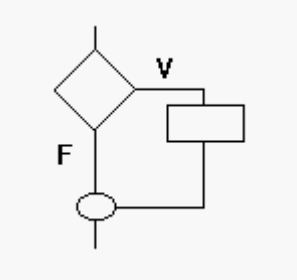
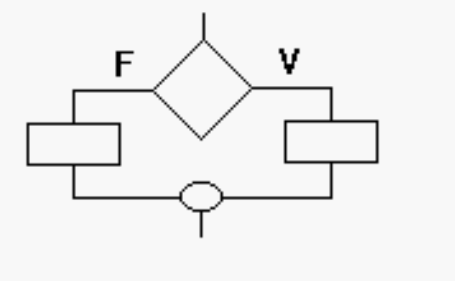
III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Criando funções e protótipos de função
  - Ao declarar ou definir uma função, é necessário descrever os tipos de argumentos ➔ protótipos de função
    - O compilador utiliza os protótipos para garantir que os argumentos apropriados são passados e que o valor de retorno é tratado corretamente
      - Se não está correto, o compilador acusa erro
    - A ordem e o tipo dos argumentos deve coincidir na declaração, na definição e na chamada da função
      - Sintaxes corretas para a declaração
        - tipoRetorno nomeFuncao (tipoArg1 a1, tipoArg2 a2, ...tioArgn an);
        - tipoRetorno nomeFuncao (tipoArg1, tipoArg2, ...tioArgn an);
          - Os tipos são obrigatórios, mas os nomes (a1, a2,...an) são opcionais na declaração
      - Sintaxe correta para a definição
        - tipoRetorno nomeFuncao (tipoArg1 a1, tipoArg2 a2, ...tioArgn an) {
          // aqui no corpo da função utiliza os nomes a1, a2, ... an
          } // o tipo de retorno em C++ é obrigatório (em C, o default é *int*)
          - No C++ pode ter argumento sem nome, desde que não usado na função
            - Possibilita reservar espaço na lista de argumentos para uso futuro
            - Se colocar nome e não usar, vai gerar *warning* na compilação
            - void nomefuncao (void) ; ➔ função não tem argumentos e não tem retorno
            - Usar return () para retornar valor

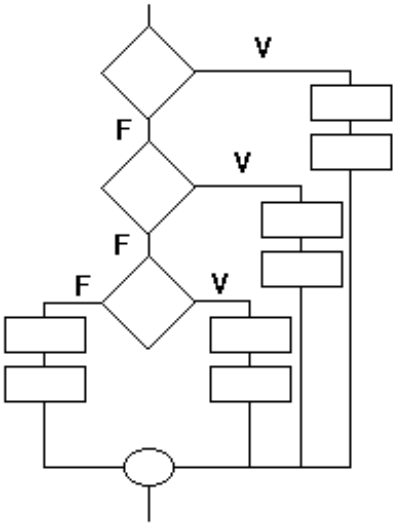
III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Estruturas de controle de execução
  - If-else
    - Formatos
      - if (expression) statement
      - if (expression) statement else statement

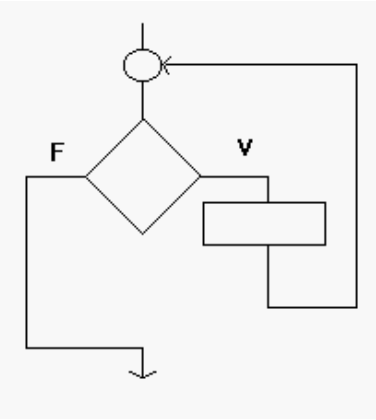




    - Características
      - “expression” resulta em *true* ou *false* (valores Booleanos)
      - “statement” pode ser simples (terminado com ;) ou composto (agrupa comandos contidos entre { e }
      - “statement” também pode conter outros ifs (ifs “aninhados”)



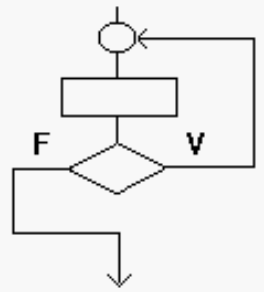
III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C



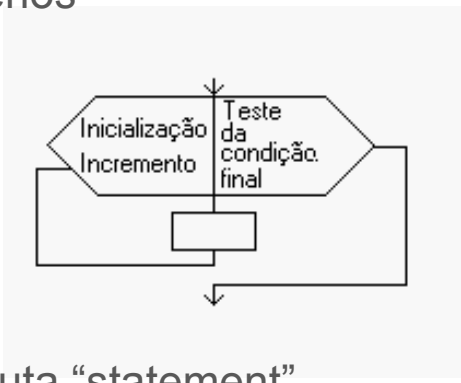
- Estruturas de controle de execução
  - While
    - Formato
      - While (expression) statement
    - Características
      - “statement” repete até que a expressão de controle seja false
      - “expression” é avaliada uma vez no início do laço e novamente após cada iteração de “statement”
      - “expression” não se restringe a um teste simples, podendo ser tão complicada quanto se queira, desde que produza como resultado *true* ou *false*
        - Ex: while (/* do a lot here */) ;
          - Neste caso, o programador escreveu a expressão não só para realizar o teste mas também para executar o trabalho
      - É possível controlar o fluxo utilizando também “break” e “continue”
        - “break”: termina o laço sem executar o restante das etapas
        - “continue”: interrompe a execução da iteração atual e passa para o início do laço da iteração seguinte

III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Estruturas de controle de execução
  - Do-while
    - Formato
      - do statement while(expression);
    - Característica
      - É diferente do while porque “statement” sempre executa pelo menos uma vez, mesmo se “expression” resultar em false na 1a vez



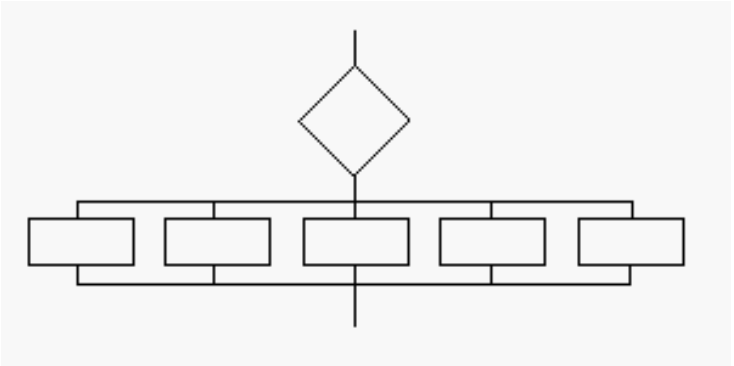
  - Laço For
    - Formato
      - For (initialization; conditional; step) statement
    - Características
      - Realiza a inicialização antes da primeira iteração
      - Realiza o teste condicional antes de cada iteração: se true, executa “statement”
        - Se “conditional” for falso, nunca executa
      - Ao final de cada iteração, incrementa ou decrementa segundo o “step” (passo)
      - Laços for são muito utilizados para tarefas que envolvem contagem
      - Qualquer uma das partes pode estar vazia
        - Se conditional vazia o programa assume que ela é sempre verdade, de modo que o laço só termina com um comando de desvio, como o “break”
          - Nota: ambos (do-while e for) também podem utilizar “break” e “continue”



III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Estruturas de controle de execução
  - switch
    - Formato

```
switch (selector) {
    case integral-value1 : statement; break;
    case integral-value2 : statement; break;
    case integral-value3 : statement; break;
    (...)
    default: statement;
}
```



    - Características
      - Seleciona porções de código baseado em uma expressão (“selector”) que resulta em valor inteiro
      - O switch compara o resultados de “selector” a cada “integral-value”
        - Se coincidir, o “statement” correspondente (simples ou composto) é executado
        - Se não coincidir, o “statement” de “default” é executado
      - “break” é opcional e faz com que a execução “pule” para o final do switch (após “}”)
        - Se não for utilizado, a execução continua e invade os demais “cases”
    - .

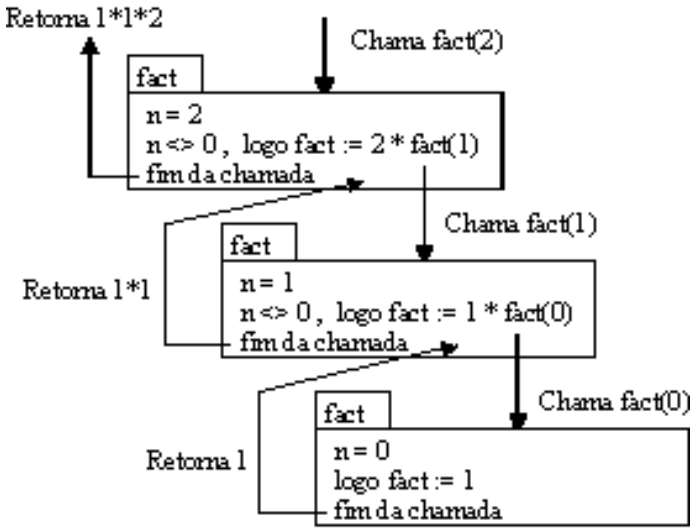
III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Funções recursivas
  - Recursividade
    - Possibilidade de chamar novamente a função na qual se está
    - É uma técnica de programação interessante e em alguns casos útil
    - É necessário ter uma situação de parada para “desempilhar” as chamadas realizadas e não “estourar” a memória
    - Ex: função fatorial

```
#include <iostream>
using namespace std;

unsigned long fatorial (int n) {
   if (n == 0)
      return 1;
   else
      return (n * fatorial(n-1));
}

int main() {
    for (int i = 0; i <= 33; i++)
       cout<<"\nFatorial de "<<i<<" = "<<fatorial(i);
}
```



III.1.2. O C dentro do C++

Revisão de conteúdo – Linguagem C

- Precedência de operadores
  - Notas:
    - Autoincremento e autodecremento
      - ++i: o incremento é realizado e o valor resultante é aplicado
      - i++: o valor atual é aplicado e depois é realizado o incremento
      - Processo semelhante para o decremento
    - Atribuição compacta (ex: `A+=1`)
      - Forma mais concisa e eficiente

```
//: C03:AutoIncrement.cpp
#include <iostream>
using namespace std;

int main() {
  int i = 0;
  int j = 0;
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
}
```

III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Tipos de dados pré-definidos em C e C++
  - **char** : para armazenamento de caracteres
    - Utiliza no mínimo 8 bits (um byte) de armazenamento
  - **int** : armazena número inteiros
    - Utiliza no mínimo 2 bytes de armazenamento
  - **float** e **double** : armazenam números de ponto flutuante
    - Formato padrão IEEE (sinal + expoente + mantissa)
    - float para precisão simples(4 bytes), double para precisão dupla (8 bytes)
  - Nota 1: a especificação do padrão C define apenas valores mínimo e máximo que cada tipo primitivo deve manipular
    - Fica a cargo de cada compilador adequar valores, respeitando o padrão
  - Nota 2: o padrão C++ inclui ainda o tipo bool (valores Booleanos)
    - True: converte para valor inteiro 1
      - Valores inteiros não zero são convertidos implicitamente para true
    - False: converte para valor inteiro 0

III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Especificadores adicionais de tipo
  - Nota: os tamanhos definidos são os menores do padrão

Tipo	Tamanho (bit/byte)	Capacidade de armazenamento de valor
char	8 bits / 1 byte	-127 à +127
signed char	8 bits / 1 byte	-127 `+127 (o mesmo que char)
unsigned char	8 bits / 1 byte	0 à +254
int	16 bits / 2 bytes	-32.767 à +32.767
signed int	16 bits / 2 bytes	-32.767 à +32.767 (o mesmo que int)
unsgned int	16 bits / 2 bytes	0 à +65.535
short int	16 bits / 2 bytes	-32.767 à +32.767 (o mesmo que int)
long int	32 bits / 4 bytes	-2.147.483.647 à +2.147.483.647
signed short int	16 bits / 2 bytes	-32.767 à +32.767 (o mesmo que int)
signed long int	32 bits / 4 bytes	-2.147.483.647 à +2.147.483.647 (o mesmo que long int)
unsigned short int	16 bits / 2 bytes	0 à +65.535 (o mesmo que unsigned int)
unsigned long int	32 bits / 4 bytes	0 à +4.294.967.295

Exercícios

3. Escreva um programa que abra um arquivo e conte o número de espaços em branco do arquivo.
4. O que o seguinte programa faz?

```
#include <iostream>
using namespace std;

int whatIsThis(int b[], int size) {
    if (size == 1)
        return (b[0]);
    else
        return (b[size-1] + whatIsThis(b,size-1));
}
int main() {
    int result, a[10] = {1,2,3,4,5,6,7,8,9,10};
    result = whatIsThis(a,10);
    cout<<"Resultado = "<<result;
}
```

Exercícios

5. Dado um vetor de números inteiros positivos aleatórios entrados via teclado (número negativo indica fim da entrada dos dados), faça um programa utilizando a classe *template* vector para comprimir o vetor suprimindo as repetições de números vizinhos através da contagem do número de repetições de cada um da seguinte forma:

```
Vetor de entrada: 1 1 1 4 1 1 4 4 25 67 67 67 67 2 2
Vetor de saída: 3 1 1 4 2 1 2 4 1 25 4 67 2 2
```

6. Faça um programa que descomprima o vetor de saída do exercício anterior, gerando o vetor de entrada correspondente.

III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Ponteiro
  - Variável que armazena o endereço de outra variável
    - Permite criar e manipular estruturas de dados dinâmicas que podem sofrer alterações (tamanho e forma) ao longo do tempo
      - Variável comum: é um nome para um endereço de memória
      - Ponteiro: armazena um endereço de memória
        - Um ponteiro leva de uma posição de memória a outra da mesma forma que um link leva de um endereço (site) para outro
  - Manipulando ponteiros
    - `int *aptr;`
      - **Declara** ponteiro para inteiro mas não inicializa (poderia atribuir NULL)
        - Um ponteiro deve ser definido para um tipo específico
        - Declaração múltipla: `int *a; int *b; int *c;` (senão apenas o primeiro é ponteiro)
    - `int a = 3; int *aptr; aptr = &a;`
      - **Atribui** ao ponteiro para inteiro o endereço da variável `a` (também inteira, = 3)
        - Um ponteiro só recebe valor de endereço de variável de mesmo tipo
    - `*aptr = 10`
      - Atribui ao endereço apontado pelo ponteiro o valor 10
        - "não modifique o meu valor, mas o da variável apontada por mim"

III.1.2. O C dentro do C++
Revisão de conteúdo – Linguagem C

- Exemplo de manipulação de ponteiros

```
#include <iostream>
using namespace std;

int main() {
    int a, b, *ptr1, *ptr2;
    a = 10;
    b = 11;
    ptr1 = &a;
    ptr2 = &b;
    *ptr2 = 20;
    b = 21;
    *(&a) = 30;
    cout<<"a = "<<a<<" b = "<<b;
}
```

Saída resultante:
a = 30 b = 21

III.1.2. O C dentro do C++

- **Ponteiros:** declaração, atribuição, operador endereço e operador de indireção ou de-referenciação (“conteúdo”)

```
#include <iostream>
using namespace std;
int main() {
   int i = 5;
   int *pti;     // Declara ponteiro para inteiro
   pti = &i;     // Inicializa o ponteiro com o endereco de i
   cout << "Ponteiro:" << pti << " Endereco de i:" << &i << " Valor de i:" << i << endl;
   *pti = 7;     // Atualiza o conteúdo apontado por pti para 7
   cout << "Ponteiro:" << pti << " Endereco de i:" << &i << " Valor de i:" << i << endl;
}
```

Saída resultante:

```
Ponteiro:0x28fee8 Endereco de i:0x28fee8 Valor de i:5
Ponteiro:0x28fee8 Endereco de i:0x28fee8 Valor de i:7
```

III.1.2. O C dentro do C++

- **Ponteiros** e **vetores:** aritmética de ponteiros

```
#include <iostream>
using namespace std;
int main() {
   int  arrint[5] = {10, 2, 33, 42, 51};
   double arrdouble[] = {20, 30, 40};
   int *pta;
   pta = arrint;
   double *ptd = arrdouble;
   for (int i = 0; i < 5; i++)
      cout << *pta++ << endl;
   cout << endl;
   for (int i = 0; i < sizeof(arrdouble)/sizeof(double); i++)
      cout << *ptd++ << endl;

}
```

Saída resultante:

```
10
2
33
42
51

20
30
40
```

III.1.2. O C dentro do C++

- Ponteiros: alocação dinâmica de memória

```
#include <iostream>
using namespace std;
int main() {
   const int tam = 10;       // Define constante de tamanho 10
   int *pti = new int;   // Aloca 1 inteiro, apontado por pti
   *pti = 5;     // Atribui 5 à posição apontada por pti
   double *ptd = new double[tam]; // Aloca "tam" doubles

   for(int i = 0; i < tam; i++)
      ptd[i] = 5.*i; // Trata cada posição do array ptd
   cout << "pti: " << pti << " conteudo: " << *pti << endl; // Impr end e valor pti
   cout << "ptd:" << ptd << endl << "conteudo:" << endl; // Imprime end de ptd
   for(int i = 0; i < tam; i++)
      cout << ptd[i] << endl; // imprime valor de ptd[i]
   delete[] ptd;     // Desaloca o array de doubles apontado por ptd
   delete pti;   // Desaloca o inteiro apontado por pti
}
```

III.1.2. O C dentro do C++

- Ponteiros: aritmética de ponteiros

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  int *ip = a;
  int* ip2;

  for(int i = 0; i < 10; i++)
    a[i] = i; // Give it index values

  cout<<"*ip =  " << *ip << endl;
  cout<<"*++ip = " << *++ip << endl;
  cout<<"*(ip + 5) = " << *(ip + 5) << endl;

  ip2 = ip + 5;

  cout<<"*ip2 = " << *ip2 << endl;
  cout<<"*(ip2 - 4) = " << *(ip2 - 4) << endl;
  cout<<"*--ip2 = " << *--ip2 << endl;
  cout<<"ip2 - ip = " << ip2 - ip << endl;
}
```

saída resultante:

```
*ip = 0
*++ip = 1
*(ip + 5) = 6
*ip2 = 6
*(ip2 - 4) = 2
*--ip2 = 5
ip2 - ip = 4
```

Exercícios

7. Seja `p` uma variável do tipo ponteiro, explique a diferença entre `p++;` `(*p)++;` `*(p++);` `*p++;` `*(p+10);`

8. Seja `a[]` um vetor qualquer, independente de tipo e tamanho, e `pa` um ponteiro para o mesmo tipo de `a[]`. Responda `V` ou `F`, justificando:

   a) Após a atribuição `pa=&a[0];` `pa` e `a` possuem valores idênticos, isto é, apontam para o mesmo endereço
   b) A atribuição `pa=&a[0];` pode ser escrita como `pa=a;`
   c) `a[i]` pode ser escrito como `*(a+i)`
   d) `&a[i]` e `a+i` são idênticos
   e) `a+i` é o endereço do `i`-ésimo elemento após `a`
   f) `pa[i]` é idêntico a `*(pa+i)`
   g) `pa=a` é uma operação válida
   h) `pa++` é uma operação válida
   i) `a=pa` é uma operação válida
   j) `a++` é uma operação válida

Exercícios

9. Após a execução do código abaixo, informe, para cada letra, se a sequência de código está correta e o que será impresso.

```
int mat[4] = {0,10,20,30}, *p, x;
p = mat;
```

a)
```
x = *p++;
cout<<x<<" "<<*p;
```

b)
```
x = (*p)++;
cout<<x<<" "<<*p;
```

c)
```
x = *(p++);
cout<<x<<" "<<*p;
```

d)
```
x = *mat++;
cout<<x<<" "<<*mat;
```

e)
```
x = (*mat)++;
cout<<x<<" "<<*mat;
```

f)
```
x = *(mat++);
cout<<x<<" "<<*mat;
```