Estudar Na Prática

Voltar

Anterior Próximo

Anotações da Aula 1

Nesta aula, vimos sobre os seguintes tópicos:

- C
- CS50 IDE
- Compilação
- Funções e argumentos
- Função principal(*main*) e arquivos de cabeçalho
- Ferramentas
- Comandos
- Tipos e Códigos de Formato
- Operadores, limitações, truncamento
- Variáveis e Açúcar Sintático(boas práticas)
- Condicionais
- Expressões booleanas, loops
- Abstração
- Mario
- Memória, imprecisão e overflow

Quero compartilhar meu aprendizado e/ou minha dúvida...

Ir para o Fórum Ir para o Discord

Recomendamos que você leia as anotações da aula, isso pode te ajudar!

C

Hoje vamos aprender uma nova linguagem, **C** : uma linguagem de programação que tem todos os recursos do Scratch e muito mais, porém talvez um pouco menos amigável, por ser puramente em texto:

```
#include <stdio.h>
int main(void)
{
    printf("olá, mundo");
}
```

Embora a princípio tentar absorver todos esses novos conceitos possa parecer como beber de uma mangueira de incêndio - pegando emprestado uma frase do MIT - , tenha certeza de que, no final do semestre, estaremos capacitados e experientes em aprender e aplicar esses conceitos.

Podemos comparar muitos dos recursos de programação em C aos blocos que já vimos e usamos no Scratch. Os detalhes da sintaxe são muito menos importantes do que as ideias, às quais já fomos apresentados.

Em nosso exemplo, embora as palavras sejam novas, as ideias são exatamente as mesmas que os blocos "quando a bandeira verde for clicada" e "diga (olá, mundo)" no Scratch:

```
when green flag clicked
say Olá Mundo!
```

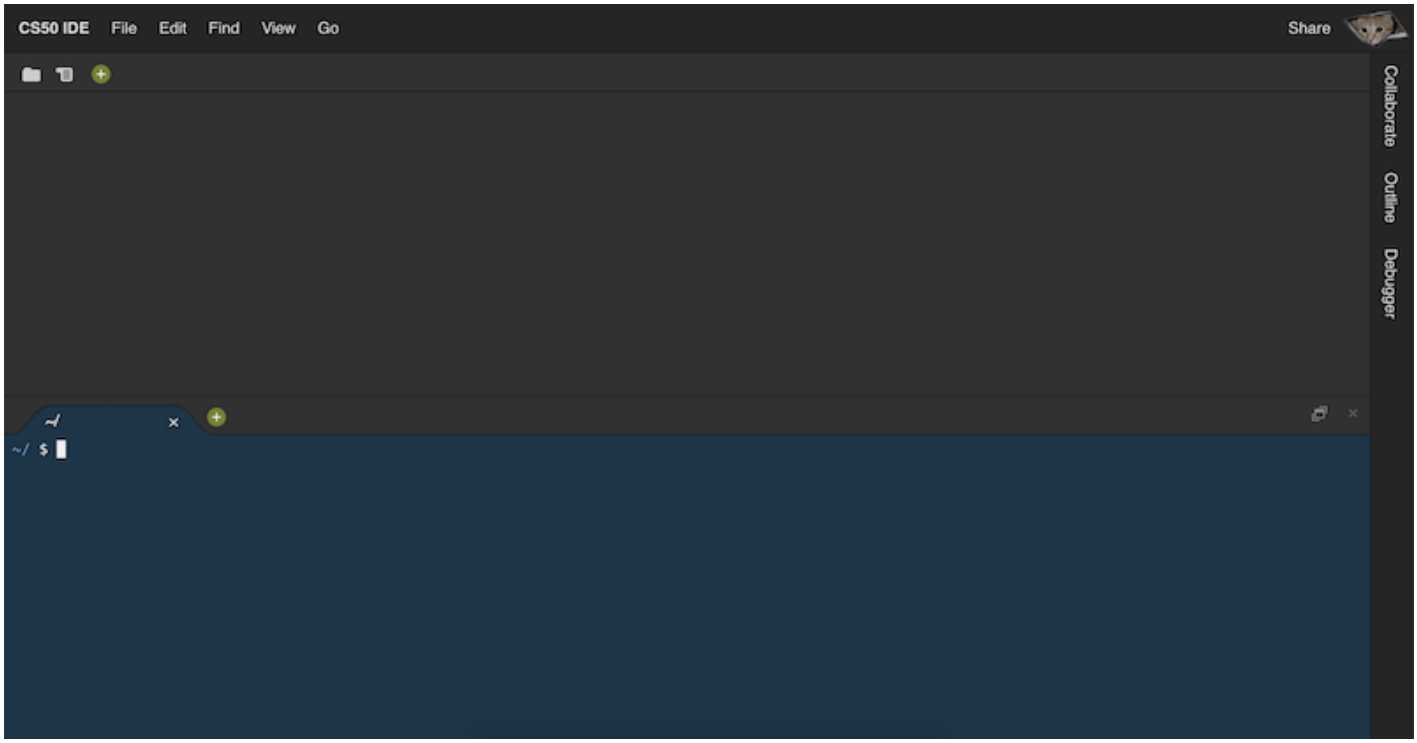
Ao escrever o código, podemos considerar as seguintes qualidades:

- **Correção**, ou se nosso código funciona corretamente, conforme planejado.
- **Design**, ou uma medida subjetiva de quão bem escrito nosso código é, com base em quão eficiente, elegante ou logicamente legível ele é, sem repetição desnecessária.
- **Estilo**, ou o quão esteticamente formatado nosso código é, em termos de indentação consistente e outra colocação de símbolos. As diferenças de estilo não afetam a exatidão ou o significado do nosso código, mas afetam o quão legível é visualmente.

CS50 IDE

Para começar a escrever nosso código rapidamente, usaremos uma ferramenta para o curso, o **CS50 IDE** , um ambiente de desenvolvimento integrado que inclui programas e recursos para escrever código. CS50 IDE é construído sobre um IDE baseado em nuvem muito popular, usado por programadores gerais, mas com recursos educacionais adicionais e personalização.

Abriremos o IDE e, após o login, veremos uma tela como esta:



- O painel superior, em branco, conterá arquivos de texto nos quais podemos escrever nosso código.
- O painel inferior, uma janela de **terminal**, nos permitirá digitar vários comandos e executá-los, incluindo programas do nosso código acima.

Nosso IDE é executado na nuvem e vem com um conjunto padrão de ferramentas, mas saiba que também existem muitos IDEs baseados em desktop, oferecendo mais personalização e controle para diferentes propósitos de programação, ao custo de maior tempo e esforço para configurá-los.

No IDE, iremos para Arquivo> Novo arquivo e, em seguida, Arquivo> Salvar para salvar nosso arquivo como **hello.c** , indicando que nosso arquivo será um código escrito em C. Veremos que o nome de nossa guia de fato mudou para **hello.c** , e agora vamos colar o código que vimos acima:

```
#include <stdio.h>
int main(void)
{
    printf("olá, mundo");
}
```

Para executar nosso programa, usaremos uma **CLI**, ou **interface de linha de comando** , um *prompt* (um "gatilho", por assim dizer) ao qual respondemos inserindo comandos de texto. Isso contrasta com a **interface gráfica do usuário**, ou GUI, como o Scratch, onde temos imagens, ícones e botões além do texto.

Compilação

No terminal no painel inferior de nosso IDE, iremos **compilar** nosso código antes de podermos executá-lo. Os computadores só entendem binário, que também é usado para representar instruções como imprimir algo na tela. Nosso **código-fonte** foi escrito em caracteres que podemos ler, mas precisa ser compilado: convertido em **código de máquina**, padrões de zeros e uns que nosso computador possa entender diretamente.

Um programa chamado **compilador** pegará o código-fonte como entrada e produzirá o código de máquina como saída. No IDE CS50, já temos acesso a um compilador, por meio de um comando chamado **make** . Em nosso terminal, digitaremos make hello, que encontrará automaticamente nosso arquivo hello.c com nosso código-fonte e o compilará em um programa chamado hello. Haverá alguma saída, mas nenhuma mensagem de erro em amarelo ou vermelho, então nosso programa foi compilado com sucesso.

Para executar nosso programa, digitaremos outro comando, ./hello, que procura na pasta atual , . , para um programa chamado hello e o executa.

Funções e argumentos

Usaremos as mesmas ideias que exploramos no Scratch.

Funções são pequenas ações ou verbos que podemos usar em nosso programa para fazer algo, e as entradas para funções são chamadas de **argumentos.**

- Por exemplo, o bloco “say” (“dizer”) no Scratch pode ter considerado algo como “olá, mundo” como um argumento. Em C, a função de imprimir algo na tela é chamada de **printf**(com **f** significando texto “formatado”, que veremos em breve). E em C, passamos os argumentos entre parênteses, como em **`printf ("hello, world")`**; . As aspas duplas indicam que queremos imprimir as letras **hello, world** literalmente, e o ponto-e-vírgula no final indica o fim de nossa linha de código.

As funções também podem ter dois tipos de saídas:

- **efeitos colaterais**, como algo impresso na tela,
- e **valores de retorno**, um valor que é passado de volta ao nosso programa que podemos usar ou armazenar para mais tarde.
    - O bloco “ask” (“perguntar”) no Scratch, por exemplo, criou um bloco “answer” (“responder”).

Para obter a mesma funcionalidade do bloco “ask”, usaremos uma **biblioteca** ou um conjunto de código já escrito. A Biblioteca CS50 incluirá algumas funções básicas e simples que podemos usar imediatamente. Por exemplo, **get_string** pedirá ao usuário uma string, ou alguma sequência de texto, e a retornará ao nosso programa. **get_string** recebe algum input e o usa como prompt para o usuário, como **“Qual é o seu nome?”**, e nós teremos que salvá-lo em uma variável com:

```
string answer = get_string("Qual é o seu nome?");
```

- Em C, o “=” indica **atribuição** ou configuração do valor à direita para a variável à esquerda. E o programa chamará a função get_string primeiro para então obter seu output.
- E também precisamos indicar que nossa variável chamada answer é do **tipo** string, então nosso programa saberá interpretar os zeros e uns como texto.
- Finalmente, precisamos nos lembrar de adicionar um ponto-e-vírgula para encerrar nossa linha de código.

No Scratch, também usamos o bloco “answer” dentro de nossos blocos “join” (“juntar”) e “say”. Em C, faremos isso:

```
printf("olá,% s", resposta);
```

- O %s é chamado de **código de formatação**, o que significa apenas que queremos que a função printf substitua uma variável onde está o marcador %s. E a variável que queremos usar é answer, que passamos para printf como outro argumento, separado do primeiro por uma vírgula. (`printf ("hello, answer")`) iria literalmente imprimir hello, answer sempre.)

De volta ao IDE CS50, nós implementaremos o que descobrimos:

```
#include <cs50.h>
#include <stdio.h>
int main(void)
{
     string answer = get_string("Qual é o seu nome?");
     printf("olá, %s", resposta);
}
```

- Precisamos dizer ao compilador para incluir a Biblioteca CS50, com **`#include <cs50.h>`,** para que possamos usar a função **get_string.**
- Também temos a oportunidade de escrever o código usando um “estilo” que favoreca intuitividade, já que poderíamos nomear nossa variável de resposta com qualquer coisa, mas um nome mais descritivo nos ajudará a entender sua finalidade melhor do que um nome mais curto como a ou x.

Depois de salvar o arquivo, precisaremos recompilar nosso programa com make hello, já que alteramos apenas o código-fonte, mas não o código de máquina compilado. Outras linguagens ou IDEs podem não exigir que recompilemos manualmente nosso código depois de alterá-lo, mas aqui temos a oportunidade de ter mais controle e compreensão do que está acontecendo nos bastidores.

Agora, ./hello executará nosso programa e solicitará nosso nome conforme pretendido. Podemos notar que o próximo prompt é impresso imediatamente após a saída de nosso programa, como em hello, Brian ~ / $. Podemos adicionar uma nova linha após a saída de nosso programa, de modo que o próximo prompt esteja em sua própria linha, com \n:

```
printf("olá, %s\n" ,resposta);
```

\n é um exemplo de **sequência de escape** ou algum texto que na verdade representa algum outro texto.

Função principal(*main*) e arquivos de cabeçalho

O bloco “quando a bandeira verde for clicada” no Scratch inicia o que consideramos ser o programa principal. Em C, a primeira linha para o mesmo é `int main (void)`, sobre a qual aprenderemos mais nas próximas semanas, seguida por uma chave aberta { e uma chave fechada } , envolvendo tudo o que deveria estar em nosso programa.

```
int main(void)
{
```

```
}
```

- Aprenderemos mais sobre como podemos modificar essa linha nas próximas semanas, mas, por enquanto, simplesmente usaremos isso para iniciar nosso programa.

Arquivos de cabeçalho que terminam com .h referem-se a algum outro conjunto de código, como uma biblioteca, que podemos usar em nosso programa. Nós os incluímos com linhas como `#include <stdio.h>` , por exemplo, para a biblioteca de entrada / saída padrão, que contém a função **printf**.

Ferramentas

Com toda a nova sintaxe, é fácil cometer erros ou esquecer algo. Temos algumas ferramentas criadas pela equipe para nos ajudar.

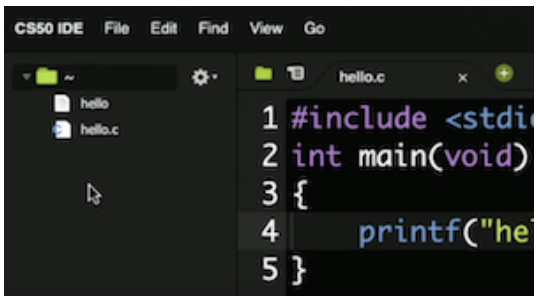
Podemos esquecer de incluir uma linha de código e, quando tentamos compilar nosso programa, vemos muitas linhas de mensagens de erro que são difíceis de entender, pois o compilador pode ter sido projetado para um público mais técnico. **help50** é um comando que podemos executar para explicar problemas em nosso código de uma forma mais amigável. Podemos executá-lo adicionando help50 à frente de um comando que estamos tentando, como help50 make hello , para obter conselhos que possam ser mais compreensíveis.

Acontece que, em C, novas linhas e indentação geralmente não afetam a forma como nosso código é executado. Por exemplo, podemos alterar nossa **função principal(main)** para uma linha, `int main (void) {printf ("hello, world");},` mas é muito mais difícil de ler, então consideramos que tem um estilo ruim. Podemos executar style50 , como style50 hello.c, com o nome do arquivo de nosso código-fonte, para ver sugestões de novas linhas e recuo.

Além disso, podemos adicionar **comentários** , notas em nosso código-fonte para nós mesmos ou para outras pessoas que não afetem a forma como nosso código é executado. Por exemplo, podemos adicionar uma linha como `// Cumprimentar o usuário`, com duas barras // para indicar que a linha é um comentário e, em seguida, escrever o propósito do nosso código ou programa para nos ajudar a lembrar mais tarde.

check50 irá verificar a exatidão do nosso código com alguns testes automatizados. A equipe escreve testes especificamente para alguns dos programas que escreveremos no curso, e as instruções para usar o check50 serão incluídas em cada conjunto de problemas ou laboratório, conforme necessário. Depois de executar check50, veremos algum output nos informando se nosso código passou nos testes relevantes.

O IDE CS50 também nos dá o equivalente a nosso próprio computador na nuvem, em algum lugar da internet, com nossos próprios arquivos e pastas. Se clicarmos no ícone da pasta no canto superior esquerdo, veremos uma árvore de arquivos, uma GUI dos arquivos em nosso IDE:



- Para abrir um arquivo, podemos apenas clicar duas vezes nele. hello.c é o código-fonte que acabamos de escrever, e hello em si terá muitos pontos vermelhos, cada um dos quais são caracteres não imprimíveis, pois representam instruções binárias para nossos computadores.

Comandos

Como o IDE CS50 é um computador virtual na nuvem, também podemos executar comandos disponíveis no Linux, um sistema operacional como o macOS ou Windows.

No terminal, podemos digitar **ls**, abreviação de list, para ver uma lista de arquivos e pastas na pasta atual:

```
~ / $ ls
ola* ola.c
```

- **ola** está em verde com um asterisco para indicar que podemos executá-lo como um programa.

Também podemos remover arquivos com **rm** , com um comando como rm ola. Isso nos solicitará uma confirmação e podemos responder com **y** ou **n** para sim ou não.

Com **mv** , ou **move** , podemos renomear arquivos. Com mv hello.c goodbye.c , renomeamos nosso arquivo ola.c com o nome goodbye.c.

Com **mkdir** , ou *diretório make*, podemos criar pastas ou diretórios. Se executarmos mkdir lecture, veremos uma pasta chamada lecture e podemos mover arquivos para diretórios com um comando como mv ola.c lecture/.

Para *mudar os diretórios* em nosso terminal, podemos usar **cd** , como em cd lecture /. Nosso prompt mudará de ~/ para ~/ lecture /, indicando que estamos no diretório de palestras(lecture) dentro de ~ . ~ representa nosso diretório inicial ou a pasta padrão de nível superior de nossa conta.

Também podemos usar .. como uma abreviação para a “pasta-mae”, ou a pasta que contém aquela na qual estamos . Dentro de ~ / lecture / , podemos executar mv ola.c .. para movê-lo de volta para ~ , já que é a pasta-mae de lecture / . cd .. , da mesma forma, mudará o diretório do nosso terminal para a mae atual. Um único ponto ,. , refere-se ao diretório atual, como em ./ola.

Agora que nossa pasta lecture/ está vazia, podemos removê-la com rmdir lecture/ também.

Tipos e Códigos de Formato

Existem muitos **tipos** de dados que podemos usar para nossas variáveis, que indicam ao computador que tipo de dados eles representam:

- bool , uma expressão booleana **verdadeira** ou **falsa**
- char , um único caractere ASCII como **a** ou **2**
- double , um valor de vírgula flutuante com mais dígitos do que um **float**
- float , um valor de vírgula flutuante ou número real com um valor decimal
- int , inteiros até um certo tamanho ou número de bits
- long , inteiros com mais bits, para que possam contar mais do que um **int**
- string , uma linha de caracteres

E a biblioteca CS50 tem funções correspondentes para obter entrada de vários tipos:

- get_char
- get_double
- get_float
- get_int
- get_long
- get_string

Para **printf**, também, existem diferentes marcadores de posição para cada tipo:

- %c para caracteres
- %f para flutuadores, duplos
- %i para ints
- %li para longos
- %s para strings

Operadores, limitações, truncamento

Existem vários operadores matemáticos que podemos usar também:

- + para adição
- - para subtração
- * para multiplicação
- / para divisão
- % para calcular o resto

Faremos um novo programa, **additional.c**:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
     int x = get_int("x: ");
```

```
    int y = get_int("y: ");

    printf("%i\n", x + y);
}
```

- Vamos incluir arquivos de cabeçalho para as bibliotecas que sabemos que iremos usar, e então vamos chamar get_int para obter inteiros do usuário, armazenando-os em variáveis nomeadas **x** e **y**.
- Em seguida, em **printf**, imprimiremos um espaço reservado para um inteiro, **%i** , seguido por uma nova linha. Ja que nós queremos imprimir a soma de **x** e **y**, vamos passar em **x + y** para **printf** para substituir na string.
- Vamos salvar, executar make add no terminal e depois ./addition para ver nosso programa funcionando. Se digitarmos algo que não seja um inteiro, veremos get_int nos pedindo um inteiro novamente. Se digitarmos um número muito grande, como **4000000000**, get_int nos alertará novamente. Isso ocorre porque, como em muitos sistemas de computador, um **int** no CS50 IDE é de 32 bits, que pode conter apenas cerca de quatro bilhões de valores diferentes. E uma vez que os inteiros podem ser positivos ou negativos, o maior valor positivo para um **int** só pode ser cerca de dois bilhões, com um valor negativo mais baixo de cerca de dois bilhões negativos, para um total de cerca de quatro bilhões de valores totais.

Podemos mudar nosso programa para usar o tipo **long**:

```
#include <cs50.h>
#include <stdio.h>

int main (void)
{
    long x = get_long("x: ");

    long y = get_long("y: ");

    printf("%li\n", x + y);
}
```

- Agora podemos digitar inteiros maiores e ver um resultado correto conforme o esperado.

Sempre que obtivermos um erro durante a compilação, é uma boa ideia rolar para cima para ver o primeiro erro e corrigi-lo primeiro, já que às vezes um erro no início do programa fará com que o resto do programa seja interpretado com erros também.

Vejamos outro exemplo, **truncation.c**:

```
#include <cs50.h>
#include <stdio.h>

int main (void)
{
    // Pega os números do usuário
    int x = get_int("x: ");
    int y = get_int("y: ");

    // Divide x por y
```

```
    float z = x / y;
    printf("%li\n", x + y);
}
```

- Vamos armazenar o resultado de **x** dividido por **y** em **z** , um valor de virgula flutuante ou número real, e imprimi-lo também como um valor flutuante.
- Mas quando compilamos e executamos nosso programa, vemos z impresso como números inteiros como **0,000000** ou **1,000000** . Acontece que, em nosso código, **x / y** é dividido como dois inteiros primeiro , portanto, o resultado fornecido pela operação de divisão também é um inteiro. O resultado é **truncado** , com o valor após a vírgula perdida. Mesmo que **z** seja um **float**, o valor que estamos armazenando nele já é um número inteiro.

Para corrigir isso, vamos fazer o *casting*, ou seja, converter nossos números inteiros para float antes de dividi-los:

```
float z = (float) x / (float) y;
```

O resultado será um float como esperamos e, na verdade, podemos lançar apenas um de **x** ou **y** e obter um float também.

Variáveis e Açúcar Sintático(boas práticas)

No Scratch, tínhamos blocos como “set [counter] to (0)” que definem uma variável para algum valor. Em C, escreveríamos `int contador = 0;` para o mesmo efeito.

Podemos aumentar o valor de uma variável com `contador = contador + 1;` , onde olhamos primeiro para o lado direito, pegando o valor original do contador , adicionando 1 e, em seguida, armazenando-o no lado esquerdo (de volta ao contador, neste caso).

C também suporta **açúcar sintático** ou expressões abreviadas para a mesma funcionalidade. Nesse caso, poderíamos dizer de maneira equivalente `contador += 1;` para adicionar um ao contador antes de armazená-lo novamente. Também poderíamos escrever `contador++;` , e podemos aprender isso (e outros exemplos) examinando a documentação ou outras referências online.

Condições

Podemos traduzir condições, ou blocos “se”, com:

```
if (x < y)
{
    printf (“x é menor que y\n”);
}
```

- Observe que em C, usamos { e } (bem como indentação) para indicar como as linhas de código devem ser aninhadas.

Podemos ter condições “if” e “else”:

```
if (x < y)
{
     printf(“x é menor que y\n”);
}
else
{
    printf(“x não é menor que y\n”);
}
```

E até mesmo “senão se(else if)”:

```
if (x < y)
{
     printf(“x é menor que y\n”);
}
else if (x > y)
{
    printf(“x é maior que y\n”);
}
else if (x == y)
{
    printf(“x é igual a y\n”);
}
```

- Observe que, para comparar dois valores em C, usamos ==, dois sinais de igual.
- E, logicamente, não precisamos de `if (x == y)` na condição final, já que esse é o único caso restante, então podemos apenas dizer o contrário com **else**:

```
if (x < y)
{
     printf(“x é menor que y\n”);
}
else if (x > y)
{
    printf(“x é maior que y\n”);
}
else
{
    printf(“x é igual a y\n”);
}
```

Vamos dar uma olhada em outro exemplo, **conditions.c**:

```
#include <cs50.h>
#include <stdio.h>
```

```
int main(void)
{
     // Usuário entra com o valor de x
     int x = get_int(“x: “);

     // Usuário entra com o valor de y
     int y = get_int(“y: “);

     // Compara x e y
     if (x < y)
     {
         printf(“x é menor que y\n”);
     }
     else if (x > y)
     {
        printf(“x é maior que y\n”);
     }
     else
     {
        printf(“x é igual a y\n”);
     }
}
```

- Nós incluímos as condições que acabamos de ver, juntamente com duas “chamadas”, ou usos, de get_int para obter x e y do usuário.
- Vamos compilar e executar nosso programa para ver se ele realmente funciona conforme o planejado.

Em **concorda.c**, podemos pedir ao usuário para confirmar ou negar algo:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
     // Solicita um caracter para o usuário
     char c = get_char("Você concorda?");

     // Verifica se concordou
     if (c == ‘S’ || c == ‘s’)
     {
         printf(“Concordo.\n”);
     }
     else if (c == ‘N’ || c == ‘n’)
     {
        printf(“Não concordo..\n”);
     }
}
```

- Com **get_char**, podemos obter um único caractere e, como só temos um em nosso programa, parece razoável chamá-lo de **c**.
- Usamos duas barras verticais, ll , para indicar um “ou” lógico (matematico), onde qualquer uma das expressões pode ser verdadeira para que a condição seja seguida. (**&&**, por sua vez, indica um “e” lógico, onde ambas as condições deveriam ser verdadeiras.) E observe que usamos dois sinais de igual, == , para comparar dois valores, bem como aspas simples, **’**, para envolver nossos valores de caracteres únicos.
- Se nenhuma das expressões for verdadeira, nada acontecerá, pois nosso programa não tem um loop.

Expressões booleanas, loops

Podemos traduzir um bloco “para sempre” no Scratch com:

```
while (true)
{
    printf (“Oi mundo!\n”);
}
```

- A palavra-chave **while** (enquanto) requer uma condição, então usamos **true** como a expressão booleana para garantir que nosso loop seja executado para sempre. **while** dirá ao computador para verificar se a expressão é avaliada como **true**(verdadeira) e, em seguida, executar as linhas dentro das chaves. Em seguida, ele repetirá isso até que a expressão não seja mais verdadeira. Nesse caso, **true** sempre será true, então nosso loop é um **loop infinito** ou que será executado para sempre.

Poderíamos fazer algo um certo número de vezes com **while**:

```
int i = 0;
while (i < 50)
{
    printf(“Oi mundo!\n”);
    i++;
}
```

- Criamos uma variável,**i**, e a definimos como 0. Então, enquanto **i** é menor que 50, executamos algumas linhas de código, incluindo uma em que adicionamos 1 a **i** a cada passagem. Dessa forma, nosso loop acabará eventualmente, quando **i** atingir um valor de 50.
- Nesse caso, estamos usando a variável **i** como contador, mas como ela não tem nenhum propósito adicional, podemos simplesmente chamá-la de **i**.

Mesmo que possamos iniciar a contagem em 1, como demonstrado abaixo, por convenção devemos começar em 0:

```
int i = 1;
while (i <= 50)
{
     printf(“Oi mundo!\n”);
     i++;
}
```

Outra solução correta, mas possivelmente menos bem projetada, pode comecar com o contador em 50 e contar para trás:

```
int i = 50;
while (i > 0)
{
     printf(“Oi mundo!\n”);
     i--;
}
```

- Nesse caso, a lógica do nosso loop é mais difícil de raciocinar sem servir a nenhum propósito adicional e pode até mesmo confundir os leitores.

Finalmente, mais comumente, podemos usar a palavra-chave for:

```
int i = 0;
for (int i = 0; i < 50; i++)
{
    printf(“Oi mundo!\n”);
}
```

- Novamente, primeiro criamos uma variável chamada **i** e a definimos como 0. Em seguida, verificamos que i < 50 toda vez que alcançamos o topo do loop, antes de executar qualquer código interno. Se essa expressão for verdadeira, executamos o código interno. Finalmente, depois de executar o código interno, usamos **i++** para adicionar um a i , e o loop se repete.
- O loop do tipo **for** é mais elegante do que o loop do tipo **while** nesse caso, uma vez que tudo relacionado ao loop está na mesma linha, e somente o código que realmente desejamos executar multiplas vezes está dentro do loop.

Observe que para muitas dessas linhas de código, como condições do tipo **if** e loops do tipo **for**, não colocamos um ponto e vírgula no final. É assim que a linguagem C foi projetada, muitos anos atrás, e uma regra geral é que apenas as linhas para ações ou verbos têm ponto e vírgula no final.

Abstração

Podemos escrever um programa que imprime **miau**(meow) três vezes:

```
#include <stdio.h>

int main(void)
{
     printf(“miau.\n”);
     printf(“miau.\n”);
     printf(“mmiau.\n”);
}
```

Poderíamos usar um loop **for**, para não ter que copiar e colar tantas linhas:

```
#include <stdio.h>

int main(void)
{
     for(int i = 0; i < 3; i++)
     {
          printf(“miau.\n”);
          printf(“miau.\n”);
          printf(“miau.\n”);
         }
}
```

Podemos mover a linha **printf** para sua própria função, como nossa própria peça de quebra-cabeça:

```
#include <stdio.h>

void miau(void)
{
    printf(“miau.\n”);
}

int main(void)
{
     for(int i = 0; i < 3; i++)
     {
         miau();
         }
}
```

- Definimos uma função, **miau**, acima de nossa função principal(main).

Mas, convencionalmente, nossa **função principal**(main) deve ser a primeira função em nosso programa, então precisamos de mais algumas linhas:

```
#include <stdio.h>

void miau(void);

int main(void)
{
     for(int i = 0; i < 3; i++)
     {
          miau();
         }
}

void miau(void)
{
```

```
      printf(“miau.\n”);
}
```

- Acontece que precisamos declarar nossa função **miau** primeiro com um **protótipo**, antes de usá-lo em *main* , e realmente defini-lo depois. O compilador lê nosso código-fonte de cima para baixo, então ele precisa saber que o **miau** existirá posteriormente no arquivo.

Podemos até mesmo alterar nossa função de **miau** para obter alguma entrada, **n** e miau **n** vezes:

```
#include <stdio.h>

void miau(int n);

int main(void)
{
     miau(3);
}

void miau(int n)
{
     for(int i = 0; i < 3; i++)
     {
          printf(“miau.\n”);
         }
}
```

- O **void** antes da função **miau** significa que ela não retorna um valor e, da mesma forma, no geral , não podemos fazer nada com o resultado de **miau** , então apenas a chamamos.

A abstração aqui leva a um design melhor, já que agora temos a flexibilidade de reutilizar nossa função **miau** em vários lugares no futuro.

Vejamos outro exemplo de abstração, get_positive_int.c:

```
#include <cs50.h>
#include <stdio.h>

int get_positive_int(void);

int main(void)
{
     int i = get_positive_int();
     printf(“%i\n”);
}

// Solicita um número inteiro positivo ao usuário
int get_positive_int(void)
{
     int n;
     do
```

```
    {
         n = get_int(“Número positivo: \n”);
        }
    while(n < 1);
    return n;
}
```

- Temos nossa própria função que chama **get_int** repetidamente até que tenhamos algum número inteiro que não seja menor que 1. Com um loop do-while, nosso programa fará algo primeiro, depois verificará alguma condição e repetirá enquanto a condição for verdadeira. Um loop while, por outro lado, verificará a condição primeiro.
- Precisamos declarar nosso inteiro **n** fora do loop do-while, pois precisamos usá-lo após o término do loop. O **escopo** de uma variável em C se refere ao contexto, ou linhas de código, dentro do qual ela existe. Em muitos casos, serão as chaves ao redor da variável.
- Observe que a função **get_positive_int** agora começa com **int**, indicando que ela tem um valor de retorno do tipo **int** e, em principal, nós o armazenamos em **i** após chamar **get_positive_int()**. Em **get_positive_int**, temos uma nova palavra-chave, **return** , para retornar o valor **n** para onde quer que a função foi chamada.

Mario

Podemos querer um programa que imprima parte de uma tela de um videogame como Super Mario Bros. Em **mario.c**, podemos imprimir quatro pontos de interrogação, simulando blocos:

```
#include <stdio.h>

int main(void)
{
    printf(“?????\n”);
}
```

Com um loop, podemos imprimir vários pontos de interrogação, seguindo-os com uma única nova linha após o loop:

```
#include <stdio.h>

int main(void)
{
    for(int i = 0; i < 3; i++)
    {
          printf(“?”);
    }
    printf (“\n”);
}
```

Podemos obter um número inteiro positivo do usuário e imprimir esse número de pontos de interrogação, usando **n** para o nosso loop:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
     //  Pega o valor de n com o usuário
    int n;
    do
    {
          n = get_int(“Largura: “);
    }
    while (n < 1);

    // Imprima pontos de interrogação
    for(int i = 0; i < n; i++)
    {
         printf(“?”);
    }
    printf(“\n”);
}
```

E podemos imprimir um conjunto bidimensional de blocos com loops aninhados, um dentro do outro:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    for(int i = 0; i < 3; i++)
    {
            for(int j = 0; j < 3; j++)
        {
                        printf(“#”);
        }
        printf(“\n”);
    }
}
```

- Temos dois loops aninhados, onde o loop externo usa **i** para fazer tudo que contem 3 vezes, e o loop interno usa **j** , uma variável diferente, para fazer algo 3 vezes para cada um desses tempos. Em outras palavras, o loop externo imprime 3 linhas, terminando cada uma delas com uma nova linha, e o loop interno imprime 3 colunas, ou caracteres tipo **#**, *sem* uma nova linha.

Memória, imprecisão e estouro

Nosso computador tem memória, em chips de hardware chamados RAM, memória de acesso aleatório. Nossos programas usam essa RAM para armazenar dados enquanto estão em execução, mas essa memória é finita.

Com **imprecision.c** , podemos ver o que acontece quando usamos valores flutuantes:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    float x = get_float("x: ");
    float y = get_float("y: ");

    printf(“%.50f\n", x / y;);
}
```

- Com **% .50f** , podemos especificar o número de casas decimais exibidas.
- Hmm, agora nós temos ...

```
x: 1
y: 10
0,10000000149011611938476562500000000000000000000000
```

- Acontece que isso é chamado de **imprecisão de vírgula flutuante**, em que não temos bits suficientes para armazenar todos os valores possíveis. Com um número finito de bits para um **float** , não podemos representar todos os números reais possíveis (dos quais existe um número *infinito* de), então o computador tem que armazenar o valor mais próximo que puder. E isso pode levar a problemas em que mesmo pequenas diferenças no valor se somam, a menos que o programador use alguma outra maneira para representar os valores decimais com a precisão necessária.

Na semana passada, quando tínhamos três bits e precisávamos contar mais do que sete (ou 111), adicionamos outro bit para obter oito, 1000 . Mas se tivéssemos apenas três bits disponíveis, não teríamos lugar para o 1 extra . Ele desapareceria e estaríamos de volta a 000. Esse problema é chamado de **overflow (“vazamento”) de inteiro**, pois um inteiro só pode atingir um tamanho especifico antes de ficar sem bits.

O problema Y2K surgiu porque muitos programas armazenavam o ano civil com apenas dois dígitos, como 98 para 1998 e 99 para 1999. Mas quando o ano 2000 se aproximou, os programas tiveram que armazenar apenas 00, levando a confusão entre os anos 1900 e 2000 .

Em 2038, também ficaremos sem bits para rastrear o tempo, já que há muitos anos alguns humanos decidiram usar 32 bits como o número padrão de bits para contar o número de segundos desde 1º de janeiro de 1970. Mas com 32 bits representando apenas números positivos, só podemos contar até cerca de quatro bilhões e, em 2038, atingiremos esse limite, a menos que atualizemos o software em todos os nossos sistemas de computador.

Vamos para a próxima aula?

Anterior Próximo

Em caso de dúvida, envie email para relacionamento@estudar.org.br

Plataforma de ensino por