, 64 bits), que é um tamanho menor do que alguns tipos como **double**, e em número limitado.
volatile: indica que um valor de uma variável pode ser alterado por fatores externos ao código do programa, como, por exemplo, uma variável que aponte para o endereço do relógio do sistema, e que, portanto, é atualizada pelo hardware. Variáveis em programas *multithreaded*[1] que são alteradas e lidas em *threads* diferentes também devem ser declaradas como **volatile**.
mutable: indica que um membro de uma estrutura (ou classe) pode ser alterado mesmo que faça parte de uma variável declarada como constante.

Conversões de tipos

Quase sempre, em um programa, existe a necessidade de se fazer a conversão entre tipos de dados para que se possa realizar operações com variáveis de tipos diferentes. A conversão de tipos (ou *type casting*) pode ser realizada da forma implícita ou explícita.

Cast implícito

Ocorre quando fazemos conversão entre variáveis de tipos compatíveis, onde não há a necessidade de utilização de palavras-chave para informar a conversão ao compilador. Ex:

```
short int numShort = 1234;
long int numLong = 314153256;
long int result;
result = numShort + numLong;
```

Cast explícito

Utilizamos o cast explícito quando queremos "informar" ao compilador que não estamos fazendo uma conversão indesejada, e também quando os tipos não são compatíveis mas a conversão é necessária. Abaixo seguem as formas de fazer o cast explícito em C++:

(tipo): conversão explícita de tipos estilo C. É considerada antiga e obsoleta.

static_cast <tipo> (expressão): é utilizado quando queremos fazer um cast em "tempo de compilação", ou seja, não há verificação de tipos em "tempo de execução". Ex:

```
int sum;
int num = 10;
float pi = 3.14159;
```

[1] Programa com várias linhas de execução, chamadas threads, que rodam em paralelo, na ótica do programa.

```
sum = static_cast <int> (pi) + num;
```

const_cast <tipo> (expressão constante): é utilizado para retirar a "constância" de expressões. No exemplo abaixo, podemos desconsiderar o fato que const_ptr é um ponteiro constante quando atribuímos ele a um ponteiro normal (não se preocupe, veremos o uso de ponteiros mais adiante):

```
char * ptr;
const char * const_ptr;
.
.
.
ptr = const_cast <char *> (const_ptr);
```

reinterpret_cast <tipo> (expressão): o reinterpret_cast é a forma mais poderosa de conversão de tipos. Ele força a reinterpretação dos bits da expressão, de forma que os valores reais dos bits de um valor são utilizados. Devido ao seu poder, seu uso é bastante perigoso, pois fazer um reinterpret_cast de um float para um int, por exemplo, poderia resultar em um valor inteiro completamente sem sentido. Um exemplo de uso do reinterpret_cast pode ser visto a seguir:

```
pointer = reinterpret_cast <int *> (0xB0000000);
```

Este exemplo faz com que o ponteiro `pointer` aponte para o endereço de memória `0xB0000000.` Se não utilizássemos o reinterpret_cast, o exemplo acima não compilaria, pois o compilador não considera a conversão do tipo **unsigned long** para **int *** algo que se possa fazer implicitamente.

dynamic_cast <tipo> (expressão): é utilizado para fazer conversões em tempo de execução. Ele faz parte da RTTI (runtime type identification – identificação de tipos em tempo de execução) e será visto com detalhes no Capítulo V.

Laços e Condicionais

Elementos essenciais em linguagens de programação, as *instruções condicionais* (ou *instruções de seleção,* de acordo com a especificação do ANSI/ISSO C++) e as *instruções de iteração* serão abordadas nesta seção, bem como as instruções de controle de fluxo (**break**, **continue**, **exit** e **abort**).

A instrução if

É utilizada para testar expressões e tomar decisões. Se a expressão entre parênteses for avaliada como verdadeira ou diferente de zero, seu bloco de instruções correspondente será executado. Possui uma cláusula opcional, **else**, que executa um bloco de instruções caso a expressão entre parênteses seja avaliada como falsa. Podemos ter vários blocos **if-else** aninhados, por exemplo para testar uma variável sucessivamente, até encontrarmos uma correspondência. A sintaxe da instrução **if** é:

```
if (expressão)
{
    bloco de instruções que
    serão executadas caso expressão seja
    avaliada como verdadeira;
}
else
{
    bloco de instruções que
    serão executadas caso expressão seja
```

```
        avaliada como falsa;
}
```

A instrução switch

Uma alternativa para a utilização de **if-else** aninhados é a instrução **switch**. Esta instrução é utilizada para desvios múltiplos, de uma forma mais sintética. A sintaxe do **switch** é:

```
switch (expressão)
{
    case valor1:
        instrução1;
        [break;]
    case valor2:
        instrução2;
        [break;]
    case valor3:
        instrução3;
        [break;]
       .
       .
       .
    default:
        instrução_padrão;
}
```

Caso a instrução **break** não seja utilizada, o programa continuará executando as instruções do próximo **case** até que haja um **break** ou que acabe o bloco **switch**.

Os laços while e do-while

Um laço **while** executa seu bloco de instruções enquanto a sua expressão for avaliada como verdadeira (ou diferente de zero). Possui as seguintes sintaxes:

```
while (expressão)
{
    instruções;
}
```

ou

```
do
{
    instruções;
} while (expressão);
```

A diferença, é que no laço **while** puro, a expressão é avaliada antes de executar as instruções, enquanto no laço **do-while** as instruções são executadas e a expressão é avaliada no final (portanto as instruções são executadas pelo menos uma vez).

O laço for

O laço **for** é utilizado quando desejamos utilizar um índice numérico, que seja incrementado/decrementado a cada iteração. Sua sintaxe é:

```
for (inicialização; expressão_de_teste; expressão_de_iteração)
{
    instruções;
}
```

A *inicialização* é executada antes de começarem as iterações do laço. É geralmente onde inicializamos as variáveis de índice que serão utilizadas no laço. A cada iteração, a *expressão_de_teste* é avaliada e o laço é encerrado quando ela se torna falsa. Após cada execução do corpo do laço, a *expressão_de_iteração* é executada. Essa expressão é, geralmente, um incremento/decremento da(s) variável(is) de índice.

Instruções de controle de fluxo

break: é utilizada para terminar a execução de um laço ou de um bloco **switch**.
continue: faz com que a execução do programa pule para a próxima iteração de um laço, ou seja, as instruções entre **continue** e o fim do corpo do laço não são executadas.
goto *label*: transfere o fluxo do programa para a instrução seguinte a *label*. Seu uso é extremamente desaconselhado, pois torna o fluxo do programa bastante difícil de acompanhar e depurar. Sempre existe uma forma de escrever um programa sem **goto**. Portanto **não use esta instrução**.
exit: termina a execução do programa. Pode ser chamada de qualquer função, não apenas da função *main.*
abort: termina imediatamente a execução do programa, sem executar as instruções de encerramento de tempo de execução do C++ (como chamar os destrutores dos objetos globais por exemplo).

Ponteiros

As variáveis ponteiro são um dos maiores recursos de C++ pelo seu poder, mas também um dos maiores causadores de *bugs* e o terror de alguns programadores. Nesta sessão veremos que os ponteiros são um dos nossos maiores aliados e também veremos como não cair nas suas armadilhas.

Uma breve discussão sobre endereços de memória

Todas as variáveis de C++ possuem um *nome* e um *conteúdo*, e estão armazenadas em alguma *posição de memória*. No nosso programa utilizamos esses nomes para referenciar os conteúdos das variáveis. Uma outra forma de referenciar uma posição de memória é através do seu endereço direto. Para obter o endereço de elementos de memória o C++ possui o operador &, chamado operador *endereço* ou *referência*. Desta forma, podemos declarar variáveis que guardem em vez de valores de tipos, endereços de memória. Isto nos permite criar variáveis em tempo de execução e guardar o seu endereço nestas variáveis especiais, chamadas ponteiros. Veremos como declarar e manipular ponteiros nas seções a seguir.

Como declarar ponteiros

Apesar de utilizarem espaços de memória do mesmo tamanho, pois todos os ponteiros guardam endereços de memória, cada ponteiro precisa ser do mesmo tipo do dado que ele aponta, ou seja um ponteiro para inteiros deve ser do tipo **int**, enquanto um ponteiro para **double** deve ser do tipo **double**. Para declarar ponteiros, utilizamos um asterisco, *, após o nome do tipo, da seguinte forma:

```
int *ponteiro_para_int;
double *ponteiro_para_double;
```

Utilizando ponteiros

Para utilizarmos os ponteiros para apontar para uma variável, utilizamos o *operador endereço*, & :

```
int um_inteiro = 10;
double um_double = 3.14159;

ponteiro_para_int = &um_inteiro;
ponteiro_para_double = &um_double;
```

Para acessar o conteúdo das variáveis apontadas pelo ponteiro, utilizamos o operador *desreferência*, *, que também é um asterisco:

```
cout << "O valor de um_inteiro é: " << *ponteiro_para_int;
cout << "O valor de um_double é: " << *ponteiro_para_double;
```

Que nos dará como saída:

```
O valor de um_inteiro é: 10
O valor de um_double é: 3.14159
```

O operador *desreferência* faz com que, em vez de exibir o conteúdo da variável ponteiro, que é o endereço de `um_inteiro`, utilize esse conteúdo como endereço e exiba o conteúdo deste endereço, que é 10. Complicado? Não, é muito mais simples do que se imagina. Vejamos alguns exemplos de uso dos operadores & e *, e o que aconteceria em cada caso:

Exemplo 1:

```
cout << "O valor de um_inteiro é: " << &um_inteiro;
```

Apesar de passar pelo compilador, este exemplo estaria semanticamente incorreto, pois o que seria exibido é o endereço de `um_inteiro` e não o seu valor.

Exemplo 2:

```
cout << "O valor de um_inteiro é: " << *um_inteiro;
```

Este exemplo causa erro de compilação pois a variável `um_inteiro`, apesar de ter o mesmo tamanho de um ponteiro e guardar um valor numérico, não foi declarada como ponteiro, e não pode ter seu conteúdo utilizado para endereçar memória.

Exemplo 3:

```
cout << "O valor de um_inteiro é: " << &ponteiro_para_int;
```

Neste exemplo, estaríamos exibindo o endereço de `ponteiro_para_int`, e não o conteúdo do endereço que ele aponta. Se estivéssemos atribuindo `&ponteiro_para_int` para uma variável, como esta variável deveria ser declarada? Correto se respondeu: `int **ponteiro_duplo;` Pois o endereço de um ponteiro deve ser guardado em um ponteiro para ponteiro.

Exemplo 4:

```
cout << "O valor de um_inteiro é: " << *ponteiro_para_int;
```

Esta seria a forma correta de utilizar o ponteiro, e o que seria exibido seria o valor guardado no endereço guardado no ponteiro.
Como veremos mais adiante, além dos tipos básicos, podemos utilizar ponteiros para tipos compostos, objetos e funções.

Criando variáveis em tempo de execução: os operadores new e delete

Até agora, toda a memória que nós utilizamos era alocada através da declaração de variáveis, o que significa que estávamos informando ao C++ o quanto de memória que o nosso programa iria utilizar. Como vimos anteriormente, a memória do nosso programa está dividida em três classes de armazenamento: o armazenamento automático (para variáveis declaradas localmente), o armazenamento estático (para variáveis declaradas com **static** e para as variáveis globais) e o armazenamento livre (para as variáveis declaradas em tempo de execução). Portanto, podemos alocar e liberar memória da área de armazenamento livre, em tempo de execução, o que nos permite uma maior eficiência no gerenciamento de memória. Isto é feito através dos operadores **new** e **delete**.
O operador **new** retorna um ponteiro para o tipo especificado, de forma que, se quisermos criar um inteiro em tempo de execução, devemos proceder da seguinte forma:

```
int *ponteiro_para_int;
ponteiro_para_int = new int;
```

Agora você entende o uso de ponteiros, pois não fez muito sentido o seu uso como nos exemplos da seção *utilizando ponteiros*, certo?
As variáveis da área de armazenamento automático são desalocadas logo que o programa sai do seu escopo, automaticamente (daí o nome armazenamento automático). Isto não ocorre com as variáveis da área de armazenamento livre. Embora possam estar sendo referenciadas por ponteiros declarados localmente (e portanto, localizados na área de armazenamento automático), todas as variáveis criadas dinamicamente continuam a existir durante toda a execução do programa, ou até que sejam desalocadas explicitamente. Vejamos um mau uso de alocação dinâmica:

```
for (int i=0; i < 1000; i++)
{
    double *pointer;
    pointer = new double;
    pointer = i * 3.14159;
}
```

Neste exemplo, a variável `pointer` é declarada dentro do corpo do **for**, e por isso é criada na área de armazenamento automático. Então, no início da execução de cada iteração, é alocada memória para esta variável. Depois, com o operador **new**, é alocada memória (desta vez na área de armazenamento livre) para uma variável do tipo **double**, e o endereço é guardado em `pointer`, de forma a poder utilizar esta área de memória. No fim da execução da iteração, a variável pointer é desalocada. Então você deve estar se perguntando: o que acontece com a área alocada pelo operador **new**? Exatamente! Como não existe nenhum ponteiro que referencie a área alocada e como ela não foi liberada, ficará ocupando espaço até o final do programa. Desta forma, ao final deste laço, teremos 1000*8 bytes de memória (o tamanho do **double** é geralmente 8 bytes) que está alocada mas não poderá ser utilizada. Abaixo veremos o código escrito de forma correta:

```
for (int i=0; i < 1000; i++)
{
    double *pointer;
```

```
        pointer = new double;
        pointer = i * 3.14159;
        delete pointer;
    }
```

A diferença entre este exemplo e o anterior é a última instrução do corpo do laço: `delete pointer;` Esta instrução desaloca a memória alocada pelo **new**. Portanto, é de suma importância a desalocação das variáveis que não serão mais utilizadas, sob pena de causar um estouro de memória no programa. Uma observação que deve ser feita aqui é: quando utilizamos o **new**, caso ele não consiga alocar a memória correspondente, retornará um ponteiro nulo (que possui o valor zero). Portanto, sempre que alocarmos memória dinamicamente, devemos verificar se isto realmente foi feito com sucesso, comparando o ponteiro com a constante `NULL`, para que o programa não encerre abruptamente devido a erros em tempo de execução.

Ponteiros para tipos derivados

Podemos utilizar ponteiros para apontar para qualquer tipo de dados em C++, inclusive arrays, estruturas, uniões e tipos enumerados (e objetos, como veremos posteriormente). Abaixo segue um exemplo de como utilizar ponteiros com estruturas (o uso com uniões é análogo):

```
struct TBall
{
    int x,
    int y,
    int radius,
    TColor color
} ball_1, *pointer;

pointer = &ball_1;

(*pointer).x = 10;
pointer->y = 20;
```

Portanto, para acessar um membro da estrutura, basta utilizar `(*pointer)` pois isto desreferencia o ponteiro. Podemos utilizar também o operador seta, `->`. Com o operador seta, podemos utilizar o ponteiro diretamente, sem precisar desreferenciá-lo com `*`.

Ponteiros e arrays

Em C++, ponteiros e arrays são bastante semelhantes. Podemos utilizar ponteiros como se fossem arrays (com o operador colchetes, []) e o nome do array como um ponteiro para a sua primeira posição. Abaixo segue um exemplo de como utilizar ponteiros como arrays:

```
int array[5];
int *pointer;
array[0] = 10;
array[1] = 20;
array[2] = 30;
array[3] = 40;
array[4] = 50;

pointer = &array[0];

cout << pointer[2];
```

O exemplo acima exibirá 30 como saída. Poderíamos fazer a atribuição `pointer = &array[0]`; da seguinte forma: `Pointer = array;` pois o nome do array funciona como um ponteiro para a sua primeira posição. Observe que, embora possamos utilizar o nome de um array como ponteiro, não podemos fazer coisas do tipo: `array = &array2[0];` Portanto, devemos considerar arrays como *ponteiros constantes.*

Usando new e delete com arrays

Podemos criar arrays em tempo de execução com o operador **new** em conjunto com **[]**:

```
int *pointer;
pointer = new int[10];
pointer[0] = 10;
```

Como C++ permite utilizarmos ponteiros com o operador colchetes, [], `pointer` se comportará como um array. Para desalocar a memória alocada com **new[]**, utilizamos **delete[]**:

```
delete[] pointer;
```

Aritmética de ponteiros

Como sabemos, os ponteiros são variáveis que guardam endereços de memória. Por serem valores inteiros, podemos utilizar operações aritméticas com ponteiros. A diferença é que, se executarmos a instrução: um_inteiro++; a variável um_inteiro será acrescida de 1. Se executarmos: pointer++; a variável pointer será acrescida de um número igual ao tamanho do tipo de pointer, ou seja, se pointer for um ponteiro para int, seu valor será acrescido de 4 (um inteiro tem geralmente 4 bytes), de forma que, se pointer estiver apontando para uma seqüência de inteiros, pointer++ fará com que ele aponte para o próximo elemento. Para ilustrar melhor a aritmética com ponteiros, segue um exemplo:

```
int *pointer, array[10];
pointer = array;

for (int i=0; i<10; i++)
{
    *pointer = i*2;
    pointer++;
}
```

Neste exemplo, em vez de acessarmos os elementos do array através do operador colchetes, [], utilizamos um ponteiro para percorrê-lo. Como vimos anteriormente, arrays e ponteiros são semelhantes, de forma que esta aritmética também vale para arrays: `*(array+n)` é equivalente a `array[n]`.

Ponteiros e const

Além de podermos declarar ponteiros para variáveis, C++ nos permite declarar ponteiros para valores constantes:

```
const int *pointer;
```

Isto não significa que o ponteiro só poderá apontar para constantes. Na realidade o que estamos fazendo é informando ao compilador que ele deve tratar o valor para o qual o ponteiro aponta como uma constante, e qualquer tentativa de tratá-lo de forma diferente causará erro de compilação. A importância de utilizar ponteiros desta forma, é nos prevenirmos de erros que ocorreriam se o valor fosse inadvertidamente alterado.

Outro motivo é que só ponteiros para constantes podem apontar para constantes, os ponteiros normais não (isto é um tanto quanto óbvio).
A palavra-chave **const** também pode ser utilizada para declarar ponteiros constantes (note a diferença de ponteiros *para* constantes). Como era de se esperar ponteiros constantes só poderão apontar para um endereço de memória constante, ou seja, não podem alterar o endereço contido nele. Exemplo:

```
int um_inteiro;
int *const pointer = &um_inteiro;
```

Portanto, como você já deve desconfiar, também podemos criar ponteiros constantes para constantes, o que significa que não poderemos alterar o endereço contido nele e que o valor contido neste endereço será tratado como constante:

```
int um_inteiro;
const int *const pointer = &um_inteiro;
```

Funções

Como vimos no início do capítulo, as funções são a base da engenharia de software estruturada. Então vamos a elas!

Definindo funções

Para definir uma função, utilizamos a seguinte estrutura:

```
tipo nome(argumentos...)
{
    instruções;
    return expressão_de_retorno;
}
```

Onde:
tipo: o tipo de retorno da função.
nome: é o nome da função.
argumentos: lista de zero ou mais parâmetros que devem ser passados à função quando a mesma é chamada.
instruções: conjunto de instruções que serão executadas quando a função for chamada.
expressão_de_retorno: expressão retornada para o ponto onde a função foi chamada. Deve ser do mesmo tipo de **tipo**.
Abaixo segue um exemplo de uma definição de função:

```
int add(int numero_1, int numero_2)
{
    int numero_3;
    numero_3 = numero_1 + numero_2;
    return numero_3;
}
```

Claro que as funções que nós criamos são geralmente um pouco mais complexas, mas este exemplo ilustra bem os elementos da definição de uma função.
Em C++, além de definirmos uma função (com todos os seus elementos), devemos também declarar as funções que definimos. Esta declaração é chamada de protótipo de uma função. O protótipo de uma função nada mais é do que a primeira linha da definição da função seguida de um ponto-e-vírgula. Ex:

```
int add(int numero_1, int numero_2);
```

O código acima demonstra o protótipo da função add, definida no exemplo anterior.
Para estruturar melhor o código, criamos um arquivo com extensão cpp com as definições das funções e um arquivo com extensão **.h** com os protótipos das funções.

Argumentos e tipos de retorno de funções

Podemos passar qualquer tipo de dados como argumento (ou parâmetro) para uma função. Abaixo veremos algumas peculiaridades dos tipos de dados mais complexos, e como retornar estes tipos.

Funções e arrays

Para passar um array para uma função, basta adicionar [] ao nome do argumento, ex:

```
int add(int array[], int num_elementos)
{
    int resultado=0;
    for (int i=0; i<num_elementos; i++)
    {
        resultado += array[i];
    }
     return resultado;
}
```

É bastante importante ressaltar o que está acontecendo aqui. Ao lembrarmos que C++ trata o nome de arrays como ponteiros constantes, podemos perceber que o que estamos passando como argumento não é uma cópia de qualquer elemento do array, mas sim um ponteiro para o array original. Como podemos utilizar ponteiros como arrays (com o operador []), a primeira vista, o código acima pode não demonstrar o realmente está acontecendo. Portanto, se alterássemos qualquer elemento do array, estaríamos alterando no array original. Para prevenirmos alterações inadvertidas, devemos declarar o parâmetro do array como constante, como veremos abaixo:

```
int add(const int array[], int num_elementos)
{
    int resultado=0;
    for (int i=0; i<num_elementos; i++)
    {
        resultado += array[i];
    }
     return resultado;
}
```

Desta forma, garantimos que a passagem de parâmetros por referência que está implícita não implique em um erro grave no nosso programa.

Retornando um array de uma função

O C++ não permite atribuir um array inteiro a outro, nem retornar um array inteiro de uma função. Portanto, para retornar um array de uma função (e eventualmente atribuir isto a outro), devemos usar ponteiros. Como existe uma semelhança entre ponteiros e arrays, este processo é bastante simples. Vejamos um exemplo:

```
int *init_array(int num_elementos)
```

```
{
    int *retorno = new int[num_elementos];

    for (int i=0; i<num_elementos; i++)
    {
        retorno[i] = 0;
    }
     return retorno;
}
```

Devemos observar os seguintes aspectos no trecho de código acima:

- O tipo de retorno da função é na realidade um ponteiro, como já foi explicado antes. Portanto, o resultado desta função só pode ser atribuído a ponteiros, e não a arrays.
- A variável de retorno não foi declarada como um array normal - como: `int array[num_elementos]` - pois, se fosse declarada desta forma, seria alocada na área de armazenamento automático, sendo desalocada ao final do seu escopo, ou seja, no fim da função. Então, quando tentássemos utilizar o array de retorno da função estaríamos trabalhando com uma área de memória desalocada, que poderia conter qualquer coisa, ou pior, ter sido alocada para outra variável.
- Note que alocamos memória dentro desta função e não desalocamos. Portanto, é tarefa do código que chamou a função desalocar esta memória com o operador **delete[]**, quando ela deixar de ser utilizada.

Funções e estruturas

Em C++, as estruturas são passadas para as funções como valor, de forma que passar uma estrutura para uma função é o mesmo que passar uma variável simples. Exemplo:

```
struct structure
{
    int um_inteiro;
    char um_char;
};

void function(structure uma_estrutura)
{
    cout << “Exibindo o número: ”<<uma_estrutura.um_inteiro<<endl;
    cout << “Exibindo o char: ” << uma_estrutura.um_char << endl;
}
```

Na passagem de parâmetros por valor, o que acontece é que a estrutura passada é copiada para uma variável local, no armazenamento automático. Portanto, para passar estruturas grandes podemos optar por passar um ponteiro para a estrutura, evitando assim o *overhead*[2] da cópia.
O mesmo se aplica ao retornar estruturas de uma função, a estrutura também é passada por valor. Exemplo:

```
structure get_structure()
{
    structure estrutura;
    estrutura.um_inteiro = 0;
    estrutura.um_char = ‘A’;
    return estrutura;
}
```

[2] Tempo gasto com execução de código que não foi criado pelo programador, e que portanto, não faz parte do programa diretamente.

Como a estrutura é passada por valor, não precisamos nos preocupar com seu escopo. Na realidade, a variável `estrutura` perde o escopo ao fim da função, mas como o que é retornado é uma cópia da mesma, não precisamos nos preocupar com isto.

Ponteiros para funções

Como vimos anteriormente, podemos utilizar ponteiros para qualquer tipo de dados. Nesta seção veremos que também podemos utilizar ponteiros para funções. Como as funções são dados compilados armazenados na memória, nada nos impede que tenhamos um ponteiro para estes dados. Um ponteiro para função aponta para posição de memória para a qual o programa é transferido quando a função é chamada. Utilizamos ponteiros para funções para passar estas funções como argumento para outras funções. Se o protótipo de uma função é:

```
int função(int i);
```

Um ponteiro para esta função poderia ser declarado da seguinte forma:

```
int (*ponteiro_para_função)(int i);
```

Da mesma forma que o nome de um array é um ponteiro constante para o array, o nome de uma função é um ponteiro constante para a função, de forma que podemos inicializar o nosso ponteiro da seguinte forma:

```
ponteiro_para_função = funçao;
```

Para passarmos uma função como parâmetro para outra, passamos a função como ponteiro:

```
void função2((*ponteiro_para_função)(int i))
{
   int parâmetro = 13723;
   (*ponteiro_para_funçao)(parametro);
}
```

Como podemos observar, `função2` recebe uma função como parâmetro e chama esta função com o argumento 13723.

Funções in-line

Quando fazemos uma chama a uma função, o fluxo do programa é desviado para o endereço da função, seus argumentos são colocados na pilha, entre outras coisas. Isto gera um *overhead* na execução de um programa. Para situações críticas, ou funções bastante pequenas (com uma ou duas instruções), C++ nos permite criar funções in-line. No lugar das chamadas a estas funções, o compilador coloca todo o código da função, de forma que nenhum desvio é necessário. A definição de uma função in-line é feita com a palavra-chave **inline**:

```
inline int dobro(int num)
{
    return 2*num;
}
```

Desta forma, toda vez que fizermos uma chamada à função `dobro`, o código: `2*num` será colocado em seu lugar.

Um pouco mais sobre argumentos

C+ também nos permite utilizar os argumentos de função de forma bastante eficiente. Podemos definir argumentos padrão ou um número variável de argumentos para uma função, como veremos a seguir.

Argumentos padrão

Definindo argumentos padrão para uma função, estamos especificando um valor que será utilizado caso não seja fornecido nenhum valor para o argumento. Para definir um argumento padrão para uma função, utilizamos o igual, **=**, seguido pelo **valor_padrão**, na definição do protótipo da função:

```
int sum(int num1, int num2 = 1);

int sum(int num1, int num2)
{
    return num1+num2;
}

result = sum(10);
```

No código acima, como `num2` foi declarado com valor padrão igual a 1, sempre que fizermos uma chamada a `sum` sem o segundo argumento, o valor 1 será utilizado no seu lugar. Portanto, quando definimos um valor padrão para um argumento, devemos omitir este argumento para utilizar seu valor padrão. Isto quer dizer que os argumentos padrão devem ser os últimos argumentos da função, pois, caso isto não ocorra, o compilador será incapaz de identificar qual dos argumentos está faltando.

Argumentos variáveis

Além de definir argumentos padrão, C++ disponibiliza um conjunto de macros para que possamos definir uma função com um número de argumentos variável. Estas macros são:

`va_list:` obtém a lista de argumentos passados para a função
`va_start(va_list list, int num_elementos):` inicializa a lista
`va_arg(va_list list, tipo):` retorna um argumento da lista
`va_end(va_list list):` finaliza o uso da lista de argumentos

Para utilizarmos as macros, devemos incluir o arquivo de cabeçalho **`cstdarg`**. Abaixo segue um exemplo que mostra como criar uma função com um número de argumentos variável:

```
int sum(int...);

int main ()
{
    cout “a soma de 1, 2 e 3 é: ” << sum(3, 1, 2, 3) << endl;
    return 0;
}

int sum(int num_elementos ...)
{
    int resultado = 0, numero;
    va_list lista;
    va_start(lista, num_elementos);

    for(int i = 0; i < num_elementos; i++)
    {
        numero = va_arg(lista, int);
```

```
            resultado += numero;
        }

        va_end(lista);

        return resultado;
    }
```

Neste exemplo, a função sum recebe um argumento obrigatório, que é a quantidade de argumentos variáveis que estamos passando, seguido de vários argumentos. Então, utiliza as macros para percorrer a lista de argumentos e somá-los.

Sobrecarga de funções

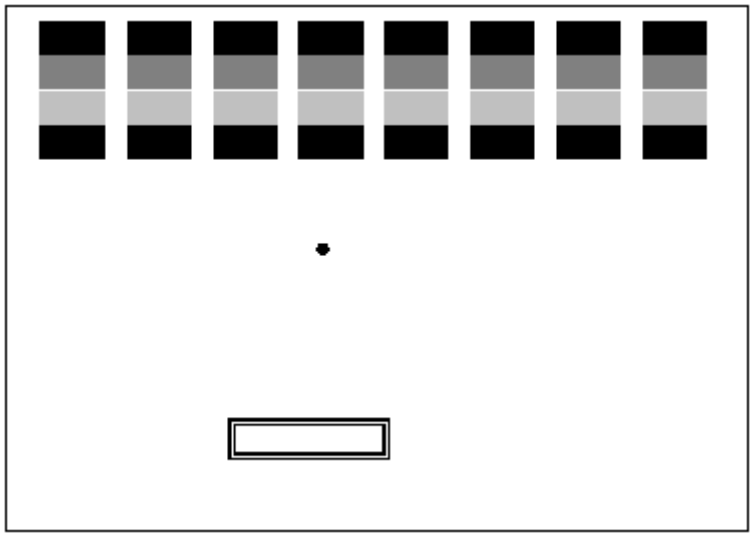
Às vezes, desejamos criar uma função que possa ser chamada com diferentes tipos de argumento. Isto nos permite personalizar o código e torná-lo muito mais fácil de trabalhar. C++ permite que sobrecarreguemos funções para que estas aceitem argumentos de tipos e números diferentes. Para isto, basta apenas definir as funções tantas vezes quantas forem necessárias. Note que a assinatura, lista de argumentos da função,deve ser diferente, ou seja: o tipo ou a quantidade (não apenas o nome) dos argumentos deve ser diferentes, nas diferentes versões de uma função sobrecarregada. Abaixo segue um exemplo de sobrecarga de função:

```
int sum(int &num1, int num2);
char *sum(char *str1, const char *str2);

int main()
{
    int num1= 10, num2 = 15, num3 = 20;
    char const *str1 = "World!";
    char str2[20] = "great ";
    char str3[20] = "Hello ";

    cout << "--- Funcao sum com inteiros ---" << endl;
    cout << "soma total: " << sum(num3, sum(num2, num1)) << endl;
    cout << "num1: " << num1 << endl;
    cout << "num2: " << num2 << endl;
    cout << "num3: " << num3 << endl << endl;
    cout << "--- Funcao sum com strings ---" << endl;
    cout << "Strings: " << sum(str3, sum(str2, str1)) << endl;
    cout << "str1: " << str1 << endl;
    cout << "str2: " << str2 << endl;
    cout << "str3: " << str3 << endl;

    cin.get();
    return 0;
}

int sum(int &num1, int num2)
{
    return (num1 = num1 + num2);
}

char *sum(char *str1, const char *str2)
{
    return strcat(str1, str2);
}
```

No código acima, a função sum pode ser utilizada da mesma forma com inteiros e strings. Foi utilizado `const char *str2`, como segundo argumento, para garantir que este argumento não seria modificado no corpo da função.

Seção prática: definindo nosso projeto

Para fixar nossos conhecimentos, teremos como projeto prático um jogo em C++. Este jogo irá evoluir a cada capítulo, sempre utilizando os novos conceitos estudados.
Portanto, desenvolveremos um jogo do tipo ***Arcanoid***, no qual o jogador controla uma barra, localizada em baixo da tela, e tem como objetivo destruir um conjunto de blocos rebatendo uma bolinha. Este jogo será desenvolvido em modo texto, pois estamos focando o ANSI/ISO C++, que é compatível com qualquer sistema operacional, e a utilização de um ambiente gráfico depende do sistema operacional utilizado. Abaixo segue um modelo da tela do jogo:



Como prática deste capítulo, iremos projetar o nosso jogo utilizando funções. Iremos criar funções para:

- Apagar a tela
- Desenhar e apagar um bloco em uma posição da tela
- Desenhar e apagar a barra em uma posição da tela
- Desenhar e apagar a bola em uma posição da tela
- Mover a bola e testar a colisão com os outros objetos
- Mover a barra de acordo com entrada do teclado

Abaixo seguem as seqüências de caracteres de escape que modificam o estado da tela, se enviadas para cout (ESC significa o caractere de escape, cujo código ascii é 27):

Função	Seqüência de caracteres
Normal	ESC[0m
Bold	ESC[1m
Blink (piscando)	ESC[5m
Reverso (cores do texto e fundo trocadas)	ESC[7m
Mover para a posição 0,0	ESC[f
Mover para a posição **x,y**	ESC[f ESC[**y**B ESC[**x**C
Apagar a tela	ESC[f ESC[2J
Mudar a cor para **atr,fg,bg** (**atr**=0(normal) ou **atr**=1 (highlight), **fg**=texto, **bg**=fundo)	ESC[**atr;fg;bg**m

Note que **atr** e **fg** definem a cor do texto enquanto **bg** define a cor de fundo. Os valores que **atr**, **fg** e **bg** podem assumir são:

Cor	Valor de atr	Valor de fg	Valor de bg
Preto	0	30	40
Vermelho	0	31	41
Verde	0	32	42
Marrom	0	33	43
Azul	0	34	44
Magenta	0	35	45
Ciano	0	36	46
Cinza-claro	0	37	47
Cinza-escuro	1	30	-
Vermelho-claro	1	31	-
Verde-claro	1	32	-
Amarelo	1	33	-
Azul-claro	1	34	-
Magenta-claro	1	35	-
Ciano-claro	1	36	-
Branco	1	37	-

Agora já temos como manipular a saída de caracteres na tela. Você deve utilizar o arquivo *screen.h* que contém as definições mencionadas acima e criar a primeira versão do nosso jogo, que implemente as funções necessárias vistas anteriormente.

Capítulo II

Engenharia de Software Baseada em Objetos

Engenharia de software baseada em objetos

Modularidade, coesão e acoplamento

Com o crescimento da complexidade dos programas, tornou-se de grande valia a decomposição dos mesmos em unidades menores. A decomposição de um programa em módulos permite diminuir o tempo de programação ao se programar os módulos em paralelo, divide o problema a ser resolvido em partes menores e mais fáceis de serem implementadas e permite compilar e testar estas partes separadamente. Mas a modularização não traz só vantagens. Uma modularização mal feita pode acarretar em problemas com reusabilidade, extensibilidade, compatibilidade, etc. Os fatores que influenciam na modularização são: a coesão, interdependência das funções e estruturas internas de um módulo; e o acoplamento, nível de dependência entre os módulos. Portanto, para modularizar um programa corretamente, devemos:

- Maximizar a coesão: as funções e estruturas de dados que implementam uma funcionalidade (ou conjunto de funcionalidades relacionadas) do programa devem estar no mesmo módulo.
- Minimizar o acoplamento: os módulos devem funcionar os mais independentes possíveis dos outros módulos, para que ao modificarmos um módulo não seja necessário modificar os demais.

A modularização também deve seguir dois princípios: *information hidding*[3] e reusabilidade, que veremos a seguir.

Information Hidding

Para garantir um fraco acoplamento entre os módulos, devemos fazer com que um módulo "conheça" somente o necessário sobre o funcionamento interno dos módulos que ele utiliza. Portanto, devemos definir uma *interface para comunicação* entre os módulos, de forma que não seja visível a sua estrutura interna, apenas a sua interface.
Então, depois de definidos os conceitos básicos da engenharia de software baseada em objetos, podemos formular uma definição de módulo mais concreta e que será usada de agora em diante:

> "Um módulo possui uma série de *operações* (funções) e um *estado* (variáveis), que guarda o efeito das operações. Possui também uma *interface*: conjunto de dados e operações que estão disponíveis aos outros módulos; e uma *implementação*: definição das estruturas de dados e do corpo das funções. Esta definição de módulo é o que chamamos de estrutura de dados abstrata."

Suporte à modularidade em C++

C++ permite a divisão de nossos programas em diversos arquivos, de forma a podermos ter arquivos para cada módulo de nosso programa. As interfaces dos módulos são definidas em arquivos com extensão **.h** (arquivos de cabeçalho de C++) enquanto a implementação está nos arquivos com extensão **.cpp**. Quando um módulo necessitar utilizar outro, deve incluir o arquivo de cabeçalho correspondente e a interface do módulo a ser utilizado ficará disponível.

Definindo interfaces

[3] Information hidding é um conceito de engenharia de software que consiste em encapsular os dados de um módulo de forma que estes só sejam acessíveis através da interface do módulo.

As interfaces dos módulos são criadas em arquivos **.h**. C++ permite a criação de espaços de nomes separados para cada módulo, para que não se corra o risco de existir conflito entre nomes de dados ou funções iguais em módulos diferentes. Isto é feito utilizando-se a palavra-chave **namespace**.
Como vimos anteriormente, podemos utilizar as palavras-chave **extern** e **static** para definir a ligação (externa ou interna) dos dados. Veremos estes conceitos na prática na próxima seção.

Organização dos arquivos

Abaixo segue um modelo básico dos arquivos de um módulo chamado modulo1, com explicações sobre suas partes:

```
//-----------------------------------------------------------------
/* Arquivo modulo1.h */

#ifndef MODULO1_H    // verifica se o arquivo já foi chamado
#define MODULO1_H    // anteriormente para que não haja redefinição
                     // do módulo

namespace modulo1  // indica que as definições abaixo pertencem ao
{                  // espaço de nomes do modulo1

    // definição das funções e estruturas de dados da interface
    // do módulo

    extern void printResults();
    extern int funcao1(int num1, int num2);
    extern char *str1;

}

#endif /* fim de modulo1.h */

//-----------------------------------------------------------------
/* Arquivo modulo1.cpp */

#include "modulo1.h"   // inclui a interface do modulo 1

namespace modulo1  // indica que as definições abaixo pertencem
{                  // ao modulo1

    // definicoes privadas
    static int funcao1(int num1, int num2);
    static char *str1 = "Modulo 1 - variavel privada";

    // variavel publica
    char *str2 = "Modulo 1 - variavel publica";

    // funcoes publicas
    int funcao2(int num1, int num2)
    {
        return num1*num2;
    }

    void printResults()
    {
        cout <<"Usando funcao1 do modulo 1: "<<funcao1(1, 2)<<endl;
```

```
        cout <<"Imprimindo str1 do modulo 1: "<< str1 << endl;
    }

    // funcao privada
    static int funcao1(int num1, int num2)
    {
        return num1+num2;
    }
} /* fim do arquivo modulo1.cpp */
//--------------------------------------------------------------------
```

Tipos abstratos de dados

Neste capítulo, aprendemos conceitos importantes da Engenharia de Software baseada em objetos. A modularização de um programa traz diversas vantagens, pois permite a divisão de tarefas, torna mais fácil a implementação e permite testes mais específicos. Mas o que fazer se necessitarmos de mais de uma instância de um modulo? Uma solução seria copiar e colar os módulos em arquivos diferentes, tantas vezes quantas forem necessárias, mas isto vai de encontro aos princípios da extensibilidade e reusabilidade. No capítulo anterior, vimos como criar tipos de dados compostos, estruturas e uniões. Como uma estrutura é um novo tipo de dado, podemos ter várias instâncias de uma estrutura. A combinação da divisão do programa em módulos com a representação dos dados do módulo através de estruturas é o que chamamos de *Tipos de Dados Abstratos* (ADT – *Abstract Data Type*).
Além das regras a serem seguidas para construção de módulos e estruturas, a construção de um ADT deve levar em conta as seguintes funcionalidades:

- Construção: deve ser possível alocar e inicializar novas instâncias do ADT.
- Atribuição: devemos poder copiar os dados de uma instância para outra.
- Comparação: deve ser possível comparar duas instâncias de um ADT (atenção às diferenças entre identidade e igualdade)
- Manipulação dos dados: devemos poder manipular os dados de um ADT sem se preocupar com a forma de armazenamento interno, de forma a encapsular o ADT.
- Destruição: devemos ter a possibilidade de liberar a memória alocada por um ADT, quando não formos mais utilizá-lo.

Ao definirmos um ADT tendo em mente estas responsabilidades estaremos implicitamente aplicando todos os conceitos de engenharia de software que aprendemos até agora.

Suporte à reusabilidade: Modelos de função

Como vimos, C++ é uma linguagem fortemente tipada, o que nos impede de, ao definirmos uma função que manipula inteiros, utiliza-la com números de ponto flutuante, por exemplo. Se desejarmos uma função que tenha o mesmo comportamento (as mesmas instruções) para ambos os casos, podemos sobrecarregar esta função, mas isto feriria o conceito de reusabilidade de código, pois teríamos códigos idênticos replicados. Mas o C++ oferece uma alternativa para isto: podemos criar modelos de função, que independam do tipo de dados, e utilizar este modelo para ambos os tipos. Para criar modelos de função (function templates), utilizaremos as palavras **template** e **typename**:

```
template <typename T>
void add(T &num);

int main()
{
    int num1 = 10;
```

```
    float num2 = 12.34;
    double num3 = 56.7890;

    add(num1);
    add(num2);
    add(num3);

    cout << “num1: ” << num1 << endl;
    cout << “num2: ” << num2 << endl;
    cout << “num3: ” << num3 << endl;

    cin.get();

    return 0;
}

template <typename T>
void add(T &num)
{
    num++;
}
```

Como podemos perceber, o compilador se encarregou de “gerar” a função para cada tipo dedado. Mas o que ocorreria se utilizássemos um tipo para o qual a função não é adequada? A função add funciona muito bem para tipos numéricos, mas não seria adequada para strings por exemplo. Podemos, então definir uma função add para o tipo string que exiba uma mensagem de erro, informando que add não é adequada para strings. Isto é feito criando especializações da função template. Para criar uma especialização para um determinado tipo, utilizamos template<>, e declaramos a função com o tipo desejado:

```
template <>
void add (char *str)
{
    cout << “Não é possível adicionar um à string ” << str << endl;
}
```

Seção prática: estruturando o nosso projeto em módulos

Continuando com o projeto do nosso jogo, iremos dividir o nosso projeto em módulos buscando torná-los o mais independente possível uns dos outros. Os módulos podem ser divididos da seguinte forma:

screen.cpp – responsável pelas rotinas de desenho e manipulação de imagens na tela.
game.cpp – responsável pelo controle principal do jogo, pontuação e entrada do teclado.
sprites.cpp – responsável pelo controle e movimentação dos elementos do jogo (blocos, bola e barra).
noid.cpp – Possui a função main e possíveis funções auxiliares.

Devemos definir as interfaces de cada módulo antes de começar a implementá-los. Faça diagramas representando as relações entre os módulos para ajudá-lo. Os módulos devem ser implementados seguindo os princípios de um ADT.

Capítulo III

Engenharia de Software Baseada em Classes

Engenharia de Software Baseada em Classes

No capítulo anterior, vimos como criar tipos abstratos de dados. O processo de dividir um problema em partes facilmente manipuláveis é chamado de abstração. O C++ fornece um suporte à construção de tipos abstratos, com a utilização de estruturas, e funções que as manipulem. Embora seja funcional, esta abordagem não está completamente de acordo com os conceitos de encapsulamento e modularidade. Para resolver estes problemas foi introduzido em C++ o conceito de classe. Uma classe é um molde a partir do qual pode-se criar tipos abstratos, ou, como chamamos, objetos. Neste capítulo veremos como definir classes e várias peculiaridades destas entidades em C++.

Classes em C++

Vamos iniciar o nosso aprendizado partindo de uma estrutura. Da mesma forma que adicionamos variáveis, podemos adicionar funções a uma estrutura:

```
struct TBall
{
    int x, y;
    TColor color;
    int radius;

    void move(int x_step, int y_step)
    {
        x += x_step;
        y += y_step;
    }
};
```

Portanto, a estrutura TBall agora possui uma função-membro. Isto significa que a função pertence à estrutura e só pode ser chamada a partir dela. Também podemos observar que os membros da estrutura são visíveis dentro da função-membro. Para transformar esta estrutura em uma classe, as únicas coisas que devemos fazer são: substituir **`struct`** por **`class`** e adicionar a palavra-chave **`public:`** para indicar que podemos acessar os membros da classe:

```
class TBall
{
  public:
    int x, y;
    TColor color;
    int radius;

    void move(int x_step, int y_step)
    {
        x += x_step;
        y += y_step;
    }
};
```

Com estes exemplos podemos perceber uma das diferenças entre classes e estruturas: nas estruturas, todos os membros são públicos, enquanto nas classes podemos determinar a visibilidade dos membros através dos seguintes modificadores:

- **public:** torna os membros da classe visíveis fora da classe
- **protected:** os membros da classe só são visíveis dentro da própria classe e nas classes derivadas (veremos como criar classes derivadas, ou subclasses, no próximo capítulo).
- **private:** os membros da classe só são visíveis dentro da própria classe.

Portanto, uma vez visto como definir uma classe, eis um exemplo da classe `TBall` com todos os requisitos de um ADT:

```
class TBall
{
  private: // os atributos da classe são todos privados
           // lembre-se do conceito de information hidding
    int x, y;
    TColor color;
    int radius;

  public: // Todos os métodos de manipulação de atributos, os
          // construtores e o destrutor devem ser públicos.

    // Construtores da classe TBall, servem para alocar memória
    // e inicializar os objetos da classe.
    // Os construtores são métodos que devem possuir o mesmo
    // nome da classe e são chamados sempre que um novo
    // objeto da classe é criado. Abaixo temos dois construtores
    // para a classe:

    // Construtor 1: não possui parâmetros, inicializando os
    // atributos com valores padrão.
    TBall ()
    {
        x = 0;
        y = 0;
        color = clBlack;
        radius = 1;
    }

    // Construtor 2: permite que os atributos sejam inicializados
    // com os valores desejados.
    TBall (int x1, int y1, TColor c, int r)
    {
        x = x1;
        y = y1;
        color = c;
        radius = r;
    }

    // Destrutor da classe TBall. Deve possuir o mesmo nome da
    // classe precedido de um ~. É chamado toda vez que um
    // objeto é destruído (com o operador delete).
    // No seu corpo geralmente existem instruções para desalocar as
    // variáveis criadas dinamicamente. Como não existem atributos
    // desta forma, o corpo do destrutor está vazio e sua
    // declaração poderia ser omitida. Foi exibida aqui a título de
    // exemplo de destrutor.
    ~TBall{}

    // Além dos construtores e do destrutor, devemos fornecer
```

```
        // métodos públicos para manipular os atributos da classe.
        void move(int x_step, int y_step)
        {
            x += x_step;
            y += y_step;
        }

        int getX()
        {
            return x;
        }

        int getY()
        {
            return y;
        }

        TColor getColor()
        {
            return color;
        }

        int getRadius()
        {
            return radius;
        }

        void setColor(TColor c)
        {
            color = c;
        }

        void setRadius(int r)
        {
            radius = r;
        }

    };
```

Portanto, aí está a nossa primeira classe, seguindo todos os requisitos da engenharia de software. É importante salientar que uma classe pode conter tantos construtores quantos forem necessários, mas deve possuir um único destrutor. Como o destrutor é chamado pela estrutura interna do programa, quando um objeto sai do escopo ou quando utilizamos o operador **delete**, um destrutor não possui parâmetros.

Classes: organização em arquivos

Apesar de poder criar classes da forma como criamos anteriormente, C++ nos dá a possibilidade de separarmos a declaração da classe (arquivos .h) de sua implementação (arquivos .cpp). Desta forma, podemos utilizar a classe em diversos outros arquivos, bastando para isto incluir o arquivo de cabeçalho que contém a definição da classe. Na realidade, esta é a forma correta de implementação, pois ao definirmos uma classe com o corpo dos métodos na declaração, o compilador trata isto como se fosse uma declaração implícita de métodos *in-line*, o que não é desejado para todos os métodos. Abaixo segue um exemplo resumido de como dividir uma classe em arquivos de cabeçalho e implementação:

```
    /* Arquivo TBall.h */
```

```
#ifndef TBALL_H
#define TBALL_H
class TBall
{
  private:
    int x, y;
    TColor color;
    int radius;

  public:
    void move(int x_step, int y_step); // declaração dos métodos
    .
    .
    .
};

#endif

/* fim de TBall.h */

/*Arquivo TBall.cpp*/

void TBall::move(int x_step, int y_step) // corpo do método
{
    x += x_step;
    y += y_step;
}
.
.
.
/* fim de TBall.cpp*/
```

Como uma classe define um escopo, a implementação dos métodos de uma classe deve vir precedida do nome da classe seguido pelo operador de escopo `::`.

Modificadores de métodos

Além da forma normal de declarar métodos, C++ oferece um conjunto de modificadores de métodos (alguns já vistos com funções) que alteram a forma como estes são utilizados.

Métodos in-line

Como vimos anteriormente, os métodos cuja implementação está dentro da definição da classe são, por padrão, in-line. Uma outra forma de criar métodos in-line é utilizando a palavra clave **inline**, como veremos a seguir:

```
inline void TBall::setColor(TColor c)
{
    color = c;
}
```

A declaração do método na definição da classe continua sendo feita da mesma forma.

Métodos constantes

Se declararmos um objeto de uma classe como constante, com o uso de **const**, só poderemos chamar os métodos do objeto que forem constantes também, ou seja, um método ser chamado constante significa que o método não alterará os atributos do objeto. Os métodos constantes também são declarados com a palavra-chave **const** colocada após a lista de argumentos do método:

```
TColor TBall::getColor() const
{
    return color;
}
```

Como criar membros estáticos

Como vimos anteriormente, podemos declarar variáveis externas como estáticas para tornar sua ligação interna e declarar variáveis automáticas como estáticas para mudar a sua classe de armazenamento para estática. Em uma classe podemos utilizar o modificador **static** de duas formas:

- Usando static com um atributo, tornamos este atributo compartilhado com todos os objetos da classe, ou seja, todos os objetos da classe compartilham a mesma posição de memória para este atributo.
- Usando static com um método, este se torna método da classe, e não do objeto, de forma que não precisamos instanciar um objeto da classe para chamar o método. Como os métodos estáticos não fazem parte de nenhum objeto de memória, eles só podem acessar os atributos estáticos da classe. Para chamar um método estático, procedemos da seguinte forma:

```
class TBall
{
  public:
    static string getInfo();
    .
    .
    .
};

static string TBall::getInfo()
{
    return “This is a ball”;
}

int main ()
{
    cout << TBall::getInfo() << endl;
    return 0;
}
```

O ponteiro this

Toda classe em C++ possui um ponteiro especial chamado **this**, que aponta para a instancia atual do objeto. Por exemplo, se criarmos um método que necessite retornar o próprio objeto, podemos utilizar **this** para isto. Ex:

```
class TBall
{
    TBall & compare(const TBall &ball);
    .
    .
```

```
    .
};

TBall & TBall::compare(const TBall &ball)
{
    if (radius >= ball.radius)
        return *this;
    else
        return ball;
}

int main ()
{
    TBall ball1(10, 20, clWhite, 30);
    TBall ball2(30, 100, clBlack, 10);

    cout << "A bola maior é a " <<
         ball1.compare(ball2).getColor.getName() << endl;

    return 0;
}
```

A saída do programa será:

```
A bola maior é a branca
```

Podemos perceber, neste exemplo, que passamos e retornamos o objeto `TBall` por referência. Por que isto é feito? Você deve lembrar que o C++, ao passar um estrutura para uma função, passa-a por valor, fazendo uma cópia da estrutura na área de armazenamento automático. Com objetos acontece o mesmo. Para evitar este *overhead* da cópia, os parâmetros são passados por referência, e, quando não serão alterados dentro do método, são passados como referência constante.

Sobrecarga de métodos e construtores

Do mesmo modo que sobrecarregamos funções, C++ nos permite sobrecarregar métodos e construtores. Para isto, redefinimos o método ou construtor com uma assinatura (lista de argumentos) diferente. Então o compilador se encarrega de verificar qual das formas sobrecarregadas do método ou construtor será utilizada, com base na sua lista de argumentos.

Construtores de cópia

Os objetos, como as estruturas, são passados de e para funções/métodos por valor. Isto que dizer que, ao retornarmos um objeto, por exemplo, uma cópia do objeto é passada em seu lugar. Isto também ocorre quando fazemos uma atribuição, com o operador igual, = (veremos como corrigir este problema com a sobrecarga do operador igual, =, mais a frente, neste capítulo). Então o que acontece se o objeto tiver membros que são ponteiros? Exatamente! Apesar do ponteiro ser copiado, ele continua apontando para a mesma posição de memória que o ponteiro do objeto antigo. Portanto, o objeto antigo e o novo terão uma mesma área de memória compartilhada.
Para resolver este problema, C++ oferece os construtores de cópia. Na realidade, toda classe possui um construtor de cópia padrão, que simplesmente copia os valores dos membros de um objeto para o outro. Quando definimos uma classe que tem membros que são ponteiros, devemos criar o nosso próprio construtor de cópia. Para isto, basta criarmos um construtor que receba uma referência constante para um objeto da mesma classe. Neste construtor, podemos reinicializar quaisquer ponteiros ou outros dados, conforme a necessidade, para criarmos a nova cópia do objeto. Vejamos um exemplo:

```
class TPlayer
{
  private:
    char *name;
    int score;
  public:
    TPlayer(char *str, int scr);
    Tplayer(const TPlayer & player);
    .
    .
    .
};

TPlayer::TPlayer(char *str, int scr)
{
    name = new char[strlen(str)+1];
    strcpy(name, str);
    score = scr;
}

TPlayer::TPlayer(const TPlayer & player)
{
    name = new char[strlen(player.name)+1];
    strcpy(name, player.name);
    score = player.score;
}
```

Usar construtores de cópia deve se tornar um hábito daqui para frente, principalmente se tivermos objetos que armazenem ponteiros.

Funções, métodos e classes *friend*

Criando funções *friend*

Funções *friend* (“amiga”) é uma classe especial de funções que tem acesso a todos os membros (públicos, protegidos e privados) de uma classe. Isto é bastante útil quando temos várias classes e não queremos criar um mesmo método utilitário para cada classe. Em vez disso, declaramos uma função como *friend* de todas as classes. Exemplo:

```
class TPlayer
{
  private:
    char *name;
    int score;
  public:
    friend void display(const Tplayer & player);
    .
    .
    .
};

void display(const TPlayer & player)
{
```

```
    cout << “Nome: ” << player.name << endl;
    cout << “Score: ” << player.score << endl;
}
```

Devemos utilizar as funções *friend* com bastante cuidado, pois seu uso indevido pode causar um furo no conceito de *informtion hidding*. Nas próximas seções veremos mais alguns motivos para criarmos funções *friend*.
Além de poder criar funções *friend*, podemos criar métodos e classes *friend*. Veremos isto nas seções a seguir.

Criando métodos *friend*

A criação de um método *friend* é análoga à de uma função *friend*, a única diferença é que estamos utilizando um método e não uma função (claro!). Vejamos um exemplo:

```
class TPlayer;

class TScreen
{
  public:
    void display(const TPlayer & player);
};

class TPlayer
{
  private:
    char *name;
    int score;
  public:
    friend void TScreen::display(const TPlayer & player);
    .
    .
    .
};

void TScreen::display(const TPlayer & player)
{
    cout << “Nome: ” << player.name << endl;
    cout << “Score: ” << player.score << endl;
}
```

Note que tivemos que declarar a classe `TPlayer` (veja a primeira linha do exemplo) antes de utilizá-la no método `display` de `TScreen`. Não podíamos simplesmente ter colocado a classe `TPlayer` antes pois ela declara o método `display` de `TScreen` como *friend,* e o compilador “reclamaria” que ainda não “viu” a classe `TScreen`.

Criando classes *friend*

Ao declarar uma classe como *friend* de outra classe, a primeira tem acesso a todos os membros da segunda (como já era de se esperar). Portanto, todos os seus métodos tornam-se *friend* da segunda classe. Exemplo:

```
class TPlayer
{
  private:
    char *name;
    int score;
```

```
    friend class TScreen;
    .
    .
    .
};

class TScreen
{
  public:
    void display(const TPlayer & player);
};

void TScreen::display(const TPlayer & player)
{
    cout << “Nome: ” << player.name << endl;
    cout << “Score: ” << player.score << endl;
}
```

Conversões entre objetos

Além de conversões entre tipos básicos, C++ nos permite definir conversões entre os nossos objetos, tornando-os o mais parecidos possível com os tipos básicos da linguagem.

Conversões para objetos

Para converter um tipo de dados para um objeto, devemos criar um construtor de conversão. Este construtor recebe como argumento o tipo de dados e cria um novo objeto, atribuindo este tipo ao membro correspondente do objeto criado. Por exemplo, se quiséssemos atribuir um inteiro a um objeto do tipo `TBall` e este inteiro representasse o raio da bola, deveríamos criar um construtor da seguinte forma:

```
TBall::TBall(int r)
{
    x = 0;
    y = 0;
    radius = r;
    color = clBlack;
}
```

Então, o código abaixo funcionaria corretamente:

```
TBall ball1;
ball1 = 10;
```

Conversão a partir de objetos

Para converter um objeto para um tipo de dado, utilizamos métodos especiais. Por exemplo, para converter um `TBall` para um **int** (que representaria o raio da bola), declaramos um método de nome:

```
operator int();
```

Eis um exemplo:

```
class TBall
{
```

```
  private:
    int x, y;
    TColor color;
    int radius;

  public:
    operator int() const;
    .
    .
    .
};

TBall::operator int() const
{
    return radius;
}
```

Assim, o código abaixo também funcionará corretamente:

```
TBall ball1(10, 20, clBlack, 30);
int raio;
raio = ball1;
```

Sobrecarga de operadores

Um dos recursos mais interessantes e potentes de C++ (e uma das diferenças entre C++ e outras linguagens POO como Java, por exemplo) é a possibilidade de sobrecarregar operadores para manipular os objetos das classes que criamos. Ao fazer sobrecarga de operadores, devemos seguir as seguintes restrições:

- não podemos sobrecarregar os operadores: **sizeof**, **.**, **::**, **.***, **?:**, **type_id**, **const_cast**, **dynamic_cast**, **reinterpret_cast e static_cast**;
- os operadores sobrecarregados devem ser usados com pelo menos um operando do tipo sobrecarregado;
- não é possível criar operadores completamente novos nem mudar um operador de binário para unário e vice-versa.

Apesar disto, podemos sobrecarregar todos os demais operadores, inclusive **new**, **delete**, **[]**, **+=**, **<<** e **>>**.
A sobrecarga de operadores é semelhante à sobrecarga de métodos, a diferença é que utilizamos a palavra-chave **operator** para indicar que estamos sobrecarregando um operador. Nas seções seguintes, veremos algumas peculiaridades da sobrecarga de operadores que necessitam de uma atenção a mais.

Sobrecarregando operadores unários, como ++ e --

Para sobrecarregar um operador unário, devemos criar um método na classe com a seguinte sintaxe:

```
tipo operatorop()
{

}
```

Onde tipo, é o tipo de retorno do operador, que neste caso é o nome da classe e *op* é o operador em si, como + + ou --.
Portanto, para sobrecarregar o operador ++ da classe `TBall` que incrementa o raio da bola, procedemos da seguinte forma:

```
class TBall
{
  private:
    int x, y;
    TColor color;
    int radius;

  public:
    TBall & operator++();
    .
    .
    .
};

TBall & TBall::operator++()
{
    radius++;
    return (*this);
}
```

Devemos observar que este código só sobrecarrega o operador ++ pré-fixado. Para sobrecarregar o operador pós-fixado, utilizamos:

```
TBall operator++(int);
```

Onde o argumento **int** diferencia os operadores.

Sobrecarregando operadores binários, como + e -

Para sobrecarregar operadores binários, devemos passar como argumento do operador um objeto do tipo do segundo operando (isto permite que utilizemos este operador com objetos de tipos diferentes, por exemplo). A sintaxe é:

```
class TBall
{
  private:
    int x, y;
    TColor color;
    int radius;

  public:
    TBall(const TBall &ball);
    TBall operator+(const TBall &ball);
    .
    .
    .
};

TBall TBall::operator+(const TBall &ball)
{
    TBall retBall(0, 0, clBlack, 0);
    retBall.x = ball1.x + ball2.x;
    retBall.y = ball1.y + ball2.y;
```

```
    retBall.radius = radius + ball.radius;
    return retBall;
}
```

Note que criamos um novo objeto dentro do corpo do operador e retornamos este objeto (na realidade uma cópia deste objeto).
Mas o que aconteceria se desejássemos fazer coisas do tipo: `ball = ball + 2` ou `ball = 2 + ball`? Como na segunda expressão, o operando não é um objeto, devemos definir uma função *friend* que receba ambos os operandos. Vejamos um exemplo:

```
class TBall
{
  private:
    int x, y;
    TColor color;
    int radius;

  public:
    TBall(int x1, int y1, TColor c, int r);
    TBall(int r);
    friend TBall operator+(const TBall &ball1, const TBall &ball2);
    .
    .
    .
};

TBall::TBall(int r)
{
    x = 0;
    y = 0;
    radius = r;
    color = clBlack;
}

TBall operator+(const TBall &ball1, const TBall &ball2)
{
    TBall retBall(0, 0, clBlack, 0);
    retBall.x = ball1.x + ball2.x;
    retBall.y = ball1.y + ball2.y;
    retBall.radius = ball1.radius + ball2.radius;
    return retBall;
}
```

Devemos notar que criamos um construtor de conversão de **int** para `TBall`, de modo esta solução atende a todas as expressões vistas anteriormente. Este é um dos principais motivos do uso de funções *friend*.

Sobrecarregando os operadores >> e <<

C++ permite sobrecarregar os operadores de inserção e extração de fluxo: >> e <<. Para sobrecarregar estes operadores, devemos passar como parâmetro e retornar o fluxo correspondente (**ostream** ou **istream**), para que estes operadores possam ser encadeados. É padrão tornar estes operadores *friend* da classe para o qual está sendo sobrecarregado. Vejamos a sintaxe da sobrecarga deste operador:

```
friend ostream & operator<<(ostream &stream, const tipo_classe &c);
friend istream & operator>>(istream &stream, const tipo_classe &c);
```

Sobrecarregando o operador []

O operador [], de índice de array, também pode ser sobrecarregado. Isto nos permite utilizar índices de array com os objetos para retornar dados do armazenamento interno do objeto. Para sobrecarregar este operador devemos criar um método da seguinte forma:

```
tipo operator[](int index);
```

O índice do operador é passado para o método pelo seu argumento inteiro, no nosso caso *index*.

Sobrecarregando o operador =

A sobrecarga do operador de atribuição, =, tem a mesma importância e uso da criação de construtores de cópia. Caso este operador não seja sobrecarregado, o compilador utiliza o operador padrão, que simplesmente copia o objeto membro a membro, causando os mesmos problemas com membros ponteiro que vimos anteriormente. Para sobrecarregar este operador, criamos um método na classe com a seguinte sintaxe:

```
tipo_da_classe & operator=(const tipo_da_classe & objeto);
```

Note que se passarmos o argumento por referência, devemos retornar o objeto por referência para que possamos encadear o operador sucessivamente.

Modelos de classes

O C++ permite criar modelos de classes (*class templates*) da mesma forma que criamos modelos de função, para dar uma maior generalização ao nosso código. Estes modelos nos permitem usar um mesmo algoritmo para vários tipos de dados, evitando reescrever uma classe para cada tipo. Para criar modelos de classes, procedemos de forma análoga à criação de modelos de função:

```
template <lista_de_parâmetros>
class nome_da_classe
{
.
.
.
};
```

Abaixo veremos a classe `TStack`, uma pilha genérica.

```
template <typename T>
class TStack
{
    private:
    T element;
    TStack *next;

    public:
    TStack();
    TStack(const T &e);
    TStack(const TStack &stack);
    ~TStack();
    TStack & operator=(const TStack &stack);
```

```
    bool isEmpty();
    void push(T &e);
    T pop();
};

template <typename T> TStack<T>::TStack()
{
    next = NULL;
}

template <typename T> TStack<T>::TStack(const T &e)
{
    element = e;
    next = NULL;
}

template <typename T> TStack<T>::TStack(const TStack &stack)
{
    element = stack.element;

    if (stack.next != NULL)
        next = new TStack(stack.next);
    else
    next = new TStack();
}

template <typename T> TStack<T>::~TStack()
{
    if (next != NULL)
        delete next;
}

template <typename T> T& TStack<T>::operator=( const TStack &stack)
{
    element = stack.element;

    if (stack.next != NULL)
        next = new TStack(stack.next);
    else
        next = new TStack();

    return *this;
}

template <typename T> bool TStack<T>::isEmpty()
{
    bool ret = false;
    if (next == NULL)
        ret = true;

    return ret;
}

template <typename T> void TStack<T>::push(const T &e)
{
```

```
    if (next != NULL)
    {
        next->push(e);
    }
    else
    {
        element = e;
        next = new TStack();
    }
}

template <typename T> T TStack<T>::pop()
{
    T e;

    if (next != NULL)
    {
        if (next->next != NULL)
            e = next->pop();
        else
        {
            e = next->element;
            delete next;
        }
    }
    else
    {
        e = NULL;
    }

    return e;
}
```

Devemos notar os seguintes aspectos nesta implementação: por utilizar membros ponteiros, a classe deve ter um construtor de cópia, um destrutor (para liberar a memória alocada para o ponteiro) e o operador de atribuição sobrecarregado (para garantir a cópia do objeto na atribuição).
Além da palavra-chave **typename**, pode ser vista a declaração de uma classe *template* utilizando a palavra-chave **class** em seu lugar. Esta forma de implementação, no entanto, está em desuso, pois **typename** é muito mais genérico do que **class**.
Os modelos de classe também permitem mais de um tipo de dados como parâmetro do modelo. O seguinte código mostra como definir um modelo de classe dependente de dois tipos de dados:

```
Template <typename T1, typename T2>
Class nome_da_classe
{
.
.
.
};
```

Note que os outros parâmetros podem ser tipos de dados genéricos (declarados com **typename**) como também tipos específicos (**char** ou **int**, por exemplo).

Seção prática: Criando classes para o jogo

Agora que aprendemos a definir classes em C++, iremos modificar o nosso jogo para que seja baseado em classes. As classes sugeridas para o nosso jogo são:

`TScreen` – representando a tela do jogo.
`TBall, TBlock e TBar` – representando os sprites do jogo. Estas classes devem prover comunicação com `TScreen`, como métodos para serem dsenhados, etc.
`TGame` – classe para controlar a dinâmica do jogo. Se desejado, pode conter níveis.
`TPlayer` – possui métodos para interação com o jogador e guarda as suas informações.

Devemos também criar um arquivo que contém a função main e todos os objetos necessários para o funcionamento do jogo. As classes acima são apenas sugestões, e é importante perceber que podemos (na realidade devemos) criar classes auxiliares para representar tipos de dados utilizados pelas classes principais, por exemplo.
Note que provavelmente será necessário sobrecarregar operadores (podemos sobrecarregar os operadores >> e << para conectar a entrada e saída do jogo com os nossos objetos). Não esqueça também que devemos implementar as nossas classes com no mínimo:

- Construtor padrão
- Construtor de cópia
- Operador de igualdade sobrecarregado
- Destrutor

Capítulo IV

Engenharia de Software Orientada a Objetos

Engenharia de Software Orientada a Objetos

Além de poder definir objetos, C++ nos permite definir objetos a partir de outros objetos. É o conceito de herança. Quando uma classe (chamada de subclasse ou classe derivada) "herda" de outra classe (chamada superclasse ou classe base), a subclasse passa a possuir todos os membros da superclasse, podendo adicionar membros novos ou redefinir os antigos. Isto possibilita ao programador uma alta reusabilidade de código, como também permite modelar o sistema de uma forma mais fiel ao que acontece na realidade. Neste capítulo, veremos como utilizar herança em C++ e como aproveitar todos os benefícios que isto representa, desenvolvendo um código realmente orientado a objetos.

Herança em C++

Vejamos um exemplo de como utilizar herança em C++. Definiremos uma classe chamada TShape. A partir desta classe, criaremos TBlock (bloco) e TBall (bola):

```
class TShape
{
    .
    .
    .
};

class TBlock : public TShape
{
    .
    .
    .
};

class TBall : public TShape
{
    .
    .
    .
};
```

Como podemos perceber, para criar uma classe que herda de outra, basta apenas incluir:

```
: modificador nome_superclasse
```

após o nome da subclasse. O modificador serve para especificar como a superclasse será "vista" dentro da subclasse. O modificador **public** indica que os membros públicos da superclasse continuam sendo públicos na subclasse. Analogamente, os modificadores **protected** e **private** tornam os membros da superclasse protegidos ou privados na subclasse.

Overload de construtores, destrutores e métodos

Vimos no capítulo anterior, como sobrepor os métodos de uma classe. Quando criamos uma classe derivada, também podemos sobrepor os métodos da superclasse, inclusive os construtores e destrutores. Quando a superclasse não tem um construtor definido, ao criarmos um objeto da subclasse, o construtor da superclasse é chamado automaticamente. Mas quando temos construtores definidos na superclasse, devemos especificar

quais argumentos serão passados para esses construtores, quando um objeto da subclasse é criado. Isto é feito utilizando-se inicializadores:

```
class TShape
{
    protected:
        int x;
        int y;
        char *name;
    public:
        TShape(int x1, int y1, const char *n);
        ~TShape();
};

class TBlock : public TShape
{
    private:
        char *pattern;
    public:
        TBlock(int x1, int y1, const char *pat);
        ~TBlock();
};

TShape::TShape(int x1, int y1, const char *n)
{
    x = x1;
    y = y1;
    name = new char[strlen(n)+1];
    strcpy(name, n);
}

TShape::~TShape()
{
    delete[] name;
}

TBlock::TBlock(int x1, int y1, const char *pat)
       :TShape(x1, y1, "block")
{
    pattern = new char[strlen(pat)+1];
    strcpy(pattern, pat);
}

TBlock::~TBlock()
{
    delete[] pattern;
}
```

Note que utilizamos `:TShape(x1, y1, "block")` para chamar o construtor da superclasse. Os construtores das superclasses são sempre chamados antes dos construtores das subclasses. Com os destrutores ocorre o inverso: os destrutores das subclasses são chamados antes dos das superclasses.
A sobreposição de métodos ocorre de forma semelhante, a diferença é que, para chamarmos métodos da superclasse dentro da subclasse utilizamos: `nome_superclasse::metodo(parametros)`.

Métodos virtuais: *dynamic binding*

Como já vimos, é possível termos ponteiros para objetos de uma classe. O C++ também permite que ponteiros para uma superclasse apontem para objetos de uma classe derivada, apesar de não poder acessar os membros específicos desta. Este processo, de um ponteiro poder apontar para diversos objetos derivados distintos é chamado polimorfismo. Então você deve estar pensando: o que acontece se tivermos um método sobreposto na subclasse e, com um ponteiro da superclasse apontando para um objeto da subclasse, chamarmos o método? Está confuso? Vejamos um exemplo que ilustre isto melhor:

```
class TShape
{
    protected:
        int x;
        int y;
        char *name;
    public:
        TShape(int x1, int y1, const char *n);
        ~TShape();
        void draw();
};

class TBlock : public TShape
{
    private:
        char *pattern;
    public:
        TBlock(int x1, int y1, const char *pat);
        ~TBlock();
        void draw();
};

void TShape::draw()
{
    cout << “Drawing a shape” << endl;
}

void TBlock::draw()
{
    cout << “Drawing a block” << endl;
}

int main()
{
    TBlock block(10, 20, “####”);
    TShape shape(10, 20, “shape”);
    TShape *shapePtr;

    shapePtr = &shape;
    shapePtr->draw();

    shapePtr = &block;
    shapePtr->draw();

    return 0;
}
```

A saída do programa será:

```
Drawing a shape
Drawing a shape
```

Por que? Bem, o ponteiro `shapePtr` é do tipo `TShape` e, apesar de podermos apontar para um objeto do tipo `TBlock` (afinal um `TBlock` é um `TShape`), ao chamarmos um método através do ponteiro, o compilador fará uma ligação estática (*static biding* – ligação em tempo de compilação) da variável ponteiro com o método de sua classe. Isto fará com que o método da classe `TShape` seja chamado independentemente de para qual objeto o ponteiro aponte (afinal, em tempo de compilação não é possível saber o conteúdo do ponteiro).
Então, como chamaríamos o método correto? O C++ oferece uma alternativa que permite que a ligação do objeto com o método seja feita dinamicamente (*dynamic biding* – ligação em tempo de execução). Isto é feito, declarando o método da superclasse como **virtual**. Quando declaramos um método desta forma, estamos "avisando" ao compilador para fazer a ligação do objeto com o método em tempo de execução, para que o método do objeto correto seja chamado. Se na declaração da classe `TShape` do exemplo anterior, tivéssemos a seguinte linha:

```
virtual void draw();
```

A saída do programa seria:

```
Drawing a shape
Drawing a block
```

Isto quer dizer que `shapePtr->draw()` chama `TShape::draw()` quando `shapePtr` aponta para um objeto do tipo `TShape` e chama `TBlock::draw()` quando `shapePtr` aponta para um objeto do tipo `TBlock`.
Portanto, é padrão para os programadores C++ **definir sempre os destrutores de uma classe como virtuais**. Desta forma, estaremos assegurando que o destrutor correto será chamado quando um objeto de uma classe derivada for destruído.

Classes abstratas

Uma classe é dita abstrata quando possui um ou mais métodos que não foram implementados (chamados métodos **virtuais puros**). Isto significa que não podemos instanciar um objeto desta classe, mas apenas de suas classes derivadas, onde estes métodos serão implementados. Continuemos com nosso exemplo, em que temos uma classe `TShape` e uma classe `TBlock`. Vamos incluir as classes `TBall` e `TBar` que herdam de `TShape` também. Portanto, no nosso programa, todo *shape* ou é um bloco, ou uma bola, ou uma barra, e que qualquer objeto que herde da classe `TShape` tenha que possuir o método `draw()`. Então devemos declarar o método `draw()` de `TShape` como virtual puro, transformando a classe `TShape` em classe abstrata e forçando soas subclasses a implementarem o método. Isto é feito colocando `=0` no lugar do corpo do método (indicando que o método não possui implementação nesta classe) como veremos a seguir:

```
class TShape
{
    public:
        .
        .
        .
        virtual void draw()=0;
};

class TBlock : public TShape
{
```

```
    public:
        .
        .
        .
        void draw();
};

void TBlock::draw()
{
    cout << “Drawing a block” << endl;
}

class TBall : public TShape
{
    public:
        .
        .
        .
        void draw();
};

void TBall::draw()
{
    cout << “Drawing a ball” << endl;
}

class TBar : public TShape
{
    public:
        .
        .
        .
        void draw();
};

void TBar::draw()
{
    cout << “Drawing a bar” << endl;
}
```

Como vimos, todas as classes derivadas de TShape deverão implementar o método `draw()`.

Herança múltipla

O C++ permite que uma classe herde de várias outras, incorporando os membros das duas classes. A herança múltipla é contestada por alguns programadores, pois isto facilita o conflito entre os membros das classes. Vejamos como declaramos uma classe com herança múltipla:

```
class TScreenObjects : public TBall, public TBar, public TBlock
{
.
.
.
};
```

Portanto, a classe `TScreenObjects` possui os membros de todas as classes que ela herda. Mas o que acontece se chamarmos o método `draw()`, que existe em todas as superclasses de `TScreenObjects`? Neste caso, o compilador geraria um erro, pois não saberia de que classe chamar o método. Uma solução seria fazer uma herança privada das superclasses (de forma que seus membros não ficassem visíveis a partir da subclasse) e definir métodos para chamar o método `draw()` de cada superclasse:

```
class TScreenObjects : private TBall, private TBar, private TBlock
{
    public:
        drawBall() { TBall::draw(); }
        drawBar() { TBar::draw(); }
        drawBlock() { TBlock::draw(); }
};
```

Outro problema que enfrentamos com a herança múltipla é que, quando criamos um objeto da subclasse, estamos criando um objeto interno de cada superclasse. E se as superclasses herdarem de uma mesma classe-base? Serão criados vários objetos internos da classe-base (note que serão criados vários **objetos internos idênticos**), e quando chamarmos um método desta classe-base, o compilador não saberá de qual dos objetos internos chamar o método. Para resolver isto, as classes devem fazer uma **herança virtual** da classe-base, de forma que um único objeto da classe base será criado:

```
class TBall : public virtual TShape
{
.
.
.
};

class TBar : public virtual TShape
{
.
.
.
};

class TBlock : public virtual TShape
{
.
.
.
};

class TScreenObjects : private TBall, private TBar, private TBlock
{
.
.
.
};
```

Usar herança virtual elimina o conflito de usarmos classes-base compartilhadas. Mas e se as superclasses de `TScreenObjects` possuírem construtores que passem argumentos distintos para a classe `TShape`? Como o programa só pode criar um objeto compartilhado da classe `TShape`, o C++ desativa a passagem automática de parâmetros (com inicializadores) das classes intermediárias para a classe-base virtual e no seu lugar chama o construtor padrão desta classe. Vejamos este exemplo:

```
TScreenObjects(int x1, int y1, const char *pattern) : TBall(x1, y1,
pattern), TBar(x1, y1, pattern), TBlock(x1, y1, pattern) {};
```

Neste exemplo, os construtores das classes `TBall, TBar` e `TBlock` não terão permissão para passar os argumentos para a classe `TShape`. Para que este exemplo funcione corretamente, devemos chamar também o construtor da classe `TShape`:

```
TScreenObjects(int x1, int y1, const char *pattern) : TShape(x1,
y1, "screenObject"), TBall(x1, y1, pattern), TBar(x1, y1, pattern),
TBlock(x1, y1, pattern) {};
```

Assim garantimos a inicialização correta dos objetos da classe `TScreenObjects`.

Exceções

Exceções são erros de tempo de execução, ou seja, não são *bugs* (erros da lógica do programa), mas sim erros gerados pela manipulação de dados inadequados (geralmente fornecidos por usuários). Vejamos um exemplo deste tipo de erro:

```
int main ()
{
    int num1, num2;

    cout << "Digite dois números: " ;
    cin  >> num1 >> num2;

    cout << "A divisão do número " << num1 << "pelo número ";
    cout << num2 << "é " << (num1/num2) << endl;

    return 0;
}
```

Neste exemplo, apesar da lógica estar correta, poderíamos ter um erro de tempo de execução (runtime error) caso o usuário informasse o valor zero para o segundo número.
Para estes tipos de problema é que existem as exceções. O mecanismo de exceções nos permite "tentar" executar (**try**) um determinado código, caso o código não possa ser executado "lançamos" (**throw**) uma exceção que será "capturada" (**catch**) e tratada em outra parte do código. Vejamos como isto funciona:

```
int main ()
{
    int num1, num2;

    cout << "Digite dois números: " ;
    cin  >> num1 >> num2;

    try
    {
        if (num2 == 0)
        {
            throw string("Erro: divisão por zero.");
        }

        cout << "A divisão do número " << num1 << "pelo número ";
        cout << num2 << "é " << (num1/num2) << endl;
```

```
    }
    catch(string str)
    {
        cout << str << endl;
    }

    return 0;
}
```

No código acima, a parte "sensível" (onde pode ocorrer uma exceção), é colocada dentro de um bloco **try**. Caso haja um erro, lançamos uma exceção com **throw**, e o fluxo do programa será desviado para fora do bloco **try**. A partir deste momento, o programa irá procurar um bloco **catch** que capture uma exceção do mesmo tipo que a lançada (no nosso caso string). Caso não encontre, passa o fluxo do programa à função/método que chamou o código que lançou a exceção, subindo pela pilha de chamadas de funções (*call stack*) até a função main. Se na função main também não houver um bloco **catch** para este tipo de exceção, o programa é encerrado.
Como você já deve ter percebido, podemos lançar uma exceção de qualquer tipo de dado (inclusive classes específicas para tratamento de exceções, com código e mensagens de erro, etc..). O ANSI/ISO C++ define uma classe de exceção padrão chamada `exception`. Esta classe possui um método constante chamado `what()` que retorna um **`const char*`** contendo uma mensagem de erro da exceção. Portanto, podemos criar nossas classes de exceção herdando da classe `exception` e sobrescrevendo o método `what()` para que exiba a mensagem de erro adequada. Vejamos um exemplo:

```
class TDivideByZeroException : public exception
{
    private:
        char *msg;
    public:
        const char *what() const {return msg;}
        TDivideByZeroException() {msg = "Erro: divisão por zero.";}
};

int main ()
{
    int num1, num2;

    cout << "Digite dois números: " ;
    cin  >> num1 >> num2;

    try
    {
        if (num2 == 0)
        {
            throw TDivideByZeroException();
        }

        cout << "A divisão do número " << num1 << "pelo número ";
        cout << num2 << "é " << (num1/num2) << endl;
    }
    catch(exception &e)
    {
        cout << e.what() << endl;
    }

    return 0;
```

```
}
```

Podemos observar que o bloco **catch** (exception &e){...} captura uma exceção do tipo exception bem como todas que herdam dela, como TDivideByZeroException.
Mas como tratar exceções inesperadas? Por padrão, o programa chama a função **unexpected** quando encontra uma exceção que não é capturada por nenhum bloco catch. Esta função, por sua vez chama a função **terminate** que chama **abort**. O arquivo de cabeçalho exception possui as funções set_unexpected e set_terminate, para definir funções de tratamento de exceções inesperadas. Podemos definir uma função com set_unexpected e esta função pode levantar uma exceção que possa ser capturada por algum bloco catch ou encerrar o programa, chamando **terminate**, **exit** ou **abort**. Isto é uma forma de converter uma exceção inesperada para uma exceção que possamos tratar. Podemos também utilizar a função set_terminate para definir uma função a ser chamada antes que o programa termine de forma anormal. Exemplo:

```
void exitFunction()
{
    cout << “Antes de terminar, imprimo isto.” << endl;
    exit(1);
}
int main ()
{
    set_terminate(exitFunction);
    try
    {
        throw string(“Erro”);
    }
    catch(float f){}

    return 0;
}
```

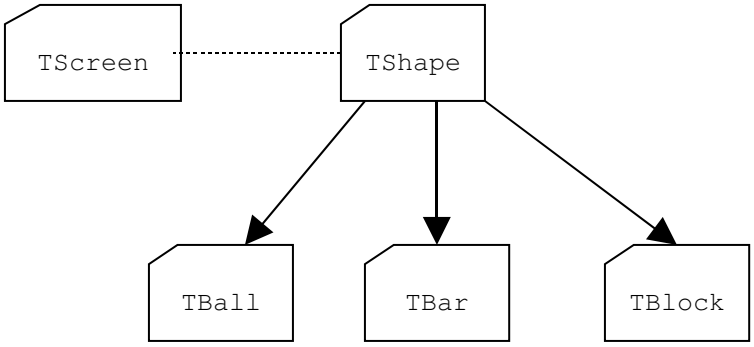
Neste caso, como a exceção não foi capturada, o programa terminou, mas a função exitFunction() foi chamada antes.
Podemos também criar blocos catch que capturem literalmente qualquer coisa. Isto é feito da seguinte forma:

```
catch (...)
{
.
.
.
}
```

Assim, qualquer exceção levantada será capturada por este bloco.

Seção prática: Utilizando herança

Nesta seção iremos "incrementar" o nosso jogo utilizando herança de classes. Podemos criar uma classe abstrata, TShape que agrupe os atributos comuns às classes TBall, TBlock e TBar. Esta classe também deve ter o conceito de "desenhável", ou seja, todo objeto que precise ser desenhado na tela, pode herdar de TShape e a classe TScreen receberá um objeto TShape no método de desenhar objetos na tela:



Podemos, pensando analogamente, verificar todo o nosso programa em busca de grupos de classes que podem ser generalizadas, aumentando a reusabilidade do nosso código.

Capítulo V

Recursos avançados do C++: RTTI e STL

A RTTI (runtime type information)

Com a introdução do polimorfismo em tempo de execução (um ponteiro pode apontar para elementos do seu tipo e dos tipos derivados), às vezes torna-se útil saber a que classe pertence o objeto para o qual o ponteiro está apontando. Isto é feito com os elementos da RTTI (*runtime type information* – informações de tipo em tempo de execução) de C++. Os componentes da RTTI são:

- O operador **dynamic_cast**: faz o cast entre um ponteiro do tipo base para um ponteiro do tipo derivado (se não for possível, retorna NULL).
- O operador **typeid**: que retorna informações sobre o tipo de dados especificado.
- A classe **type_info**: armazena informações sobre os tipos de dados retornadas por **typeid**.

Note que para utilizar os elementos da RTTI de C++ é necessário incluir o arquivo de cabeçalho **typeinfo**.

O operador dynamic_cast

Este operador é utilizado quando necessitamos converter um ponteiro de uma classe-base para uma classe derivada, para poder utilizar os membros da classe derivada. Isto só é possível, se o ponteiro da classe-base estiver apontando para um objeto da classe derivada. Se não for o caso, o operador retorna NULL (ponteiro nulo). Vejamos um exemplo:

```
int main()
{
    TBall bola;
    TShape *shapePtr = &bola;
    TBall *ballPtr = NULL;

    ballPtr = dynamic_cast <TBall *> (shapePtr);

    if (ballPtr != NULL)
    {
        cout << “cast realizado com suceso!” << endl;
    }
    else
    {
        cout << “não foi possível realizar o cast!” << endl;
    }
    return 0;
}
```

Note que um ponteiro do tipo TShape pode apontar tanto para um objeto TShape quanto para objetos das classes derivadas, como TBall. O que o operador dynamic_cast faz é utilizar as informações de tempo de execução de cada tipo para saber se é possível realizar o cast.

Utilizando o operador typeid e a classe type_info

O operador **typeid** é utilizado para obter informações de um determinado tipo em tempo de execução. Ele retorna um objeto da classe **type_info**, com as informações do tipo. Os membros da classe **type_info** variam de acordo com o compilador (pois os fabricantes estão sempre querendo “incrementar” as capacidades dos seus compiladores). O ANSI/ISO C++ determina que esta classe deve ter um construtor, um construtor de cópia, um operador de atribuição, um destrutor e um método chamado `name()`, que retorna o nome do tipo do objeto. Vejamos um exemplo:

```
int main()
{
    TBall bola;
    TShape *shapePtr = &bola;
    TBall *ballPtr = NULL;


    if (typeid(*shapePtr) == typeid(TBall))
    {
        cout << “cast dinâmico pode ser realizado!” << endl;
        ballPtr = dynamic_cast <TBall *> (shapePtr);
    }
    else
    {
        cout << “cast dinâmico não pode ser realizado!” << endl;
    }

    cout << “o tipo do objeto apontado por shapePtr é ”;
    cout << typeid(*shapePtr).name() << endl;

    return 0;
}
```

Neste caso, comparamos o tipo do objeto apontado por `shapePtr` com o tipo `TBall`, para nos certificarmos que `shapePtr` aponta para um objeto do tipo `TBall`, antes de fazer o cast dinâmico. No fim do programa exibimos o nome do tipo do objeto apontado por `shapePtr`, que é `TBall` (em alguns compiladores existe uma variação deste nome, e você poderia ver `4TBall`, por exemplo).

A STL (standard template library)

Nos capítulos anteriores, vimos como criar classes template, reutilizando algoritmos com diversos tipos de dados. Além de permitir o uso de classes template, o C++ oferece um conjunto de classes template já prontas para serem utilizadas pelos programadores. Este conjunto de classes template é chamado de STL (standard template library – biblioteca de modelos padrão). A STL faz parte do ANSI/ISO C++, de forma que todo compilador que deseje estar de acordo com o C++ padrão ANSI/ISO, deverá dar suporte à STL na sua totalidade, embora isto seja difícil de acontecer.
A STL é dividida em quatro grupos de classes:

- **Contêineres**: classes utilizadas para armazenar e manipular conjuntos de objetos.
- **Iteradores**: ponteiros especiais para os contêineres da STL, que nos permitem percorrer os dados do contêiner.
- **Algoritmos**: conjunto de funções que podemos aplicar aos contêineres da STL.
- **Objetos-função**: podemos personalizar o comportamento de alguns algoritmos da STL, utilizando objetos-função, ou functores.

Veremos agora cada um dos grupos de elementos da STL detalhadamente.

Contêineres da STL

A STL do C++ possui onze contêineres. Antes de conhecê-los, veremos os tipos comuns a todos os contêineres, que são utilizados para criar nossas variáveis (na lista abaixo, contêiner representa o nome do contêiner):

- `contêiner::value_type` : retorna o tipo do elemento do contêiner.
- `contêiner::reference` : retorna uma referência para o contêiner. É equivalente a `T&`.
- `contêiner::const_reference` : retorna uma referência constante para o contêiner. É equivalente a `const T&`.
- `contêiner::iterator` : o tipo do iterator do contêiner. Um iterator é uma generalização para `T*`.
- `contêiner::const_iterator` : o tipo do iterator constante do contêiner. É uma generalização para `const T*`.
- `contêiner::difference_type` : um tipo inteiro com sinal que representa a distância entre iteradores.
- `contêiner::size_type` : um tipo inteiro sem sinal que representa o tamanho de objetos de dados, número de elementos e índices.

Além destes tipos, existem um conjunto de métodos que são comuns a todos os contêineres (exceto bitset) e serão mostrados abaixo:

- `begin` : retorna um iterador para o primeiro elemento do contêiner.
- `end` : retorna um iterador para o elemento após o último, do contêiner.
- `rbegin` : retorna um iterador reverso para o primeiro elemento do contêiner.
- `rend` : retorna um iterador reverso para o elemento após o último, do contêiner.
- `size` : retorna o número de elementos do contêiner.
- `maxsize` : retorna o tamanho máximo do contêiner.
- `empty` : retorna true se o contêiner estiver vazio.
- `swap` : permuta o conteúdo de dois contêineres.
- Operadores condicionais `==, !=, <, >, <= e >=` : utilizados para comparar dois contêineres.

Agora que conhecemos um pouco dos contêineres da STL, veremos cada um detalhadamente. Abaixo segue uma lista com o nome, descrição, interface e o arquivo de cabeçalho associado a cada contêiner:

<table>
<tr><th colspan="2">vector</th></tr>
<tr><td>Descrição</td><td>Representa um array unidimensional. A principal vantagem de utilizar este contêiner, em vez de um array, é que podemos alterar o seu tamanho facilmente em tempo de execução.</td></tr>
<tr><td>Arquivo de cabeçalho</td><td>vector</td></tr>
<tr><td>Interface</td><td>

```
template <class T, class Allocator = allocator<T> >
class vector
{
public:
// Typedefs
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference
                 const_reference;
class iterator;
class const_iterator;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type
                 difference_type;
typedef typename std::reverse_iterator<iterator>
                 reverse_iterator;

typedef typename std::reverse_iterator<const iterator>
                 const_reverse_iterator;

// Construtores/Cópia/Destrutores
explicit vector (const Allocator& = Allocator());
explicit vector (size_type, const Allocator& =
                 Allocator ());
vector (size_type, const T&, const Allocator& =
        Allocator());
vector (const vector<T, Allocator>&);
template <class InputIterator>
         vector (InputIterator, InputIterator,
         const Allocator& = Allocator ());
~vector ();
vector<T,Allocator>& operator= (const
                     vector<T, Allocator>&);

template <class InputIterator>
void assign (InputIterator first, InputIterator last);
void assign (size_type, const);
allocator_type get_allocator () const;

// Iterators
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
```

</td></tr>
</table>

```
reverse_iterator rend ();
const_reverse_iterator rend () const;

// Capacidade
size_type size () const;
size_type max_size () const;

void resize (size_type);
void resize (size_type, T);
size_type capacity () const;
bool empty () const;
void reserve (size_type);

// Acesso aos elementos
reference operator[] (size_type);
const_reference operator[] (size_type) const;
reference at (size_type);
const_reference at (size_type) const;
reference front ();
const_reference front () const;
reference back ();
const_reference back () const;

// Modificadores
void push_back (const T&);
void pop_back ();
iterator insert (iterator, const T&);

void insert (iterator, size_type, const T&);
template <class InputIterator>
void insert (iterator, InputIterator, InputIterator);
iterator erase (iterator);
iterator erase (iterator, iterator);
void swap (vector<T, Allocator>&);
void clear()
};
```

list	
Descrição	Representa uma lista duplamente encadeada. Devemos utilizar este contêiner quando necessitamos de freqüentes inserções e remoções no meio da lista, pois este contêiner oferece um tempo de acesso constante aos elementos, independente do tamanho da lista.
Arquivo de cabeçalho	`list`
Interface	`template <class T, class Allocator = allocator<T> > class list { public: // typedefs class iterator; class const_iterator; typedef typename Allocator::reference reference; typedef typename Allocator::const_reference const_reference; typedef typename Allocator::size_type size_type; typedef typename Allocator::difference_type`

```
                    difference_type;
   typedef T value_type;
   typedef Allocator allocator_type;
   typedef typename std::reverse_iterator<iterator>
                    reverse_iterator;
   typedef typename
             std::reverse_iterator<const_iterator>
             const_reverse_iterator;

// Construtores/Cópia/Destrutores
   explicit list (const Allocator& = Allocator());
   explicit list (size_type);
   list (size_type, const T&, const Allocator& =
         Allocator())
   template <class InputIterator> list (InputIterator,
            InputIterator, const Allocator& =
            Allocator());

   list(const list<T, Allocator>& x);
   ~list();
   list<T,Allocator>& operator= (const
       list<T,Allocator>&);
   template <class InputIterator> void assign
            (InputIterator, InputIterator);
   void assign (size_type n, const T&);
   allocator_type get allocator () const;

// Iterators
   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;

   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacidade
   bool empty () const;
   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize (size_type, T);

// Acesso aos elementos
   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

// Modificadores
   void push_front (const T&);
   void pop_front ();
   void push_back (const T&);
   void pop_back ();
```

```
   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
   template <class InputIterator>
            void insert (iterator, InputIterator,
            InputIterator);
   iterator erase (iterator);
   iterator erase (iterator, iterator);
   void swap (list<T, Allocator>&);
   void clear ();

// Operações especiais
   void splice (iterator, list<T, Allocator>&);
   void splice (iterator, list<T, Allocator>&,
                iterator);
   void splice (iterator, list<T, Allocator>&,
                iterator, iterator);
   void remove (const T&);
   template <class Predicate> void remove_if
            (Predicate);
   void unique ();
   template <class BinaryPredicate>
             void unique (BinaryPredicate);
   void merge (list<T, Allocator>&);
   template <class Compare>
            void merge (list<T, Allocator>&, Compare);
   void sort ();
   template <class Compare> void sort (Compare);
   void reverse();
};
```

deque	
Descrição	Do inglês ***d**ouble **e**nded **que**ue,* representa uma fila com dois finais, ou seja, podemos adicionar ou remover elementos em ambas as extremidades. Possui operações de inserção e apagamento com tempo constante para o início e final da fila, mas o tempo de acesso cresce linearmente com o tamanho da fila para acessos a elementos do meio.
Arquivo de cabeçalho	`deque`
Interface	

```
template <class T, class Allocator = allocator<T> >
 class deque
 {
public:
 // Typedefs
   class iterator;
   class const_iterator;
   typedef T value_type;
   typedef Allocator allocator_type;
   typedef typename
           Allocator::reference reference;
   typedef typename
           Allocator::const_reference const_reference;
   typedef typename
           Allocator::size_type size_type;
   typedef typename
           Allocator::difference_type difference_type;
   typedef typename
```

```
            std::reverse_iterator<iterator>
            reverse_iterator;
   typedef typename
            std::reverse_iterator<const_iterator>
            const_reverse_iterator;

// Construtores/Cópia/Destrutores
   explicit deque (const Allocator& = Allocator());
   explicit deque (size_type);
   deque (size_type, const T& value,
          const Allocator& = Allocator ());
   deque (const deque<T,Allocator>&);
   template <class InputIterator>
           deque (InputIterator, InputIterator,
           const Allocator& = Allocator ());
   ~deque ();
   deque<T,Allocator>& operator=
                      (const deque<T,Allocator>&);
   template <class InputIterator>
            void assign (InputIterator, InputIterator);
   void assign (size_type, const T&);
   allocator_type get allocator () const;

 // Iterators
   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;

   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacidade
   size_type size () const;
   size_type max_size () const;
   void resize (size_type);
   void resize (size_type, T);
   bool empty () const;

// Acesso aos elementos
   reference operator[] (size_type);
   const_reference operator[] (size_type) const;
   reference at (size_type);
   const_reference at (size_type) const;

   reference front ();
   const_reference front () const;
   reference back ();
   const_reference back () const;

// Modificadores
   void push_front (const T&);
   void push_back (const T&);
   iterator insert (iterator, const T&);
   void insert (iterator, size_type, const T&);
```

```
   template <class InputIterator>
            void insert (iterator, InputIterator,
            InputIterator);
   void pop_front ();
   void pop_back ();
   iterator erase (iterator);
   iterator erase (iterator, iterator);

   void swap (deque<T, Allocator>&);
   void clear();
};
```

queue	
Descrição	Representa uma fila em que podemos fazer inserções no início e remover elementos do fim. É uma forma restrita do contêiner deque.
Arquivo de cabeçalho	`Queue`
Interface	

```
template <class T, class Container = deque<T> >
 class queue
 {
public:
// typedefs
   typedef typename Container::value_type value_type;
   typedef typename Container::size_type size_type;
   typedef Container container_type;

// Construtores/Cópia/Destrutores
   explicit queue (const Container& = Container());

// Acesso
   bool empty () const;
   size_type size () const;
   value_type& front ();
   const value_type& front () const;
   value_type& back ();
   const value_type& back () const;

   void push (const value_type&);
   void pop ();
};
```

priority_queue	
Descrição	Representa uma fila de prioridade. A ordem dos elementos é determinada pelo operador menor que, < ou pelo comparador **compare**. Isto significa que os objetos guardados neste contêiner devem ter este operador sobrecarregado além de um construtor de cópia, destrutor e operador =.
Arquivo de cabeçalho	`queue`
Interface	

```
template <class T, class Container = vector<T>,
          class Compare = less<typename
                              Container::value_type> >
 class priority_queue
 {
public:
// typedefs
   typedef typename Container::value_type value_type;
```

```
   typedef typename Container::size_type size_type;
   typedef Container container_type;

//  Construct
   explicit priority_queue (const Compare& = Compare(),
                            const Container& =
                            Container());
   template <class InputIterator>
     priority_queue (InputIterator first,
                     InputIterator last,
                     const Compare& = Compare(),
                     const Container& = Container());
   bool empty () const;
   size_type size () const;
   const value_type& top () const;
   void push (const value_type&);
   void pop();
};
```

stack	
Descrição	Representa uma pilha, com operações para empilhar, desempilhar e ver o topo da pilha.
Arquivo de cabeçalho	`stack`
Interface	`template <class T, class Container = deque<T> >` `class stack` `{` `public:` `// typedefs` `typedef typename Container::value_type value_type;` `typedef typename Container::size_type size_type;` `typedef Container container_type;` `// Construtor` `explicit stack (const Container& = Container());` `// Acesso` `bool empty () const;` `size_type size () const;` `value_type& top ();` `const value_type& top () const;` `void push (const value_type&);` `void pop ();` `};`

set	
Descrição	Representa um conjunto de objetos, semelhante a um conjunto matemático, sendo que cada elemento do conjunto deve ser único. Implementa as operações comuns sobre conjuntos, como união, interseção, etc.
Arquivo de cabeçalho	`set`
Interface	`template <class Key, class Compare = less<Key>,` `class Allocator = allocator<Key> >` `class set` `{`

```
public:

// typedefs
typedef Key key_type;
typedef Key value_type;
typedef Compare key_compare;
typedef Compare value_compare;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference
                 const_reference;
class iterator;
class const_iterator;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type
                 difference_type;

typedef typename std::reverse_iterator<iterator>
                      reverse_iterator;
typedef typename std::reverse_iterator<const_iterator>
                      const_reverse_iterator;

// Construtores/Cópia/Destrutor
explicit set (const Compare& = Compare(),
              const Allocator& = Allocator ());
template <class InputIterator>
set (InputIterator, InputIterator,
     const Compare& = Compare(),
     const Allocator& = Allocator ());
set (const set<Key, Compare, Allocator>&);

~set ();
set<Key, Compare, Allocator>& operator=
   (const set <Key, Compare, Allocator>&);
allocator_type get_allocator () const;

// Iterators
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator rend ();
const_reverse_iterator rend () const;

// Capacidade
bool empty () const;
size_type size () const;
size_type max_size () const;

// Modificadores
pair<iterator, bool> insert (const value_type&);
iterator insert (iterator, const value_type&);
template <class InputIterator>
         void insert (InputIterator, InputIterator);
void erase (iterator);
```

```
size_type erase (const key_type&);
void erase (iterator, iterator);
void swap (set<Key, Compare, Allocator>&);
void clear ();

// Observação de elementos
key_compare key_comp () const;
value_compare value_comp () const;

// Operações sobre conjuntos
size_type count (const key_type&) const;

pair<iterator, iterator> equal_range (const  key_type&)
const;
iterator find (const key_type&) const;
iterator lower_bound (const key_type&) const;
iterator upper_bound (const key_type&) const;
};
```

multiset	
Descrição	O contêiner multiset representa um conjunto (semelhante ao set) em que os elementos não precisam ser únicos.
Arquivo de cabeçalho	`set`
Interface	

```
template <class Key, class Compare = less<Key>,
          class Allocator = allocator<Key> >
 class multiset
 {
public:
// typedefs
   typedef Key key_type;
   typedef Key value_type;
   typedef Compare key_compare;
   typedef Compare value_compare;
   typedef Allocator allocator_type;
   typedef typename Allocator::reference reference;
   typedef typename
           Allocator::const_reference const_reference;
   class iterator;
   class const_iterator;

   typedef typename Allocator::size_type size_type;
   typedef typename
           Allocator::difference_type difference_type;
   typedef typename std::reverse_iterator<iterator>
                         reverse_iterator;
   typedef typename
                 std::reverse_iterator<const_iterator>
                         const_reverse_iterator;

// Construtores/Cópia/Destrutores
   explicit multiset (const Compare& = Compare(),
                      const Allocator& = Allocator());

   template <class InputIterator>
    multiset (InputIterator, InputIterator,
```

```
                const Compare& = Compare(),
                const Allocator& = Allocator());
   multiset (const multiset<Key, Compare, Allocator>&);
   ~multiset ();
   multiset<Key, Compare, Allocator>&
            operator= (const multiset<Key,
                       Compare, Allocator>&);

// Iterators
   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;

   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;
   reverse_iterator rend ();
   const_reverse_iterator rend () const;

// Capacidade
   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modificadores
   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
            void insert (InputIterator, InputIterator);
   void erase (iterator);
   size_type erase (const key_type&);

   void erase (iterator, iterator);
   void swap (multiset<Key, Compare, Allocator>&);
   void clear ();

// Observação de elementos
   key_compare key_comp () const;
   value_compare value_comp () const;

// Operações sobre Multiset
   iterator find (const key_type&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range
       (const key_type&) const;

   };
```

map	
Descrição	Este contêiner representa um mapa, associação chave/valor em que se usa a chave para acessar o valor correspondente. Neste contêiner, cada chave deve ser única, de modo que não podemos ter chaves repetidas.
Arquivo de cabeçalho	`map`
Interface	`template <class Key, class T, class Compare = less<Key>`

```
          class Allocator = allocator<pair<const Key,
          T> > >
class map
{
public:

// typedefs
   typedef Key key_type;
   typedef typename Allocator::pointer pointer;
   typedef typename Allocator::const_pointer
                    const_pointer;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;
   typedef typename Allocator::reference reference;
   typedef typename
           Allocator::const_reference  const_reference;
   class iterator;
   class const_iterator;
   typedef typename
           Allocator::size_type size_type;
   typedef typename
           Allocator::difference_type  difference_type;
   typedef typename std::reverse_iterator<iterator>
                         reverse_iterator;
   typedef typename
           std::reverse_iterator<const_iterator>
                         const_reverse_iterator;

   class value_compare
      : public binary_function<value_type, value_type,
                               bool>
   {
     friend class map<Key, T, Compare, Allocator>;
     protected :
       Compare comp;
       value_compare(Compare c): comp(c) {}
     public :
       bool operator() (const value_type&,
                        const value_type&) const;
   };

// Construtores/Cópia/Destrutores
   explicit map (const Compare& = Compare(),
                 const Allocator& = Allocator ());
   template <class InputIterator>
    map (InputIterator, InputIterator,
         const Compare& = Compare(),
         const Allocator& = Allocator ());
   map (const map<Key, T, Compare, Allocator>&);
   ~map();
   map<Key, T, Compare, Allocator>&
      operator= (const map<Key, T, Compare,
                 Allocator>&);
   allocator_type get_allocator () const;
```

```
// Iterators
   iterator begin();
   const_iterator begin() const;
   iterator end();
   const_iterator end() const;
   reverse_iterator rbegin();
   const_reverse_iterator rbegin() const;

   reverse_iterator rend();
   const_reverse_iterator rend() const;

// Capacidade
   bool empty() const;
   size_type size() const;
   size_type max_size() const;

// Acesso aos elementos
   mapped_type& operator[] (const key_type&);

// Modificadores
   pair<iterator, bool> insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
            void insert (InputIterator, InputIterator);
   void erase (iterator);
   size_type erase (const key_type&);

   void erase (iterator, iterator);
   void swap (map<Key, T, Compare, Allocator>&);
   void clear();

// Observação de elementos
   key_compare key_comp() const;
   value_compare value_comp() const;

// Operações sobre map
   iterator find (const key_value&);
   const_iterator find (const key_value&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const
                                         key_type&);
   pair<const_iterator, const_iterator>
     equal_range (const key_type&) const;
};
```

multimap	
Descrição	O contêiner multimap representa um mapa, semelhante ao contêiner map, mas possibilitando o armazenamento de múltiplos valores para uma mesma chave.
Arquivo de cabeçalho	`map`
Interface	`template <class Key, class T, class Compare = less<Key>, class Allocator = allocator<pair<const Key, T> > >`

```
 class multimap
 {
public:
// typedefs
   typedef Key key_type;
   typedef T mapped_type;
   typedef pair<const Key, T> value_type;
   typedef Compare key_compare;
   typedef Allocator allocator_type;
   typedef typename Allocator::reference reference;
   typedef typename
           Allocator::const_reference  const_reference;
   class iterator;
   class const_iterator;
   typedef typename Allocator::size_type size_type;
   typedef typename
           Allocator::difference_type difference_type;
   typedef typename std::reverse_iterator<iterator>
                         reverse_iterator;
   typedef typename
                  std::reverse_iterator<const_iterator>
                         const_reverse_iterator;
 class value_compare
 : public binary_function<value_type, value_type, bool>
    {
     friend class multimap<Key, T, Compare, Allocator>;
     protected :
       Compare comp;
       value_compare (Compare C) : comp(c) {}
     public :
       bool operator() (const value_type&,
                        const value_type&) const;
    };

// Construtores/Cópia/Destrutores
   explicit multimap (const Compare& = Compare(),
                      const Allocator& = Allocator());
   template <class InputIterator>
    multimap (InputIterator, InputIterator,
              const Compare& = Compare(),
              const Allocator& = Allocator());
   multimap (const multimap<Key, T, Compare,
             Allocator>&);
   ~multimap ();
   multimap<Key, T, Compare, Allocator>& operator=
       (const multimap<Key, T, Compare, Allocator>&);
   allocator_type get_allocator () const;

// Iterators
   iterator begin ();
   const_iterator begin () const;
   iterator end ();
   const_iterator end () const;
   reverse_iterator rbegin ();
   const_reverse_iterator rbegin () const;

   reverse_iterator rend ();
```

```
   const_reverse_iterator rend () const;

// Capacidade
   bool empty () const;
   size_type size () const;
   size_type max_size () const;

// Modificadores
   iterator insert (const value_type&);
   iterator insert (iterator, const value_type&);
   template <class InputIterator>
            void insert (InputIterator, InputIterator);
   void erase (iterator);
   size_type erase (const key_type&);
   void erase (iterator, iterator);
   void swap (multimap<Key, T, Compare, Allocator>&);

   void clear ();

// Observação de elementos
   key_compare key_comp () const;
   value_compare value_comp () const;

// Operações sobre Multimap
   iterator find (const key_type&);
   const_iterator find (const key_type&) const;
   size_type count (const key_type&) const;
   iterator lower_bound (const key_type&);
   const_iterator lower_bound (const key_type&) const;
   iterator upper_bound (const key_type&);
   const_iterator upper_bound (const key_type&) const;
   pair<iterator, iterator> equal_range (const
                                         key_type&);

   pair<const_iterator, const_iterator>
                  equal_range (const key_type&) const;
};
```

bitset	
Descrição	Este contêiner representa um conjunto de bits (bitset), que podem ser acessados individualmente com o operador colchetes, []. Oferece operações lógicas bit-a-bit bem como métodos de manipulação dos bits individualmente ou em conjunto.
Arquivo de cabeçalho	`bitset`
Interface	

```
template <size_t N>
class bitset
{
public:
// bit reference
  class reference
  {
       friend class bitset;
     public:
       ~reference();
       reference& operator= (bool);
       reference& operator= (const reference&);
       bool operator~() const;
```

```
        operator bool() const;
        reference& flip();
  };

// Construtores
  bitset ();
  bitset (unsigned long);
  template<class charT, class traits, class Allocator>
  explicit bitset (const basic_string<charT, traits,
                   Allocator>, typename basic_string
                   <charT, traits, Allocator>
                   ::size_type=0, typename basic_string
                   <charT, traits, Allocator>
                   ::size_type= basic_string<charT,
                   traits, Allocator>::npos);

  bitset (const bitset<N>&);
  bitset<N>& operator= (const bitset<N>&);

// Operadores bit-a-bit
   bitset<N>& operator&= (const bitset<N>&);
   bitset<N>& operator|= (const bitset<N>&);
   bitset<N>& operator^= (const bitset<N>&);
   bitset<N>& operator<<= (size_t);
   bitset<N>& operator>>= (size_t);

// Set, Reset, Flip
   bitset<N>& set ();
   bitset<N>& set (size_t, int = 1);
   bitset<N>& reset ();
   bitset<N>& reset (size_t);
   bitset<N> operator~() const;
   bitset<N>& flip ();
   bitset<N>& flip (size_t);

// Acesso aos elementos
   reference operator[] (size_t);
   unsigned long to_ulong() const;
   template<class charT, class traits, class Allocator>
            basic_string<charT, traits, Allocator>
            to_string();
   size_t count() const;
   size_t size() const;
   bool operator== (const bitset<N>&) const;
   bool operator!= (const bitset<N>&) const;
   bool test (size_t) const;
   bool any() const;
   bool none() const;
   bitset<N> operator<< (size_t) const;
   bitset<N> operator>> (size_t) const;
};

// Operadores não-membros
template <size_t N> bitset<N>
        operator& (const bitset<N>&, const bitset<N>&);
template <size_t N> bitset<N>
        operator| (const bitset<N>&, const bitset<N>&);
```

	`template <size_t N> bitset<N>` `operator^ (const bitset<N>&, const bitset<N>&);` `template <size_t N> istream&` `operator>> (istream&, bitset<N>&);` `template <size_t N> ostream&` `operator<< (ostream&, const bitset<N>&);`

O modelo ato_ptr

A biblioteca padrão do C++ possui um modelo de ponteiro, `auto_ptr`, que oferece desalocação automática da memória quando o ponetiro sai do escopo. Portanto, é ideal para utilizarmos ponteiros locais sem ter que nos preocuparmos em estar liberando memória quando sairmos do escopo. É importante lembrar que, como o ponteiro chama o operador **delete** automaticamente, não podemos alocar arrays com `auto_ptr`, pois o operador chamado é **delete** e não **delete[]**.
Abaixo segue um exemplo de código que utiliza o auto_ptr:

```
int main()
{
    auto_ptr<TBlock> block;
    block = new TBlock(10, 20, “####”);
    block->draw();
}
```

A classe template valarray

A classe valarray é um modelo de classe projetado para ser utilizado com valores numéricos. Semelhante ao contêiner vector, valarray é otimizada para permitir aplicar operações e funções matemáticas ao array, com o máximo de eficiência. Vejamos um exemplo de uso:

```
int main()
{

    valarray<int> array1(10);
    valarray<int> array2(10);

    for(int i = 0; i<10; i++)
    {
        array1[i] = i;
    }

    array2 = array1 + 10; // soma 10 a cada elemento de array1 e
                          // atribui a array2

    array1 += array2;  // soma cada elemento de array1 com array2

    for(int i = 0; i<10; i++)
    {
        cout << array1[i] << “ ”;
    }
}
```

É importante salientar que apenas os tipos numéricos podem ser utilizados com valarray, e que esta classe template também possui implementações das principais funções matemáticas, como seno, cosseno, etc.

Iteradores

A STL de C++ possui um conjunto de ponteiros genéricos utilizados para manipular os dados dos contêiners. Da mesma forma que o uso de contêineres nos permite utilizar os algoritmos independente do tipo de dados, os iteradores generalizam o uso dos algoritmos, independente do contêiner. O uso de iteradores é semelhante ao uso de ponteiros de forma que o código abaixo:

```
int nums[10];
int *ptr;

for (int i=0, ptr = nums; ptr != nums+10; i++, ptr++)
    *ptr = i;
```

Pode ser escrito desta forma:

```
vector<int> nums(10);
vector<int>::iterator itr;

for (int i=0, itr = nums.begin(); itr != nums.end(); i++, itr++)
    *itr = i;
```

Então, por que utilizar iteradores? Porque em contêineres diferentes, operações como se mover para o próximo elemento causam resultados diferentes também. Então, utilizamos iteradores com qualquer função STL sem nos preocuparmos com o tipo de contêiner em que os dados estão armazenados, pois o uso dos iteradores é o mesmo.
Vejamos alguns iteradores predefinidos da STL:

- **iterator**: iterador padrão.
- **ostream_iterator**: iterador de manipulação de fluxos ostream, como cout.
- **istream_iterator**: iterador de manipulação de fluxos istream, como cin.
- **reverse_iterator**: iterador que percorre os dados em sentido reverso.
- **insert_iterator**: iterador de inserção.
- **front_insert_iterator**: insere elementos em um contêiner a partir do início.
- **back_insert_iterator**: insere elementos em um contêiner a partir do fim.

Algoritmos da STL

Os algoritmos da STL trabalham junto com os iteradores para manipular os dados dos contêineres da STL. Como os iteradores são idependentes dos tipos dos contêineres, podemos aplicar os algoritmos aos iteradores de qualquer contêiner, pois os iteradores é que se encarregarão de percorrer internamente os dados contêineres. Vejamos os algoritmos da STL:

accumulate	
Descrição	acumula valores de operações matemáticas sucessivas.
Arquivo de cabeçalho	`numeric`
Interface	`template <class InputIterator, class T>` `T accumulate (InputIterator first,` `InputIterator last,` `T init);`

	`template <class InputIterator, class T, class BinaryOperation> T accumulate (InputIterator first, InputIterator last, T init, BinaryOperation binary_op);`

copy	
Descrição	Copia dados entre contêineres.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator, class OutputIterator> OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result); template <class BidirectionalIterator1, class BidirectionalIterator2> BidirectionalIterator2 copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);`

count e count_if	
Descrição	Conta o número de elementos de um contêiner. O algoritmo count_if conta os elementos do contêiner que satisfazem um predicado.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator, class T> typename iterator_traits<InputIterator>::difference_type count(InputIterator first, InputIterator last, const T& value); template <class InputIterator, class T, class Size> void count(InputIterator first, InputIterator last, const T& value, Size& n); template<class InputIterator, class Predicate> typename iterator_traits<InputIterator>::difference_type count_if(InputIterator first, InputIterator last, Predicate pred); template <class InputIterator, class Predicate, class Size> void count_if(InputIterator first, InputIterator last, Predicate pred, Size& n);`

equal	
Descrição	Compara os elementos de dois contêineres dentro de uma faixa delimitada pelos iteradores first e last.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator1, class InputIterator2> bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2); template <class InputIterator1, class InputIterator2, class BinaryPredicate>`

	`bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, BinaryPredicate binary_pred);`

find	
Descrição	Procura por um determinado valor no contêiner e retorna sua primeira ocorrência. Se não encontrar, retorna o iterator last.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator, class T> InputIterator find(InputIterator first, InputIterator last, const T& value);`

for_each	
Descrição	Aplica uma função a todos os elementos do contêiner no intervalo entre first e last.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator, class Function> void for_each(InputIterator first, InputIterator last, Function f);`

min_element e max_element	
Descrição	Retornam um iterador que aponta para o mínimo e o máximo elemento do contêiner, respectivamente.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator> ForwardIterator min_element(ForwardIterator first, ForwardIterator last);` `template <class ForwardIterator, class Compare> InputIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);` `template <class ForwardIterator> ForwardIterator max_element(ForwardIterator first, ForwardIterator last);` `template <class ForwardIterator, class Compare> ForwardIterator max_element(ForwardIterator first, ForwardIterator last, Compare comp);`

random_shuffle	
Descrição	Embaralha aleatoriamente os elementos dentro do intervalo de itaradores first e last, com distribuição uniforme. Pode receber como argumento um objeto-função que gere números aleatórios para alterar a distribuição.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class RandomAccessIterator> void random_shuffle (RandomAccessIterator first, RandomAccessIterator last);` `template <class RandomAccessIterator, class RandomNumberGenerator> void random_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& rand);`

remove	
Descrição	Remove elementos de um contêiner que satisfazem a condição elemento == valor.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator, class T>` `ForwardIterator remove (ForwardIterator first,` `ForwardIterator last, const T& value);`

replace	
Descrição	Substitui elementos de um contêiner que são iguais a `old_value` por `new_value`.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator, class T>` `void replace (ForwardIterator first,` `ForwardIterator last, const T& old_value,` `const T& new_value);`

reverse	
Descrição	Inverte a ordem dos elementos do contêiner que estão no intervalo (first, last).
Arquivo de cabeçalho	`algorithm`
Interface	`template <class BidirectionalIterator>` `void reverse (BidirectionalIterator first,` `BidirectionalIterator last);`

rotate	
Descrição	Rotaciona os elementos do segmento que vai de `first` até `middle-1` com os elementos do segmento de `middle` até `last`. O algoritmo rotate_copy retorna uma cópia do contêiner rotacionado.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator>` `void rotate (ForwardIterator first,` `ForwardIterator middle,` `ForwardIterator last);` `template <class ForwardIterator, class OutputIterator>` `OutputIterator rotate_copy (ForwardIterator first,` `ForwardIterator middle,` `ForwardIterator last,` `OutputIterator result);`

search	
Descrição	**`search`** procura no intervalo (`first1`, `last1`) por uma seqüência igual à do intervalo (`first2`, `last2`), enquanto **`search_n`** retorna um iterador para a subseqüência de `count` elementos que são iguais a `value`. Podemos também especificar um predicado para que seja testado na busca.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator1,` `class ForwardIterator2>`

	``ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2); template <class ForwardIterator1, class ForwardIterator2, class BinaryPredicate> ForwardIterator1 search (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate binary_pred); template <class ForwardIterator, class Size, class T> ForwardIterator search_n (ForwardIterator first, ForwardIterator last, Size count, const T& value); template <class ForwardIterator, class Size, class T, class BinaryPredicate> ForwardIterator search_n (ForwardIterator first, ForwardIterator last, Size count, const T& value, BinaryPredicate pred)```

sort	
Descrição	Ordena os elementos de um contêiner. Para comparação entre os elementos do contêiner, pode ser utilizado o operador menor que, <, ou um objeto-função `compare`.
Arquivo de cabeçalho	`algorithm`
Interface	```template <class RandomAccessIterator> void sort (RandomAccessIterator first, RandomAccessIterator last); template <class RandomAccessIterator, class Compare> void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);```

swap e swap_ranges	
Descrição	**`swap`** permuta o conteúdo de dois contêineres, enquanto **`swap_ranges`** permuta os elementos do intervalo (`first1`, `last1`) com os elementos de um intervalo do mesmo tamanho que inicia em `first2`.
Arquivo de cabeçalho	`algorithm`
Interface	```template <class T> void swap (T& a, T& b); template <class ForwardIterator1, class ForwardIterator2> ForwardIterator2 swap_ranges (ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);```

transform	
Descrição	Aplica uma função a um intervalo de valores de um contêiner.

Arquivo de cabeçalho	`algorithm`
Interface	`template <class InputIterator, class OutputIterator, class UnaryOperation> OutputIterator transform (InputIterator first, InputIterator last, OutputIterator result, UnaryOperation op);` `template <class InputIterator1, class InputIterator2, class OutputIterator, class BinaryOperation> OutputIterator transform (InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, OutputIterator result, BinaryOperation binary_op);`

unique e unique_copy	
Descrição	Unique apaga os valores duplicados consecutivos em um contêiner. Unique_copy copia o primeiro elemento de cada grupo de elementos iguais consecutivos para result. Podemos utilizar um predicado para determinar a seleção dos objetos.
Arquivo de cabeçalho	`algorithm`
Interface	`template <class ForwardIterator> ForwardIterator unique (ForwardIterator first, ForwardIterator last);` `template <class ForwardIterator, class BinaryPredicate> ForwardIterator unique (ForwardIterator first, ForwardIterator last, BinaryPredicate binary_pred);` `template <class InputIterator, class OutputIterator> OutputIterator unique_copy (InputIterator first, InputIterator last, OutputIterator result);` `template <class InputIterator, class OutputIterator, class BinaryPredicate> OutputIterator unique_copy (InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate binary_pred);`

Objetos-função

Alguns algoritmos da STL recebem classes especiais, que contém funções para serem utilizadas com os algoritmos da STL. São chamados de objetos-função, que podem possuir funções normais e ponteiros para funções e objetos, sobrecarregando o operador parêntesis, bastando para isto, definir um método **`operator`**`()()`.
Objetos-função podem ser divididos nos seguintes grupos:

- Geradores: objetos-função que não recebem argumentos.
- Funções unárias: objetos-função que recebem um argumento.
- Funções binárias: objetos-função que recebem dois argumentos.
- Predicados: funções unárias que retornam um valor boleano.
- Predicados binários: funções binárias que retornam um valor boleano.

Existem vários objetos-função predefinidos na STL de C++. Para utilizá-los, deveremos incluir o arquivo de cabeçalho `functional`. Todas as operações matemáticas e lógicas simples possuem objetos-função predefinidos, observe a tabela abaixo:

Operador	Objeto-função correspondente
+	plus
-	minus
*	multiplies
/	divides
%	modulus
- (unário)	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

Com isto encerramos a nossa visão geral da STL. Estamos prontos para utilizar todos os recursos desta poderosa ferramenta do C++.