Introdução à Programação

A comunicação humana é possível por meio de diferentes linguagens. No caso do computador, o mesmo se aplica, pois fazemos uso de uma **linguagem de programação** para elaborar rotinas de computação, ensinando o passo-a-passo para o computador.

> Programar é a arte de escrever sequências de instruções(comandos) para computacionalmente resolver problemas do mundo real por meio de uma linguagem de programação. **Problema > Algoritmo > Código-Fonte > Programa**

Existem diferentes tipos de linguagem de programação. Na sequência, cada um deles será apresentado com maiores detalhes.

- **Linguagens de máquina**: Interpretada diretamente pelo hardware. É a linguagem que o processador reconhece. Compiladores e interpretadores convertem linguagens de mais alto nível para linguagem de máquina.

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD989
```

- **Linguagens assembly**: Pequenas abstrações sobre a linguagem de máquina.

```
;-------------------------------------------------------------------;

START:
;-------------------------------------------------------------------;
;    TEST FOR PRESENCE OF CALCULATOR                                ;
;-------------------------------------------------------------------;
           SUB     AX,AX
           MOV     ES,AX
           SUB     BH,BH
           MOV     BL,INT_NUMBER
           SHL     BX,1
           SHL     BX,1
           MOV     DI,ES:[BX]
           MOV     ES,ES:[BX+2]
           ADD     DI,4
           LEA     SI,TAG
           MOV     CX,TAG_LEN
     REPE  CMPSB
           JE      CALL_CALC
           MOV     BX,SCREEN_HANDLE
           MOV     CX,MESSAGE_LEN
           LEA     DX,MESSAGE
           MOV     AH,40h
           INT     21h
           JMP     SHORT CALC_EXIT
;-------------------------------------------------------------------;
;    CALL CALCULATOR                                ;
;-------------------------------------------------------------------;
CALL_CALC:
           MOV     AL,INT_NUMBER
           MOV     BYTE PTR INT_CODE,AL
           DB      0CDh   ; INT
```

Trecho de código retirado do endereço https://assembly.happycodings.com/code13.html (https://assembly.happycodings.com/code13.html)

- **Linguagens de alto nível**: Independente do tipo de máquina. Aproximam-se da forma de comunicação escrita humana. Possuem alto nível de abstração.

```
#include <iostream>

using namespace std;

int saldo;//variável global

int main(){
    int parcela1, parcela2;//variáveis locais
    parcela1 =10, parcela2 = 12;
    saldo = parcela1 + parcela2;

    {//início de bloco
        cout<< saldo << endl;
        int saldo = 14; //variavel local
        cout<< saldo << endl;
    }//fim de bloco
    cout<< saldo << endl ;
    return 0;
}
```

- **Linguagens de sistema**: Utilizada para atividades de baixo nível que envolvem comunicação e controle de hardware ou rotinas de baixo nível. São exemplos de softwares codificados com tais linguagens os sistemas operacionais, utilitários, drivers, compiladores e *linkers*. A linguagem C++ enquadra-se também nesta categoria.

- **Linguagens de script**: De alto nível, muito utilizadas na administração de conjunto de operações corriqueiras. Um exemplo desta categoria de linguagens é o *Shell Script*. Saiba mais sobre linguagens de script aqui (http://www2.ic.uff.br/~bazilio/cursos/lp/material/LinguagensScript.pdf).

```
#!/bin/bash

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
for f in $*
do
    l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
    echo "$f: $l"
        n=$[ $n + 1 ]
        s=$[ $s + $l ]
done

echo "$n files in total, with $s lines in total"
```

Exemplo de código extraído de https://www.macs.hw.ac.uk/~hwloidl/Courses/LinuxIntro/x961.html (https://www.macs.hw.ac.uk/~hwloidl/Courses/LinuxIntro/x961.html)

- **Linguagens de domínio específico**: De alto nível, aplicada a um contexto específico, com expressividade limitada. Expressões regulares são exemplos desta categoria. Saiba mais sobre DSL (http://www.dcc.ufrj.br/~fabiom/dsl/Aula02.pdf).

- **Linguagens visuais**: toda linguagem que não é baseada em texto. Exemplos deste tipo de linguagem são o Scratch (https://scratch.mit.edu/) e o Blockly (https://developers.google.com/blockly/)

- **Linguagens esotéricas**: Linguagens que não tem propósito de uso, especificamente. Neste link (https://exame.abril.com.br/tecnologia/conheca-10-linguagens-bizarras-nas-quais-voce-nunca-vai-ter-que-programar/) você encontrará algumas delas.

Saiba mais clicando aqui (http://cs.lmu.edu/~ray/notes/pltypes/)

Paradigmas de Programação

Paradigmas são formas de fazer alguma coisa. Em linguagens de programação, diferentes paradigmas podem ser utilizados para expressar ideias em código e as linguagens em si oferecem suporte a uma ou mais destas formas de expressão.

Embora hajam diversos paradigmas, comumente quatro deles são amplamente utilizados na indústria e academia.

- **Imperativo**: Neste paradigma, o programador descreve todos os passos necessários para informar como a computação deve ser feita. O controle de fluxo é explícito e cada passo modifica o estado global do programa (conjunto de todos os valores armazenados nas variáveis de um programa em um dado momento do tempo). Este paradigma foi utilizado nas primeiras linguagens de alto nível.

- **Estruturado**: Superconjunto do paradigma imperativo que faz uso de laços, subrotinas e escopos de variável para controlar o fluxo de execução do programa. Linguagens com C, Pascal e ADA se enquadram nesta categoria.

- **Orientado a Objetos**: Aqui o programa é composto por diversos objetos, criado a partir de suas classes. Objetos trocam mensagens entre si por meio da invocação de métodos. Cada objeto tem seu estado próprio e expõe aos demais apenas as operações desejadas. **Este é o paradigma mais utilizado atualmente na indústria**. São exemplo de linguagem com suporte a este paradigma o C++, Java, C#, PHP, Ruby e Python.

- **Declarativo**: O controle do fluxo é implícito. O programador informa como o resultado deve ser e não a lógica para obtê-lo. **SQL** é uma linguagem deste paradigma.

- **Funcional**: O controle de fluxo do programa é realizado por meio da combinação de funções. Utiliza conceitos como imutabilidade, concorrência, recursão, entre outros. **Este paradigma tem se popularizado em diversos segmentos da indústria, especialmente em fintechs e empresas de telecomunicações**. A Nubank (https://www.nubank.com.br/), por exemplo, utiliza a linguagem *Closure* (https://clojure.org/) em seus sistemas. Muitos sistemas de telecomunicação foram codificados em (*Erlang*)[https://www.erlang.org/ (https://www.erlang.org/)]. Temos também uma linguagem deste paradigma chamada Elixir (https://elixir-lang.org/), criada pelo brasileiro José Valim, que tem sido amplamente utilizada em diversas empresas.

Saiba mais clicando aqui (http://cs.lmu.edu/~ray/notes/paradigms/)

Ambiente de Desenvolvimento

Para codificar softwares em C++ é preciso minimamente do compilador (https://www.youtube.com/watch?v=_C5AHaS1mOA) e de um editor de textos. A instalação do compilador apresenta diferenças de acordo com a plataforma. Já o editor de texto pode ser qualquer um dos disponíveis.

Quando falamos de codificação profissional de software, os editores de texto são substituídos por IDEs (*Integrated Development Environment*) com a finalidade de aumentar a produtividade e a qualidade do código produzido.

Na sequência apresentamos a configuração do ambiente nas plataformas Linux (Debian based) e Windows.

Plataforma Linux

Normalmente as distribuições Linux já possuem o compilador **g++** instalado. Para verificar se o compilador está corretamente instalado, podemos abrir o terminal e executar o comando abaixo, que solicita a versão do mesmo.

```
g++ --version
```

Uma vez instalado, esperamos como saída algo semelhante a

```
g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO warranty; not even for
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Caso o compilador não esteja instalado no sistema, será preciso utilizar a ferramenta de gestão de pacotes da distribuição para instalação. Em distruições baseadas no Debian (Ubuntu, por exemplo), o comando *apt-get install* realiza a instalação do compilador via terminal. A checagem da instalação pode ser realizada utilizando o comando *g++ --version*, conforme demonstrando anteriormente.

```
sudo apt-get install g++
```

Plataforma Windows

Assim como na plataforma Linux, no Windows temos diferentes opções de editores e IDEs para desenvolvimento em C++. Uma destas soluções é o DevC++ (https://sourceforge.net/projects/orwelldevcpp/files/latest/download), uma ferramenta gratuita e com bons recursos para quem está iniciando na linguagem. Outras opções também podem ser consideradas, como Code::Blocks, Netbeans, Eclipse, Clion e VSCode. Basicamente, a diferença entre uma IDE e outra refere-se ao conjunto de ferramentas e recursos oferecidos pelas mesmas.

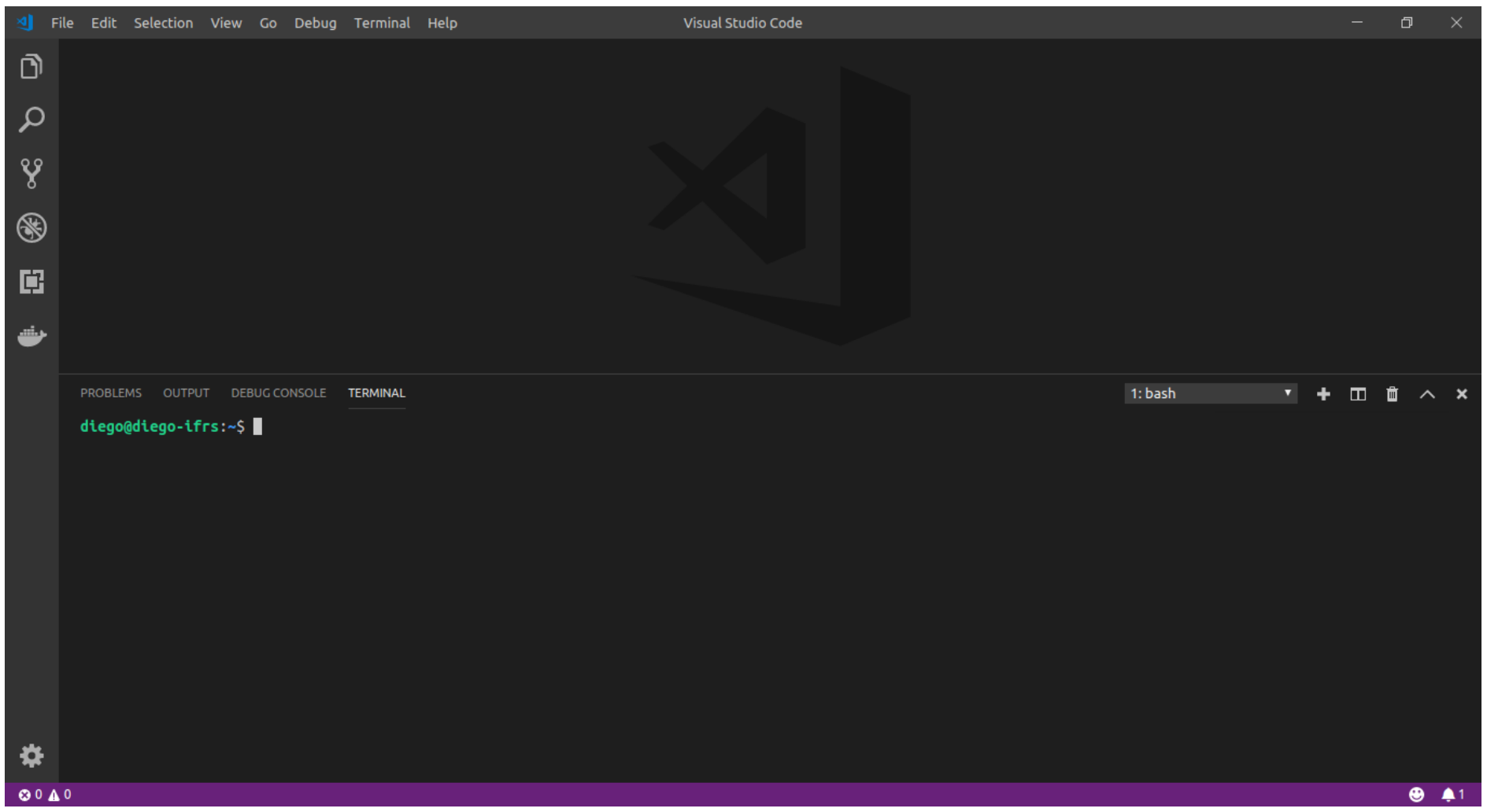
Interessante observar que boa parte destas ferramentas vem com o compilador integrado, o que facilita o processo de instalação. Dos listados abaixo, a exceção fica para o Visual Studio Code.

1. DevC++ (https://sourceforge.net/projects/orwelldevcpp/files/latest/download)
2. **Visual Studio** (https://visualstudio.microsoft.com/)
3. Visual Studio Code (https://code.visualstudio.com/)
4. NetBeans (https://netbeans.org/downloads/index.html)
5. Eclipse (https://www.eclipse.org/downloads/packages/)
6. CLion (https://www.jetbrains.com/clion/)
7. CodeBlocks (http://www.codeblocks.org/)

Para nossas atividades, utilizaremos o editor de códigos **Visual Studio Code** (https://code.visualstudio.com/). O processo de instalação do editor é simples, compreendendo o download do arquivo de pacote (.deb ou .rpm) ou instalador (conforme sistema operacional escolhido) e sua posterior instalação.

O VsCode, como também é conhecido, é um editor avançado com suporte para inúmeras linguagens e tecnologias. Apresenta ainda interface intuitiva, baixo consumo de memória e terminal integrado. Não podemos deixar de comentar também que a ferramenta é gratuita, estando o usuário livre de qualquer cobrança/restrição em relação ao seu uso.

A tela inicial do software contém à esquerda o *explorer*, no qual apresentam-se os arquivos e diretórios do projeto, ferramenta de *busca*, integração com ferramenta de *versionamento de código*, *debug*, *extensões* e outros recursos, dependendo das extensões instaladas. A parte central da tela é ocupada pelos diferentes arquivos em edição e na parte inferior consta o *terminal integrado*.



Estrutura de um programa C++

C++ é uma linguagem de programação de propósito geral que possui recursos de abstração de dados, programação orientada a objetos e programação genérica.

Todo programa C++ é escrito por meio de diversas instruções. As instruções podem apresentar diferentes tipos, dentre eles:

- **Declarações**: São utilizadas para declarar variáveis e constantes em um programa.

```
float capital, montante;
string nome;
bool ativo;
```

- **Atribuições**: Permitem modificar valores de variáveis.

```
capital = 4000.00;
montante = capital * 0.02;
ativo = false;
```

- **Diretivas de pré-processamento**: Instruções adicionadas ao código para uso pelo pré-processador de código.

```
#include <iomanip>
#define MAXIMO 4
```

- **Comentários**: Informações adicionadas ao código que não possuem significado para o processo de compilação. Os comentários são úteis aos programadores que manipulam o código para torná-lo mais compreensível.

```
//Comentário de uma única linha
/*Comentário
de
várias
linhas
*/
```

- **Declaração de funções**: Construções utilizadas para especificar novas funções no código. Funções são rotinas que executam uma operação específica que é necessária em vários locais do programa.

```
double calcular_juro_composto(float capital, float taxa, float tempo){
    return capital * pow(1+taxa,tempo);
}
```

- **Instruções**: Comandos para executar operações e instruções.

```
std::cout << calcular_juro_composto(200.45,0.03,12);
```

Escrevendo o clássico "Hello World"

A estrutura mínima de um programa contém os seguintes elementos:

```
#include <iostream>

int main(){
    std::cout << "Olá mundo";
    return 0;
}
```

Desmembrando a estrutura do programa acima em termos de instruções, considerando o **fluxo de leitura como sendo de cima para baixo (fluxo de execução do programa)**, temos inicialmente a diretiva de pré-processamento `#include <iostream>`

In [2]:
```
#include <iostream>
```

Diretivas de pré-processamento instruem o compilador a realizar determinada ação **antes de iniciar o processo de compilação**. Existem diferentes diretivas, sendo uma delas `include`. A diretiva `include` informa ao compilador que a biblioteca informada (no exemplo `iostream`) deve ser adicionada ao programa para que a compilação seja possível. Perceba que diretivas de pré-processamento sempre iniciam com # e não apresentam ponto-e-vírgula ao final da linha.

Na diretiva `#include <iostream>` estamos importando a biblioteca padrão da linguagem para fluxos de entrada e saída. Esta instrução nos permite a seguir utilizar o fluxo padrão de saída `cout`. Detalhes adicionais sobre a mesma podem ser encontrados no site da documentação oficial (http://www.cplusplus.com/reference/iostream/).

Na sequência temos `int main()`. Essa instrução declara uma função, que é obrigatória para todo e qualquer programa escrito em C++. A função `main` define o ponto de entrada do programa, ou seja, onde ele inicia sua execução.

Desmembrando os elementos da declaração da função `main`, temos:

- **int**: indica que o tipo de retorno da função será um valor inteiro. A idéia de retorno é muito simples: quando desejamos que o chamador função receba um valor ao concluir o processamento da mesma, especificamos qual será o tipo deste valor. No caso da função `main` não podemos alterar. Logo será sempre *int* (valores inteiros).
- **main**: toda função tem um nome específico, afinal é pelo nome que a chamamos. No projeto da linguagem definiu-se o nome de `main` para a função principal.
- **()**: Os parênteses indicam os parâmetros formais da função. Parâmetros são *variáveis* por meio das quais podemos passar valores específicos para a função.

  > Imagine uma função para somar dois valores inteiros. Poderíamos declarar a função como `int soma(int a, int b)`. Perceba neste exemplo que `int a` e `int b` são os parâmetros da função. Quando a função não define parâmetros, basta deixarmos na declaração `()`.

Logo após o "fecha parenteses", temos o símbolo `{`. A partir deste momento estamos definindo o corpo da função (o que ela faz) na forma de um bloco. Uma ou mais instruções podem compor este bloco, sendo que o mesmo finaliza no fechamento de chaves `}` correspondente.

In [3]:
```
std::cout <<"Olá mundo!";
```

```
Olá mundo!
```

A instrução acima imprime o literal (valor fixo) `"Olá mundo!"` para saída no console por meio da função (para fins didáticos, estamos chamando de função) `cout`. O operador << é chamado de *operador de inserção* e nos permite enviar informações para que a função `cout` efetue a saída.

In [4]:
```
return 0;
```

Conforme aprendemos anteriormente, toda função deve retornar algo a quem a chamou (esse algo pode ser *nada*). A função `main` é chamada pelo sistema operacional (SO), pois é o ponto de entrada no programa. Quando escrevemos `return 0;`, estamos dizendo: finaliza a função e devolve o valor inteiro 0 (zero) para o SO.

Mas por que 0 (zero)?

Por convenção. Tornou-se comum retornar 0 para indicar que o programa concluiu com sucesso. Qualquer valor diferente disso seria considerado retorno de erro. Lembre-se que a função `main` retorna ao SO o valor. Logo, este retorno serve a ele ou a outro programa que o tenha executado.

Compilação

Escrever nosso código em um arquivo de código-fonte é somente parte do processo para gerar um software funcional. Nas linguagens compiladas, como é o caso do C++, outros passos são necessários para gerar o programa objeto.

Embora os passos que o compilador execute sejam diversos, vamos simplificar o conceito afirmando que devemos **compilar o código-fonte** para obtermos o programa funcional, que efetivamente pode ser executado pelo sistema operacional.

O compilador é um software que recebe como entrada o código-fonte (um ou mais arquivos). Dentre suas atribuições está a verificação do código com objetivo de identificar erros **sintáticos e semâmticos**.

- **Erros sintáticos**: Compreendem a formulação incorreta de instruções pela falta ou má disposição dos símbolos. É fácil compreender a ideia: quando escrevemos um texto, sabemos que as palavras e símbolos de pontuação devem aparecer em determinada ordem. Ao não respeitar tais regras, cometemos erros de sintaxe. Observe a instrução a seguir:

  ```
  "Olá mundo" << std::cout;
  ```

A instrução acima está **gramaticalmente** incorreta, pois a ordem dos componentes da instrução difere do que o compilador espera. O correto seria escrever:

```
std::cout << "Olá mundo";
```

- **Erros semânticos**: São erros que, por vezes, não podem ser identificados pelo compilador por relacionam-se com o que o programa deveria fazer. Ocorre, portanto, quando você escreve código visando um objetivo, mas na execução do mesmo, o objetivo não é atingido. São exemplos de erros semânticos laços infinitos não previstos (looping), acesso a índices não existentes em *arrays*, laços que executam mais vezes do que o necessário, entre outros.

Como compilamos em linha de comando (Terminal Linux)

No terminal do Linux, utilizamos (para uso básico) o seguinte comando para **compilar** o código-fonte:

```
g++ -std=c++11 arquivo.cpp -o nome-executavel
```

Se compilado com sucesso, a execução do programa deve ser feita adicionando `./` a frente do nome do arquivo executável.

```
./nome-executavel
```

Variáveis, Constantes e Literais

Lembra que comentamos algumas vezes sobre variáveis e constantes? Bem, agora é o momento de as conhecermos de perto.

A execução de um software invariavelmente pressupõe manipulação de dados (número, texto, etc.), afinal é essa a função de um sistema computacional. Mas se precisamos manipular dados, é preciso ter um local para armazená-los enquanto o programa precisa deles, correto?

Correto! Estes locais de armazenamento são chamados **variáveis** e representam espaços, frações da memória RAM (*Random Access Memory*) do computador. Logo, sempre que o programador precise armazenar dados em seu programa, ele **declara** uma ou mais variáveis no código.

Percebeu a ideia? Variáveis são **declaradas**. E declarar significa especificar um **tipo** e um **nome** (mnemônico) para ela. O ***tipo*** *restringe que dados podemos armazenar*, afinal a linguagem é fortemente tipada, ou seja, não podemos armazenar "laranjas" onde aceita-se somente "uvas". Já o ***nome*** *identifica* àquela variável no contexto do programa.

Vejamos alguns exemplos de declarações de variáveis tipos **numéricos inteiros**:

In [1]:
```
short peso;
int quantidade;
long total_pessoas;
```

Tipos inteiros permitem armazenar números **sem informação decimal** (reais). Existem diferentes tipos (e modificadores, mas isso não vem ao caso agora) para que o programador possa escolher o mais adequado em relação a necessidade de representação.

O princípio básico é: quanto maior em termos absolutos o número for, mais bytes são necessários para armazená-lo. Se você pensou então que a diferença entre os tipos está na capacidade de representação (*range* ou intervalo), você acabou de ganhar uma estrelinha :)

Segue tabelinha contendo algumas características dos tipos inteiros.

Tipo	Bits (mínimo)	Representação
short	16 bits (2 bytes)	-32768 a 32767
int	32 bits (4 bytes)	-2147483648 a 2147483647
long	64 bits (8 bytes)	-2,147,483,648 a 2,147,483,647

Vejamos agora declarações de variáveis aptas a armazenar **dados de ponto flutuante** (números reais).

In [ ]:
```
float taxa_juros;
double capital;
```

Novamente nossa tabelinha contendo características dos tipos de dados de ponto flutuante.

Tipo	Bits (mínimo)	Representação
float	32 bits (4 bytes)	+/- $3.4e^{+/-38}$ (~7 digitos de precisão)
double	64 bits (8 bytes)	+/- $1.7e^{+/-308}$ (~15 digitos de precisão)

E, para finalizar, vamos declarar variáveis para armazenar **dados textuais**.

Saiba mais clicando aqui (http://www.cplusplus.com/doc/tutorial/variables/)

In [ ]:
```
std::cout << valor;
```