

# A linguagem C++

Programação Avançada  
Universidade Federal de Pernambuco  
Adrien Durand-Petiteville  
`adrien.durandpetiteville@ufpe.br`

- Os computadores modernos são incrivelmente rápidos e cada vez mais rápidos.
- No entanto, os computadores também têm algumas restrições significativas: eles entendem apenas nativamente um **conjunto limitado de comandos** e devem ser **informados exatamente o que fazer**.

- Um programa de computador (*Computer program*)(também comumente chamado de aplicativo) é um conjunto de instruções que o computador pode executar para realizar alguma tarefa.
- O processo de criação de um programa é chamado de **programação** (*Programming*).
- Os programadores normalmente criam programas produzindo **código-fonte** (*Source code*), que é uma lista de comandos digitados em um ou mais arquivos de texto.

- A coleção de partes físicas do computador que o compõe e executa programas é chamada de **hardware**.
- Quando um programa de computador é carregado na memória e o hardware executa cada instrução sequencialmente, isso é chamado de **execução** ou **execução do programa**.

- A CPU de um computador é incapaz de falar C, C++ ou Python.
- O conjunto limitado de instruções que uma CPU pode entender diretamente é chamado de código de máquina (ou linguagem de máquina ou conjunto de instruções).
- Aqui está um exemplo de instrução em linguagem de máquina:  
10110000 01100001

# Assembly Language - 1



- Como a linguagem de máquina é tão difícil para os humanos lerem e entenderem, a linguagem assembly foi inventada.
- Em uma linguagem assembly, cada instrução é identificada por uma abreviatura curta (em vez de um conjunto de bits) e nomes e outros números podem ser usados.
- Aqui está a mesma instrução acima em linguagem assembly:  
`mov al, 061h`
- Isso torna o assembly muito mais fácil de ler e escrever do que a linguagem de máquina.
- No entanto, a CPU não consegue entender a linguagem assembly diretamente.
- O programa de montagem deve ser traduzido para a linguagem de máquina antes de ser executado pelo computador.
- Isso é feito usando um programa chamado **assembler**.

- Os programas escritos em linguagens assembly tendem a ser muito rápidos, e o assembly ainda é usado hoje quando a velocidade é crítica.
- No entanto, a assembly ainda tem algumas desvantagens:
  - As linguagens assembly ainda exigem muitas instruções para fazer até tarefas simples.
  - Entender o que um programa inteiro está fazendo pode ser desafiador
  - A linguagem assembly ainda não é muito portátil: um programa escrito em assembly para uma CPU provavelmente não funcionará em hardware que usa um conjunto de instruções diferente e teria que ser reescrito ou extensivamente modificado.

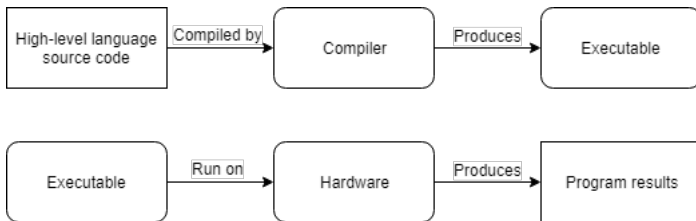
- Para tratar das questões de legibilidade e portabilidade, novas linguagens de programação como C, C ++, Java, Python foram desenvolvidas
- Essas linguagens são chamadas de linguagens de alto nível, pois são projetadas para permitir que o programador escreva programas sem ter que se preocupar com o tipo de computador em que o programa será executado
- Aqui está a mesma instrução acima em C/C++: `a = 97;`
- Programas escritos em linguagens de alto nível devem ser traduzidos para um formato que o computador possa entender antes de serem executados
- Existem duas maneiras principais de fazer isso: **compilando** e **interpretando**



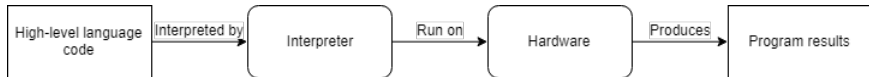
# High-level Languages - 2

	low level <i>Assembler</i>	mid/high level <i>C/C++</i>	high level <i>Java/C#</i>	scripting <i>Python</i>
Development speed				
Performance				
Low-level optimization	++	+	-	-
Meta programming	+	++	-/+	+
Cross-platform support	individual code for each platform	compiled everywhere	compiled once, run everywhere	interpreted on many platforms
Supporting OOP	-	+	+	+
Type of linking	mostly static	static/dynamic	dynamic only	n/a

- Um **compilador** é um programa que lê o código-fonte e produz um programa executável autônomo que pode ser executado.
- Depois que seu código foi transformado em um executável, você não precisa do compilador para executar o programa.



- Um **interpretador** é um programa que executa diretamente as instruções no código-fonte sem exigir que sejam compiladas em um executável primeiro.
- Os interpretadores tendem a ser mais flexíveis do que os compiladores, mas são menos eficientes ao executar programas porque o processo de interpretação precisa ser feito toda vez que o programa é executado.
- Isso significa que o interpretador é necessário sempre que o programa é executado.

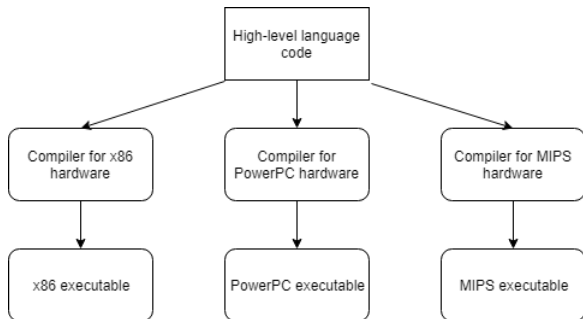


# Avantagens das linguagens de alto nível - 1

- As linguagens de alto nível são muito mais fáceis de ler e escrever porque os comandos estão mais próximos da linguagem natural que usamos todos os dias
- As linguagens de alto nível requerem menos instruções para executar a mesma tarefa que as linguagens de nível inferior, tornando os programas mais concisos e fáceis de entender
- Em C ++, você pode fazer algo como  
`a = b * 2 + 5;` em uma linha  
Em linguagem assembly, isso exigiria 5 ou 6 instruções diferentes. Terceiro, os programas podem ser compilados (ou interpretados) para muitos sistemas diferentes, e você não precisa alterar o programa para rodar em CPUs diferentes (basta recompilar para aquela CPU). Como um exemplo:

## Avantagens das linguagens de alto nível - 2

- Os programas podem ser compilados (ou interpretados) para muitos sistemas diferentes, e você não precisa alterar o programa para rodar em CPUs diferentes (basta recompilar para aquela CPU)



- A linguagem C foi desenvolvida em 1972 por Dennis Ritchie nos laboratórios Bell Telephone, principalmente como uma linguagem de programação de sistemas (uma linguagem para escrever sistemas operacionais).
- Os objetivos principais de Ritchie eram produzir uma linguagem minimalista que
  - fosse fácil de compilar
  - permitisse acesso eficiente à memória
  - produzisse código eficiente
  - fosse independente (não dependente de outros programas)
- Para uma linguagem de alto nível, ela foi projetada para dar ao programador muito controle, enquanto ainda encoraja a independência de plataforma (hardware e sistema operacional) (ou seja, o código não precisava ser reescrito para cada plataforma).

- C++ foi desenvolvido por Bjarne Stroustrup na Bell Labs como uma extensão para C, começando em 1979
- C++ adiciona muitos novos recursos à linguagem C, e talvez seja melhor pensado como um superconjunto de C
- A fama do C++ resulta principalmente do fato de ser uma linguagem orientada a objetos
- Três atualizações importantes para a linguagem C++ (C++11, C++14 e C++17) foram feitas, cada uma adicionando funcionalidade adicional
- O C++11 em particular adicionou um grande número de novos recursos e, neste ponto, é amplamente considerado a nova linha de base

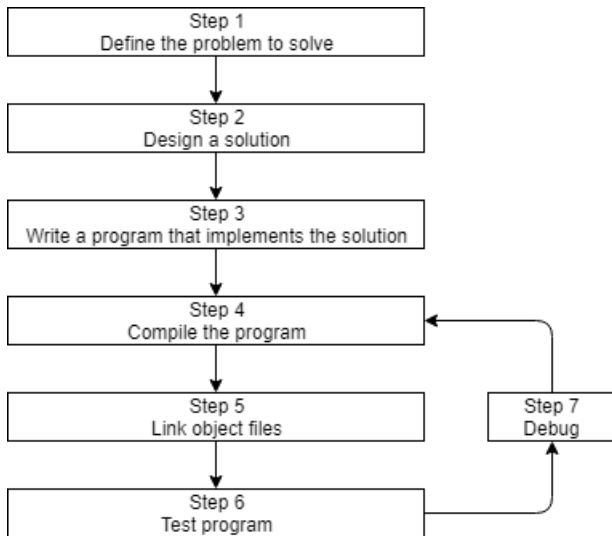
- A filosofia de design subjacente de C e C++ pode ser resumida como **“confiar no programador”** - o que é maravilhoso e perigoso.
- C++ é projetado para permitir ao programador um alto grau de liberdade para fazer o que quiser.
- No entanto, isso também significa que a linguagem muitas vezes não o impedirá de fazer coisas que não fazem sentido, porque presumirá que você está fazendo isso por algum motivo que não entende.
- Existem algumas armadilhas nas quais os novos programadores provavelmente cairão se forem pegos de surpresa.
- **Esta é uma das principais razões pelas quais saber o que você não deve fazer em C / C++ é quase tão importante quanto saber o que você deve fazer.**



# No que C++ é bom?

- C++ se destaca em situações em que é necessário alto desempenho e controle preciso sobre a memória e outros recursos
- Aqui estão alguns tipos comuns de aplicativos que provavelmente seriam escritos em C++:
  - Jogos de vídeo
  - Sistemas em tempo real (por exemplo, para transporte, fabricação, etc ...)
  - Aplicativos financeiros de alto desempenho (por exemplo, negociação de alta frequência)
  - Aplicações gráficas e simulações
  - Produtividade / aplicativos de escritório
  - Software embarcado
  - Processamento de áudio e vídeo

# Introdução ao desenvolvimento C++



# Etapa 1: Definir o problema que você gostaria de resolver

- Esta é a etapa “o quê”, onde você descobre qual problema pretende resolver
  - “Quero escrever um programa que me permita inserir muitos números e, em seguida, calcular a média.”
  - “Quero escrever um programa que gere um labirinto 2D e permita ao usuário navegar por ele. O usuário ganha se chegar ao fim. ”

## Etapa 2: Determinar como você vai resolver o problema

- Esta é a etapa “como”, onde você determina como resolverá o problema que surgiu na etapa 1
- É também a etapa mais negligenciada no desenvolvimento de software
- O cerne da questão é que existem muitas maneiras de resolver um problema - no entanto, algumas dessas soluções são boas e outras são ruins
- Normalmente, boas soluções têm as seguintes características:
  - Diretas: não excessivamente complicados ou confusos
  - Bem documentadas
  - Construídas de maneira modular, para que as peças possam ser reutilizadas ou alteradas posteriormente sem impactar outras partes do programa
  - Robustas e podem recuperar ou fornecer mensagens de erro úteis quando algo inesperado acontece

## Etapa 2: Determinar como você vai resolver o problema - 2

- Estudos mostraram que apenas 20% do tempo de um programador é realmente gasto escrevendo o programa inicial
- Os outros 80% são gastos em manutenção, que pode consistir em:
  - depuração (remoção de bugs)
  - atualizações para lidar com as mudanças no ambiente (por exemplo, para rodar em uma nova versão do sistema operacional)
  - melhorias (pequenas mudanças para melhorar a usabilidade ou capacidade)
  - melhorias internas (para aumentar a confiabilidade ou facilidade de manutenção).
- Consequentemente, vale a pena gastar um pouco de tempo extra antecipadamente (antes de começar a codificar) pensando sobre a melhor maneira de resolver um problema

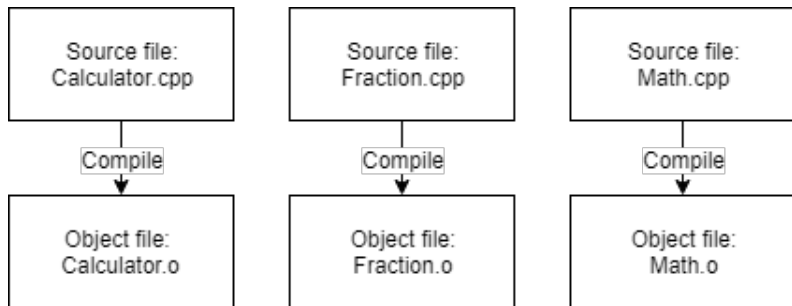
## Etapa 3: Escrever o programa

- Precisamos de um editor de texto para escrever e salvar nossos programas escritos.
- Os programas que escrevemos usando instruções C++ são chamados de código-fonte
- É possível escrever um programa usando qualquer editor de texto que você quiser
- Recomendamos fortemente que você use um editor projetado para programação
  - 1) Numeração de linha. A numeração de linha é útil quando o compilador fornece um erro, como um erro típico do compilador indica: algum código / mensagem de erro, linha 64
  - 2) Destaque e coloração da sintaxe. O realce de sintaxe e a coloração alteram a cor de várias partes do programa para facilitar a identificação dos diferentes componentes do programa
- Os programas serão nomeados algo.cpp, onde algo é substituído pelo nome de sua escolha para o programa
- A extensão .cpp informa ao compilador que este é um arquivo de código-fonte C++ que contém instruções C++

## Etapa 4: Compilar seu código-fonte

- Para compilar um programa C++, usamos um compilador C++
- O compilador C++ examina sequencialmente cada arquivo de código-fonte (.cpp) em seu programa e executa duas tarefas importantes:
  - Ele verifica seu código para certificar-se de que segue as regras da linguagem C++  
Caso contrário, o compilador fornecerá um erro (e o número da linha correspondente) para ajudar a identificar o que precisa ser consertado. O processo de compilação também será interrompido até que o erro seja corrigido.
  - Ele traduz seu código-fonte C++ em um arquivo de linguagem de máquina denominado arquivo-objeto  
Os arquivos de objeto são normalmente denominados name.o ou name.obj, em que name é o mesmo nome do arquivo .cpp a partir do qual foi produzido.

## Etapa 4: Compilar seu código-fonte - 2

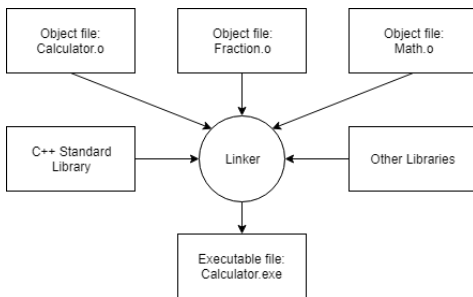




## Etapa 5 :Combinar arquivos de objetos e bibliotecas

- Depois que o compilador cria um ou mais arquivos-objeto, outro programa chamado **ligador**, **vinculador** ou **editor de ligação** entra em ação
- O trabalho do vinculador é triplo:
  - Pegue todos os arquivos-objeto gerados pelo compilador e combine-os em um único programa executável.
  - Além de ser capaz de vincular arquivos de objeto, o vinculador também é capaz de vincular arquivos de biblioteca.  
Um **arquivo de biblioteca** é uma coleção de código pré-compilado que foi “empacotado” para reutilização em outros programas
  - O vinculador garante que todas as dependências de arquivo cruzado sejam resolvidas corretamente.  
**EX:** se você definir algo em um arquivo .cpp e, em seguida, usá-lo em outro arquivo .cpp, o vinculador conecta os dois. Se o vinculador não conseguir conectar uma referência a algo com sua definição, você receberá um erro do vinculador e o processo de vinculação será abortado.

## Etapa 5: Combinar arquivos de objetos e bibliotecas - 2



- A linguagem central do C++ é bastante pequena e concisa
- No entanto, C++ também vem com uma extensa biblioteca chamada C++ Standard Library, que fornece funcionalidade adicional que você pode usar em seus programas
- Uma das partes mais comumente usadas da biblioteca padrão C++ é a biblioteca `iostream`, que contém funcionalidade para imprimir texto em um monitor e obter entrada de teclado de um usuário
- Você também pode opcionalmente vincular com outras bibliotecas

## Etapas 6 e 7: teste e depuração

- Você pode rodar seu executável e ver se ele produz a saída que você esperava
- Se o seu programa é executado, mas não funciona corretamente, é hora de alguma depuração para descobrir o que está errado

## ■ Duas opções:

- Ambiente de desenvolvimento integrado (*Integrated Development Environment - IDE*) é um software que contém todas as coisas que você precisa para **desenvolver**, **compilar**, **vincular** e **depurar** seus programas
- Editor de texto com Linha de Comando (compilar, vincular)

## ■ Nessa disciplina usamos a segunda opção:

- Liberdade para a escolha do editor de texto  
Editor sugerido: Atom (<https://atom.io/>)
- Uso de um compilador Open Source (GNU GCC/G++ Compiler)  
Instalação no Windows:  
<https://www.youtube.com/watch?v=KYTGaxyZCwI>
- Uso de Makefile:  
[https://sourceforge.net/projects/gnuwin32/files/make/3.81/make-3.81.exe/download?use\\_mirror=megalink&download=](https://sourceforge.net/projects/gnuwin32/files/make/3.81/make-3.81.exe/download?use_mirror=megalink&download=)  
**Atualizar a Variável de Ambiente PATH para Make como foi feito para o Mingw-w64**

- A janela, que geralmente é chamada de linha de comando ou interface de linha de comando, é uma aplicação de texto para ver e manipular arquivos em seu computador. É como o Windows Explorer ou o Finder no Mac, mas sem a interface gráfica. Outros nomes para a linha de comando são: cmd, CLI, prompt, console ou terminal.
- Abra a interface de linha de comando:
  - Vá para o menu ou tela de iniciar, e digite "Prompt de comando" no campo de busca
  - Vá para o menu Iniciar → Sistema Windows → Prompt de comando.
  - Vá em Iniciar → Todos os Programas → Acessórios → Prompt de comando.

- Pasta atual  
`cd`
- Listando arquivos e pastas  
`dir`
- Mudar a pasta atual  
`cd Desktop`
- Arrumando  
`cd ..`
- Criando Pastas  
`mkdir practice`

# Linha de comando - 3

Comando (Windows)	Comando (Mac OS / Linux)	Descrição	Exemplo
exit	exit	Fecha a janela	<b>exit</b>
cd	cd	Muda a pasta	<b>cd test</b>
cd	pwd	Mostra o diretório atual	<b>cd</b> (Windows) ou <b>pwd</b> (Mac OS / Linux)
dir	ls	Lista as pastas e/ou arquivos	<b>dir</b>
copy	cp	Copia um arquivo	<b>copy c:\test\test.txt</b> <b>c:\windows\test.txt</b>
move	mv	Movimenta um arquivo	<b>move c:\test\test.txt</b> <b>c:\windows\test.txt</b>
mkdir	mkdir	Cria uma pasta	<b>mkdir testdirectory</b>
rmdir (ou del)	rm	Exclui arquivo	<b>del c:\test\test.txt</b>
rmdir /S	rm -r	Exclui diretório	<b>rm -r testdirectory</b>
[CMD] /?	man [CMD]	obtem ajuda para um comando	<b>cd /?</b> (Windows) or <b>man cd</b> (Mac OS / Linux)

# Estrutura de um programa



- Um programa de computador é uma sequência de instruções que dizem ao computador o que fazer
- Uma declaração é o que faz com que o programa execute alguma ação
- Eles são a menor unidade de computação independente na linguagem C++
- A maioria das instruções em C++ termina em ponto-e-vírgula
- Existem muitos tipos diferentes de instruções em C++:
  - Instruções de declaração
  - Instruções de salto
  - Instruções de expressão
  - Instruções compostas
  - Instruções de seleção (condicionais)
  - Instruções de iteração (laços)
  - Tente blocos

# Funções e a função principal

- Em C++, as instruções são normalmente agrupadas em unidades chamadas funções
- Uma função é uma coleção de instruções que são executadas sequencialmente (em ordem, de cima para baixo)
- **Todo programa C++ deve ter uma função especial chamada `main` (todas as letras minúsculas)**
- Quando o programa é executado, as instruções dentro de *main* são executadas em ordem sequencial
- Os programas normalmente terminam (terminam a execução) quando a última instrução dentro da função `main` foi executada
- As funções são normalmente escritas para fazer um trabalho específico

# Primeiro programa

```
#include <iostream>

int main()
{
    std::cout << "Hello, \uworld!";
    return 0;
}
```

- A linha 1 é um tipo especial de linha chamada diretiva de pré-processador.  
Esta diretiva de pré-processador indica que gostaríamos de usar o conteúdo da biblioteca iostream, que é a parte da biblioteca padrão C++ que nos permite ler e escrever texto de/para o console
- A linha 2 está em branco e é ignorada pelo compilador.
- A linha 3 diz ao compilador que iremos escrever (definir) uma função chamada main.

## Primeiro programa - 2

```
#include <iostream>

int main()
{
    std::cout << "Hello, \uworld!";
    return 0;
}
```

- As linhas 4 e 7 informam ao compilador quais linhas fazem parte da função principal. Tudo entre a chave de abertura na linha 4 e a chave de fechamento na linha 7 é considerado parte da função principal. Isso é chamado de corpo da função.
- A linha 5 é a primeira instrução dentro da função main e é a primeira instrução que será executada quando executarmos nosso programa. `std::cout` (que significa “saída de caractere”) e o operador `<<` nos permite enviar letras ou números para o console. Esta declaração cria a saída visível do programa.

## Primeiro programa - 3

```
#include <iostream>

int main()
{
    std::cout << "Hello, \uworld!";
    return 0;
}
```

- A linha 6 é uma declaração de retorno. Quando um programa executável termina de ser executado, o programa envia um valor de volta ao sistema operacional para indicar se foi executado com êxito ou não.

Essa instrução de retorno especifica retorna o valor 0 para o sistema operacional, o que significa “tudo deu certo!”.

Esta é a última instrução do programa que é executado.

# Comentário

- Um comentário é uma nota legível pelo programador inserida diretamente no código-fonte do programa.
- Os comentários são ignorados pelo compilador e são apenas para uso do programador.
- O símbolo `//` começa um comentário C++ de uma linha, que diz ao compilador para ignorar tudo, desde o símbolo `//` até o final da linha.

```
/* Programação avançada
 * UFPE
 * DEMEC */

#include <iostream>

int main()
{
    std::cout << "Hello, \uworld!"; // prints Hello World!
    return 0;
}
```

- O par de símbolos `/ *` e `* /` denota um comentário de várias linha. Tudo entre os símbolos é ignorado
- **Comente seu código e escreva seus comentários como se estivesse falando com alguém que não tem ideia do que o código faz. Não presuma que você vai se lembrar por que fez escolhas específicas.**

```
/* Programação avançada
 * UFPE
 * DE MEC */

#include <iostream>

int main()
{
    std::cout << "Hello, world!"; // prints Hello World!
    return 0;
}
```

- Todos os elementos da biblioteca padrão C++ são declarados dentro do que é chamado de namespace: o namespace `std`.
- Para se referir aos elementos no namespace `std`, um programa deve qualificar todo e qualquer uso de elementos da biblioteca (como fizemos prefixando `cout` com `std::`), ou introduzir visibilidade de seus componentes.
- A maneira mais comum de apresentar visibilidade desses componentes é por meio do uso de declarações:

```
using namespace std;
```

- A declaração acima permite que todos os elementos no namespace `std` sejam acessados de maneira não qualificada (sem o prefixo `std::`).



# Compilar e executar o primeiro programa

- Copie o programa para um arquivo e salve-o com uma extensão `.cpp` (ex: `p1.cpp`).
- Abra um terminal e vá para o diretório que contém o arquivo.
- Para compilar (criação do arquivo objeto), digite no terminal:

```
g++ -c p1.cpp
```

A opção `-c` indica ao compilador `g++` que ele deve criar um arquivo objeto (compilação)

## Compilar e executar - 2

- Para ligar (criação do arquivo executável), digite no terminal:

```
g++ p1.obj % for Windows
```

```
g++ p1.o % for Linux-Mac
```

Nesse caso o nome do arquivo executável é **a.exe** (Windows) ou **a.out** (Linux)

Usamos a opção `-o` para definir o nome do arquivo executável

```
g++ p1.obj -o p1.exe % for Windows
```

- Para executar, digite no terminal:

```
p1.exe % for Windows
```

```
./p1 % for Linux
```

# Variáveis e tipos

# Introdução a objetos e variáveis

- Os programas produzem resultados manipulando (lendo, alterando e gravando) **dados**.
- Na computação, **dados** são quaisquer informações que podem ser movidas, processadas ou armazenadas por um computador.
- Todos os computadores têm **memória**, chamada RAM (abreviação de memória de acesso aleatório), que está disponível para uso pelos programas.
- Um dado, armazenado na memória, é chamado de **valor**.
- Um **objeto** é uma região de armazenamento (geralmente memória) que tem um valor
- Os objetos podem ser nomeados ou não nomeados (anônimos).
- Um objeto nomeado é chamado de **variável** e o nome do objeto é chamado de **identificador**.
- Em nossos programas, a maioria dos objetos que criamos e usamos serão variáveis.

# Identificador

- Um identificador válido é uma sequência de uma ou mais letras, dígitos ou caracteres de sublinhado (`_`)
- Espaços, sinais de pontuação e símbolos não podem fazer parte de um identificador
- Os identificadores devem sempre começar com uma letra
- A linguagem C++ é uma linguagem que diferencia maiúsculas de minúsculas
- C++ usa várias palavras-chave para identificar operações e descrições de dados; portanto, os identificadores criados por um programador não podem corresponder a essas palavras-chave

`alignas, alignof, and, and_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16_t, char32_t, class, compl, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not_eq, nullptr, operator, or, or_eq, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, xor, xor_eq`

- Os valores das variáveis são armazenados em algum lugar em um local não especificado na memória do computador como zeros e uns
- Nosso programa não precisa saber o local exato onde uma variável está armazenada; ele pode simplesmente se referir a ele pelo nome
- O que o programa precisa estar ciente é o tipo de dados armazenados na variável
- Armazenar um inteiro simples não é o mesmo que armazenar uma letra: não são interpretados da mesma forma e, em muitos casos, não ocupam a mesma quantidade de memória

## Tipos de dados fundamentais - 2

- Os tipos de dados fundamentais são tipos básicos implementados diretamente pela linguagem que representam as unidades básicas de armazenamento suportadas nativamente pela maioria dos sistemas. Eles podem ser classificados principalmente em:
- **Tipos de caracteres:** Eles podem representar um único caractere, como 'A' ou '\$'. O tipo mais básico é `char`, que é um caractere de um byte. Outros tipos também são fornecidos para caracteres mais largos.
- **Tipos inteiros numéricos:** eles podem armazenar um valor de número inteiro, como 7 ou 1024. Eles existem em uma variedade de tamanhos e podem ser assinados ou não, dependendo se suportam valores negativos ou não.
- **Tipos de vírgula flutuante:** podem representar valores reais, como 3,14 ou 0,01, com diferentes níveis de precisão, dependendo de qual dos três tipos de vírgula flutuante é usado.
- **Tipo booleano:** o tipo booleano, conhecido em C++ como `bool`, pode representar apenas um de dois estados, verdadeiro ou falso.



# Tipos de dados fundamentais - 3

Group	Type names*	Notes on size / precision
Character types	<code>char</code>	Exactly one byte in size. At least 8 bits.
	<code>char16_t</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>char32_t</code>	Not smaller than <code>char16_t</code> . At least 32 bits.
	<code>wchar_t</code>	Can represent the largest supported character set.
Integer types (signed)	<code>signed char</code>	Same size as <code>char</code> . At least 8 bits.
	<code>signed short int</code>	Not smaller than <code>char</code> . At least 16 bits.
	<code>signed int</code>	Not smaller than <code>short</code> . At least 16 bits.
	<code>signed long int</code>	Not smaller than <code>int</code> . At least 32 bits.
	<code>signed long long int</code>	Not smaller than <code>long</code> . At least 64 bits.
Integer types (unsigned)	<code>unsigned char</code>	(same size as their signed counterparts)
	<code>unsigned short int</code>	
	<code>unsigned int</code>	
	<code>unsigned long int</code>	
	<code>unsigned long long int</code>	
Floating-point types	<code>float</code>	
	<code>double</code>	Precision not less than <code>float</code>
	<code>long double</code>	Precision not less than <code>double</code>
Boolean type	<code>bool</code>	
Void type	<code>void</code>	no storage
Null pointer	<code>decltype(nullptr)</code>	

# Tipos de dados fundamentais - 4

- Os tamanhos de tipo acima são expressos em bits; quanto mais bits um tipo tem, mais valores distintos ele pode representar, mas ao mesmo tempo, também consome mais espaço na memória:

Size	Unique representable values	Notes
8-bit	256	$= 2^8$
16-bit	65 536	$= 2^{16}$
32-bit	4 294 967 296	$= 2^{32}$ (~4 billion)
64-bit	18 446 744 073 709 551 616	$= 2^{64}$ (~18 billion billion)

- Para tipos inteiros, ter mais valores representáveis significa que o intervalo de valores que eles podem representar é maior
- Para tipos de ponto flutuante, o tamanho afeta sua precisão, por ter mais ou menos bits para seu significante e expoente
- Se o tamanho ou a precisão do tipo não for uma preocupação, `char`, `int` e `double` são normalmente selecionados para representar caracteres, inteiros e valores de ponto flutuante, respectivamente.

# Instanciação de variável

- Para criar uma variável, usamos um tipo especial de declaração chamada **definição**

```
int x; // define a variable named x, of type int
```

- Em tempo de compilação, quando o compilador vê essa instrução, ele faz uma nota para si mesmo que estamos definindo uma variável, dando a ela o nome `x`, e que ela é do tipo `int`.  
Desse ponto em diante sempre que o compilador vir o identificador `x`, ele saberá que estamos fazendo referência a essa variável.
- Quando o programa é executado (denominado runtime), a variável é instanciada.
- **Instanciação** ou **declaração** significa que o objeto será criado e receberá um endereço de memória.
- As variáveis devem ser instanciadas antes de serem usadas para armazenar valores.

- Após a definição de uma variável, você pode atribuir a ela um valor (em uma instrução separada) usando o operador =

```
int width; // define an integer variable named width  
width = 5; // copy assignment of value 5 into variable width
```

- Uma desvantagem da atribuição é que ela requer pelo menos duas instruções: uma para definir a variável e outra para atribuir o valor.
- Essas duas etapas podem ser combinadas. Quando uma variável é definida, você também pode fornecer um valor inicial para a variável ao mesmo tempo. Isso é chamado de **inicialização**. O valor usado para inicializar uma variável é chamado de inicializador.

```
int a = 5; // inicialização tipo c  
int a(5); // inicialização tipo construtor  
int a{5}; // inicialização uniforme
```

- Os tipos fundamentais representam os tipos mais básicos tratados pelas máquinas onde o código pode ser executado
- Mas um dos principais pontos fortes da linguagem C++ é seu rico conjunto de tipos compostos, dos quais os tipos fundamentais são meros blocos de construção
- Um exemplo de tipo composto é a classe `string`. Variáveis deste tipo podem armazenar sequências de caracteres, como palavras ou frases
- Uma primeira diferença com os tipos de dados fundamentais é que, para declarar e usar objetos (variáveis) desse tipo, o programa precisa incluir o cabeçalho onde o tipo é definido na biblioteca padrão (`header <string>`)

# Introdução às strings - 2

```
// my first string
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    string mystring;
    mystring = "This is the initial string content";
    cout << mystring << endl;
    mystring = "This is a different string content";
    cout << mystring << endl;
    return 0;
}
```

- Podemos começar a operar com às variáveis usando **operadores**.
- Operador de atribuição (=)
  - O operador de atribuição atribui um valor a uma variável.
  - A operação de atribuição sempre ocorre da direita para a esquerda.

```
x = y;
```

```
y = 2 + (x = 5);
```

```
x = y = z = 5;
```



- $+$ : adição
- $-$ : subtração
- $*$ : multiplicação
- $/$ : divisão
- $\%$ : modulo, resto de uma divisão de dois valores

# Atribuição composta

- Os operadores de atribuição composta modificam o valor atual de uma variável executando uma operação nela.

```
y += x;    →  y = y + x;
```

```
x -= 5;    →  x = x - 5;
```

```
x /= y;    →  x = x / y;
```

```
price *= units + 1; → price = price * (units+1);
```

# Incremento e decremento

- O operador de aumento ( $++$ ) e o operador de diminuição ( $--$ ) aumentam ou reduzem em um o valor armazenado em uma variável

`++x; → x++; → x+=1; → x=x+1;`

- Uma peculiaridade desse operador é que ele pode ser usado tanto como prefixo quanto como sufixo
  - Caso o operador de aumento seja utilizado como prefixo ( $++x$ ) do valor, a expressão avalia para o valor final de  $x$ , uma vez que já esteja aumentado.
  - Caso seja usado como sufixo ( $x++$ ), o valor também é aumentado, mas a expressão avalia o valor que  $x$  tinha antes de ser aumentada.

```
x = 3;
```

```
y = ++x; // x contains 4, y contains 4
```

```
x = 3;
```

```
y = x++; // x contains 4, y contains 3
```

# Operadores relacionais e de comparação

- Duas expressões podem ser comparadas usando operadores relacionais e de igualdade
- O resultado de tal operação é verdadeiro ou falso (ou seja, um valor booleano)
- `==`: Igual a
- `!=`: Diferente de
- `<`: Menor que
- `>`: Maior que
- `<=`: Menor ou igual a
- `>=`: Maior ou igual a

- O operador `!` é o operador C++ para a operação booleana NOT. Retorna o valor booleano oposto da avaliação de seu operando

```
!(5 == 5) // evaluates to false because the expression at its  
right (5 == 5) is true
```

```
!(6 <= 4) // evaluates to true because (6 <= 4) would be  
false
```

- Os operadores lógicos `&&` e `||` são usados ao avaliar duas expressões para obter um único resultado relacional.
- O operador `&&` corresponde à operação lógica booleana AND, que resulta em verdadeiro se ambos os operandos forem verdadeiros, e falso em caso contrário.
- O operador `||` corresponde à operação lógica booleana OR, que resulta em verdadeiro se algum de seus operandos for verdadeiro, sendo falso apenas quando ambos os operandos forem falsos.

```
( (5 == 5) && (3 > 6) ) // false ( true && false )  
( (5 == 5) || (3 > 6) ) // true  ( true || false )
```

# Operador ternário condicional (?)

- O operador condicional avalia uma expressão, retornando um valor se essa expressão for avaliada como verdadeira e um diferente se a expressão for avaliada como falsa.
- Sua sintaxe é:  
`condição ? resultado1 : resultado2`
- Se a condição for verdadeira, a expressão inteira será avaliada como resultado1 e, caso contrário, como resultado2.

```
7==5 ? 4 : 3 // evaluates to 3, since 7 is not equal to 5
```

```
7==5+2 ? 4 : 3 // evaluates to 4, since 7 is equal to 5+2
```

# Operador de conversão de tipo explícito

- Os operadores de conversão de tipo permitem converter um valor de um determinado tipo em outro tipo.
- Existem várias maneiras de fazer isso em C++.
- O mais simples é preceder a expressão a ser convertida pelo novo tipo entre parênteses ():

```
int i;  
float f = 3.14;  
i = (int) f;
```

- O código anterior converte o número de ponto flutuante 3,14 em um valor inteiro (3); o restante está perdido.



- Este operador aceita um parâmetro, que pode ser um tipo ou uma variável, e retorna o tamanho em bytes desse tipo ou objeto:

```
x = sizeof (char);
```

- Aqui, x recebe o valor 1, porque char é um tipo com tamanho de um byte.
- O valor retornado por sizeof é uma constante de tempo de compilação, portanto, é sempre determinado antes da execução do programa.

# Input/Output

- C++ usa uma abstração conveniente chamada streams (fluxo) para realizar operações de entrada e saída em mídia sequencial, como a tela, o teclado ou um arquivo.
- Um fluxo é uma entidade onde um programa pode inserir ou extrair caracteres.
- Não há necessidade de saber detalhes sobre a mídia associada ao fluxo ou qualquer uma de suas especificações internas.
- Tudo o que precisamos saber é que os streams são uma origem / destino de caracteres e que esses caracteres são fornecidos/aceitos sequencialmente (ou seja, um após o outro).
- A biblioteca padrão define um punhado de objetos de fluxo que podem ser usados para acessar o que são considerados as fontes e destinos padrão de caracteres pelo ambiente onde o programa é executado

## Saída padrão (cout)

- Na maioria dos ambientes de programa, a saída padrão é a tela e o objeto de fluxo C++ definido para acessá-lo é `cout`.
- Para operações de saída formatadas, `cout` é usado junto com o operador de inserção, que é escrito como `<<`.
- O operador `<<` insere os dados que o seguem no fluxo que o precede

```
cout << "Output sentence"; // prints Output sentence on
screen
cout << 120; // prints number 120 on screen
cout << x; // prints the value of x on screen
```

## Saída padrão (cout) - 2

- Múltiplas operações de inserção (<<) podem ser encadeadas em uma única instrução

```
cout << "I am " << age << "years old and my zipcode is " <<
zipcode;
```

- Em C++, um caractere de nova linha pode ser especificado como `\n`

```
cout << "First sentence.\n";
cout << "Second sentence.\n Third sentence.";
```

- Alternativamente, o manipulador `endl` também pode ser usado para quebrar linhas.

```
cout << "First sentence."<< endl;
cout << "Second sentence."<< endl;
```

## Entrada padrão (cin)

- Na maioria dos ambientes de programa, a entrada padrão é o teclado e o objeto de fluxo C++ definido para acessá-lo é `cin`.
- Para operações de entrada formatadas, `cin` é usado junto com o operador de extração, que é escrito como `>>`
- Este operador é seguido pela variável onde os dados extraídos são armazenados.

```
int age;  
  
cin >> age;
```

- A primeira instrução declara uma variável do tipo `int` chamada `idade`, e a segunda extrai de `cin` um valor para ser armazenado nela.
- Esta operação faz o programa esperar pela entrada de `cin` geralmente, isso significa que o programa irá esperar que o usuário insira alguma sequência com o teclado.
- Os caracteres introduzidos com o teclado são transmitidos ao programa apenas quando a tecla ENTER (ou RETURN) é pressionada

- A operação de extração em `cin` usa o tipo da variável após o operador `>>` para determinar como ele interpreta os caracteres lidos da entrada
- No exemplo, se o usuário inserir algo que não pode ser interpretado como um número inteiro, a operação de extração falhará.
- O programa continua sem definir um valor para a variável, produzindo resultados indeterminados se o valor da variável for usado posteriormente.

- Para obter uma linha inteira de `cin`, existe uma função, chamada `getline`, que recebe o stream (`cin`) como primeiro argumento e a variável `string` como segundo.

```
string mystr;  
cout << "What's your name?  ";  
getline (cin, mystr);  
cout << "Hello " << mystr << endl;
```



- O cabeçalho padrão `<sstream>` define um tipo chamado `stringstream` que permite que uma `string` seja tratada como um fluxo e, portanto, permite operações de extração ou inserção de / para strings da mesma forma como são realizadas em `cin` e `cout`.
- Este recurso é mais útil para converter strings em valores numéricos e vice-versa

```
string mystr ("1204");  
  
int myint;  
  
stringstream(mystr) >> myint;
```

- Com essa abordagem de obter linhas inteiras e extrair seus conteúdos, separamos o processo de obter a entrada do usuário de sua interpretação como dados, permitindo que o processo de entrada seja o que o usuário espera e, ao mesmo tempo, ganhando mais controle sobre a transformação de seus conteúdo em dados úteis pelo programa.

## stringstream - 2

```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
    string mystr;
    float price=0;
    int quantity=0;

    cout << "Enter price: ";
    getline (cin,mystr);
    stringstream(mystr) >> price;
    cout << "Enter quantity: ";
    getline (cin,mystr);
    stringstream(mystr) >> quantity;
    cout << "Total price: " << price*quantity << endl;
    return 0;
}
```

# Controle de fluxo

- Uma instrução C++ simples é cada uma das instruções individuais de um programa, como as declarações de variáveis e expressões vistas nas seções anteriores.
- Eles sempre terminam com um ponto-e-vírgula (;) e são executados na mesma ordem em que aparecem em um programa.
- Mas os programas não se limitam a uma sequência linear de afirmações.
- Durante seu processo, um programa pode repetir segmentos de código ou tomar decisões e bifurcar.
- Para esse propósito, C++ fornece instruções de controle de fluxo que servem para especificar o que deve ser feito por nosso programa, quando e sob quais circunstâncias.

- A palavra-chave `if` é usada para executar uma instrução ou bloco, se, e somente se, uma condição for atendida  
`if (condition) statement`
- `condition` é a expressão que está sendo avaliada.
- Se esta condição for verdadeira, a instrução é executada.
- Se for falso, a instrução não é executada (é simplesmente ignorada) e o programa continua logo após a instrução de seleção inteira.

```
if (x == 100)
{
    cout << "x is ";
    cout << x;
}
```

```
if (x == 100) { cout << "x is "; cout << x; }
```

- As instruções de seleção com `if` também podem especificar o que acontece quando a condição não é atendida, usando a palavra-chave `else` para introduzir uma instrução alternativa  
`if (condition) statement1 else statement2`
- `statement1` é executada caso a condição seja verdadeira
- Caso não seja, `statement2` é executada

```
if (x == 100)
    cout << "x is 100";
else
    cout << "x is not 100";
```

- Várias estruturas if + else podem ser concatenadas com a intenção de verificar um intervalo de valores.

```
if (x > 0)
    cout << "x is 100";
else if (x < 0)
    cout << "x is not 100";
else
    cout << "x is 0";
```

# While

- O tipo mais simples de laço é o laço while  
while (expression) statement
- O laço while simplesmente repete statement enquanto a expression for verdadeira.
- Se, após qualquer execução da instrução, expression não for mais verdadeira, o laço termina e o programa continua logo após o laço

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

int main ()
{
    int n = 10;

    while (n>0) {
        cout << n << ",□";
        --n;
    }

    cout << "liftoff!\n";
}
```



# Do-While

- Um laço muito semelhante é o laço do-while do statement while (condition);
- Ele se comporta como um laço while, exceto que a condição é avaliada após a execução da instrução em vez de antes, garantindo pelo menos uma execução da instrução, mesmo se a condição nunca for cumprida.

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

int main ()
{
    string str;
    do {
        cout << "Enter text: ";
        getline (cin, str);
        cout << "You entered: " << str << '\n';
    } while (str != "goodbye");
}
```

- O laço `for` é projetado para iterar um determinado número de vezes `for (initialization; condition; increase) statement;`
- Como o laço `while`, este laço repete a instrução enquanto a condição `for` verdadeira.
- Mas, além disso, o laço `for` fornece locais específicos para incluir uma inicialização e uma expressão de aumento, executada antes do laço começar pela primeira vez e após cada iteração, respectivamente
- Portanto, é especialmente útil usar variáveis de contador como condição.

- 1 - a inicialização é executada. Geralmente, isso declara uma variável de contador e a define para algum valor inicial. Isso é executado uma única vez, no início do laço.
- 2 - a condição é verificada. Se for verdade, o laço continua; caso contrário, o laço termina e a instrução é ignorada, indo diretamente para a etapa 5.
- 3 - declaração é executada. Como de costume, pode ser uma única instrução ou um bloco entre chaves.
- 4 - o aumento é executado e o laço volta para a etapa 2.
- 5 - o laço termina: a execução continua na próxima instrução após ela.

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

int main ()
{
    for (int n=10; n>0; n--) {
        cout << n << ", ";
    }
    cout << "liftoff!\n";
}
```

- break
- switch
- go to

# Funções

- As funções permitem estruturar programas em segmentos de código para realizar tarefas individuais.
- Em C++, uma função é um grupo de instruções que recebem um nome e que podem ser chamadas de algum ponto do programa.
- A sintaxe mais comum para definir uma função é:

`type name ( parameter1, parameter2, ...) {statements}`

- **type** é o tipo do valor retornado pela função.
- **name** é o identificador pelo qual a função pode ser chamada.
- **Parameters**(quantos forem necessários): Cada parâmetro consiste em um tipo seguido por um identificador, com cada parâmetro sendo separado do próximo por uma vírgula.

Cada parâmetro se parece muito com uma declaração de variável regular (por exemplo: `int x`) e, de fato, atua dentro da função como uma variável regular que é local para a função. O objetivo dos parâmetros é permitir a passagem de argumentos para a função a partir do local de onde ela é chamada.

- **statements** são o corpo da função. É um bloco de instruções entre colchetes `{}` que especificam o que a função realmente faz.

# Exemplo

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

int addition (int a, int b)
{
    int r;
    r=a+b;
    return r;
}


int main ()
{
    int z;
    z = addition (5,3);
    cout << "The result is " << z;
}
```



## Exemplo - 2

- Este programa está dividido em duas funções: `adição` e `principal`.
- Lembre-se de que, independentemente da ordem em que são definidos, um programa C++ sempre começa chamando `main`.
- Na verdade, `main` é a única função chamada automaticamente, e o código em qualquer outra função só é executado se sua função for chamada de `main`.
- A chamada para uma função segue uma estrutura muito semelhante à sua declaração.

```
int addition (int a, int b)  
z = addition ( 5 , 3 );
```



## Exemplo - 3

- No ponto em que a função é chamada de dentro de `main`, o controle é passado para a função de adição: aqui, a execução de `main` é interrompida e só será retomada quando a função de adição terminar.
- No momento da chamada da função, o valor de ambos os argumentos (5 e 3) são copiados para as variáveis locais `int a` e `int b` dentro da função.
- A declaração final dentro da função `return r;` termina a função e retorna o controle ao ponto onde a função foi chamada.
- O programa retoma seu curso em `main` retornando exatamente no mesmo ponto em que foi interrompido pela chamada da função.
- Mas, além disso, como a função tem um tipo de retorno, a chamada é avaliada como tendo um valor, e esse valor é o valor especificado na instrução de retorno que encerrou a função

```
int addition (int a, int b)
    ↓
z = addition ( 5 , 3 );
```

# Exemplo

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

int subtraction (int a, int b)
{
    int r;
    r=a-b;
    return r;
}

int main ()
{
    int x=5, y=3, z;
    z = subtraction (7,2);
    cout << "The first result is " << z << '\n';
    cout << "The second result is " << subtraction (7,2) << '\n';
    cout << "The third result is " << subtraction (x,y) << '\n';
    z= 4 + subtraction (x,y);
    cout << "The fourth result is " << z << '\n';
}
```

# Argumentos passados por valor e por referência

- Nas funções vistas anteriormente, os argumentos sempre foram passados por **valor**.
- Isso significa que, ao chamar uma função, o que é passado para a função são os valores desses argumentos no momento da chamada, que são copiados para as variáveis representadas pelos parâmetros da função.
- Em certos casos, porém, pode ser útil acessar uma variável externa de dentro de uma função.
- Para fazer isso, os argumentos podem ser passados por **referência**, em vez de por valor.
- Quando uma variável é passada por referência, o que é passado não é mais uma cópia, mas a própria variável, a variável identificada pelo parâmetro da função, torna-se de alguma forma associada ao argumento passado para a função, e qualquer modificação em suas variáveis locais correspondentes dentro as funções são refletidas nas variáveis passadas como argumentos na chamada.

## Argumentos passados por valor e por referência - 2

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

void duplicate (int& a, int& b, int& c)
{
    a*=2;
    b*=2;
    c*=2;
}

int main ()
{
    int x=1, y=3, z=7;
    duplicate (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

## Argumentos passados por valor e por referência - 3

- Na verdade, a, b e c se tornam apelidos dos argumentos passados na chamada de função (x, y e z) e qualquer mudança em a dentro da função está, na verdade, modificando a variável x fora da função.
- Qualquer mudança em b modifica y, e qualquer mudança em c modifica z.

```
void duplicate (int& a,int& b,int& c)  
               ↑  
               x  
duplicate (    x    ,    y    ,    z    );  
               ↑    ↑    ↑  
               y    z
```

- **As funções não podem ser chamadas antes de serem declaradas.**
- É por isso que, em todos os exemplos anteriores de funções, as funções sempre foram definidas antes da função principal, que é a função a partir da qual as outras funções foram chamadas.
- Se `main` fosse definido antes das outras funções, isso quebraria a regra de que as funções devem ser declaradas antes de serem usadas e, portanto, não seriam compiladas.
- O protótipo de uma função pode ser declarado sem realmente definir a função completamente, fornecendo apenas detalhes suficientes para permitir que os tipos envolvidos em uma chamada de função sejam conhecidos.
- Naturalmente, a função deve ser definida em outro lugar, como mais tarde no código. Mas pelo menos, uma vez declarado assim, já pode ser chamado.

## Declarando funções - 2

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

void odd (int x);
void even (int x);

int main()
{
    int i;
    do {
        cout << "Please, enter number (0 to exit): ";
        cin >> i;
        odd (i);
    } while (i!=0);
    return 0;
}
```



## Declarando funções - 3

```
void odd (int x)
{
    if ((x%2)!=0) cout << "It is odd.\n";
    else even (x);
}

void even (int x)
{
    if ((x%2)==0) cout << "It is even.\n";
    else odd (x);
}
```

# Recursividade

- É a propriedade das funções serem chamadas por si mesmas
- É útil para algumas tarefas, como classificar elementos ou calcular o fatorial de números

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

long factorial (long a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return 1;
}

int main ()
{
    long number = 9;
    cout << number << "! = " << factorial (number);
    return 0;
}
```

# Sobrecarga de função (Overload)

- Em C++, duas funções diferentes podem ter o mesmo nome se
  - seus parâmetros forem diferentes
  - ou porque eles têm um número diferente de parâmetros
  - ou porque qualquer um de seus parâmetros é de um tipo diferente.
- O compilador sabe qual chamar examinando os tipos passados como argumentos quando a função é chamada.
- Geralmente, espera-se que duas funções com o mesmo nome tenham, pelo menos, um comportamento semelhante
- Observe que uma função não pode ser sobrecarregada apenas por seu tipo de retorno. Pelo menos um de seus parâmetros deve ter um tipo diferente.

## Sobrecarga de função - 2

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

int operate (int a, int b)
{
    return (a*b);
}

double operate (double a, double b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;
    double n=5.0,m=2.0;
    cout << operate (x,y) << '\n';
    cout << operate (n,m) << '\n';
    return 0;
}
```

# Template

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

int sum (int a, int b)
{
    return a+b;
}

double sum (double a, double b)
{
    return a+b;
}

int main ()
{
    cout << sum (10,20) << '\n';
    cout << sum (1.0,1.5) << '\n';
    return 0;
}
```

- A função sum pode ser sobrecarregada para vários tipos e pode fazer sentido que todos tenham o mesmo corpo

## Template - 2

- C++ tem a capacidade de definir funções com tipos genéricos, conhecidos como função template.
- A definição de função template segue a mesma sintaxe de uma função regular, exceto que é precedida pela palavra-chave `template` e uma série de parâmetros do template entre colchetes angulares `<>`:  
`template <template-parameters> function-declaration`
- Os parâmetros do template são uma série de parâmetros separados por vírgulas.
- Esses parâmetros podem ser tipos de modelo genéricos, especificando a palavra-chave `class` seguida por um identificador.
- Esse identificador pode então ser usado na declaração da função como se fosse um tipo regular.

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{return a+b;}
```

# Template - 3

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

template <class T>
T sum (T a, T b)
{
    T result;
    result = a + b;
    return result;
}

int main () {
    int i=5, j=6, k;
    double f=2.0, g=0.5, h;
    k=sum<int>(i,j);
    h=sum<double>(f,g);
    cout << k << '\n';
    cout << h << '\n';
    return 0;
}
```

# Template - 4

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

template <class T, class U>
bool are_equal (T a, U b)
{
    return (a==b);
}

int main ()
{
    if (are_equal(10,10.0))
        cout << "x and y are equal\n";
    else
        cout << "x and y are not equal\n";
    return 0;
}
```



# Visibilidade do nome

- Variáveis, funções e tipos compostos precisam ser declarados antes de serem usados em C++.
- O ponto no programa onde esta declaração acontece influencia sua visibilidade
- Uma entidade declarada fora de qualquer bloco tem escopo global, o que significa que seu nome é válido em qualquer parte do código.
- Uma entidade declarada dentro de um bloco, como uma função ou uma instrução seletiva, tem escopo de bloco e só é visível dentro do bloco específico em que é declarada, mas não fora dele.
- Variáveis com escopo de bloco são conhecidas como variáveis locais.

## Escopo - 2

```
int foo;           // global variable

int some_function ()
{
    int bar;       // local variable
    bar = 0;
}

int other_function ()
{
    foo = 1;  // ok: foo is a global variable
    bar = 2;  // wrong: bar is not visible from this function
}
```

- Em cada escopo, um nome pode representar apenas uma entidade.

```
int some_function ()
{
    int x;
    x = 0;
    double x;    // wrong: name already used in this scope
    x = 0.0;
}
```

## Escopo - 4

- A visibilidade de uma entidade com escopo de bloco se estende até o final do bloco, incluindo blocos internos.
- Um bloco interno, por ser um bloco diferente, pode reutilizar um nome existente em um escopo externo para se referir a uma entidade diferente

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

int main () {
    int x = 10;
    int y = 20;
    {
        int x;    // ok, inner scope.
        x = 50;   // sets value to inner x
        y = 50;   // sets value to (outer) y
        cout << "inner_block, x=" << x << "y=" << y << '\n';
    }
    cout << "outer_block, x=" << x << "y=" << y << '\n';
    return 0;
}
```

- Apenas uma entidade pode existir com um nome específico em um escopo específico
- Isso raramente é um problema para nomes locais, uma vez que os blocos tendem a ser relativamente curtos
- Nomes não locais trazem mais possibilidades de colisão de nomes, especialmente considerando que as bibliotecas podem declarar muitas funções, tipos e variáveis, nenhum deles de natureza local, e alguns deles muito genéricos.
- Os namespaces nos permitem agrupar entidades nomeadas que, de outra forma, teriam escopo global em escopos mais restritos, dando-lhes escopo de namespace.
- Isso permite organizar os elementos dos programas em diferentes escopos lógicos chamados por nomes

- A sintaxe para declarar namespaces é:

```
namespace identifier
{
    named_entities
}
```

- Onde `identifier` é qualquer identificador válido e `named_entities` é o conjunto de variáveis, tipos e funções incluídos no namespace.

# Namespaces - 3

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

namespace foo
{
    int value() { return 5; }
}

namespace bar
{
    const double pi = 3.1416;
    double value() { return 2*pi; }
}

int main () {
    cout << foo::value() << '\n';
    cout << bar::value() << '\n';
    cout << bar::pi << '\n';
    return 0;
}
```



- A palavra-chave `using` introduz um nome na região declarativa atual (como um bloco), evitando assim a necessidade de qualificar o nome.

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

namespace first
{
    int x = 5;
    int y = 10;
}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}
```

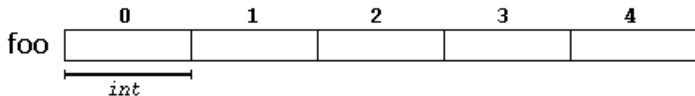
```
int main () {  
    using first::x;  
    using second::y;  
    cout << x << '\n';  
    cout << y << '\n';  
    cout << first::y << '\n';  
    cout << second::x << '\n';  
    return 0;  
}
```

```
int main () {  
    using namespace first;  
    cout << x << '\n';  
    cout << y << '\n';  
    cout << second::x << '\n';  
    cout << second::y << '\n';  
    return 0;  
}
```

# Arranjo

# Arranjo

- Um arranjo é uma **série** de elementos do **mesmo tipo** colocados em locais de memória **contíguos** que podem ser referenciados individualmente adicionando um índice a um identificador exclusivo.
- Por exemplo, cinco valores do tipo `int` podem ser declarados como um arranjo sem ter que declarar 5 variáveis diferentes
- Usando um arranjo, os cinco valores `int` são armazenados em locais de memória contíguos e todos os cinco podem ser acessados usando o mesmo identificador, com o índice adequado.
- Por exemplo, um arranjo contendo 5 valores inteiros do tipo `int`, chamado `foo`, pode ser representada como (os elementos são numerados de 0 a 4):



# Declaração de arranjo

- Um arranjo deve ser declarado antes de ser usado

`type name [elements];`

- Onde:

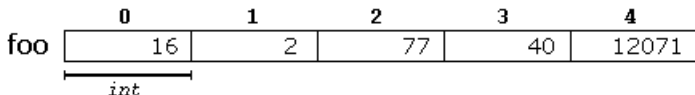
- `type` é um tipo válido (como `int`, `float` ...)
- `name` é um identificador válido
- `elements` especifica o comprimento do arranjo em termos de número de elemento (sempre entre colchetes)

- O campo de elementos entre colchetes, que representa o número de elementos no arranjo, deve ser uma expressão constante, pois os arranjos são blocos de memória estática cujo tamanho deve ser determinado em tempo de compilação, antes da execução do programa.

# Inicialização de arranjo

- Por padrão, os arranjos regulares não são inicializados
- Nenhum de seus elementos está definido com qualquer valor específico
- Os elementos em um arranjo podem ser inicializados explicitamente com valores específicos quando são declarados

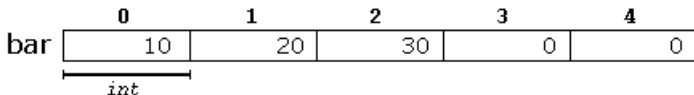
```
int foo [5] = { 16, 2, 77, 40, 12071 };
```



## Inicialização de arranjo - 2

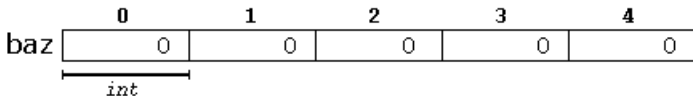
- O número de valores entre colchetes não deve ser maior que o número de elementos no arranjo
- Se declarado com menos, os elementos restantes são definidos com seus valores padrão

```
int bar [5] = { 10, 20, 30 };
```



- O inicializador pode não ter valores

```
int baz [5] = { };
```

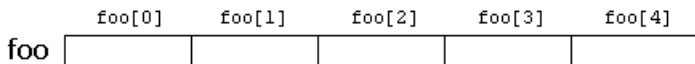




## Acessando os valores de um arranjo

- Os valores de qualquer um dos elementos em um arranjo podem ser acessados da mesma forma que o valor de uma variável regular do mesmo tipo

`name[index]`



```
foo[2] = 75;
```

```
int x = foo[2];
```

## Exemplo: arranjo

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

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
    for ( n=0 ; n<5 ; ++n )
    {
        result += foo[n];
    }
    cout << result;
    return 0;
}
```

# Arranjos multidimensionais

- Arranjos multidimensionais podem ser descritas como "arranjos de arranjos"

```
int jimmy [3][5];
```

		0	1	2	3	4
jimmy {	0					
	1					
	2					

```
jimmy[1][3];
```

		0	1	2	3	4
jimmy {	0					
	1					
	2					

↓  
**jimmy[1][3]**

# Exemplo: Arranjos multidimensionais

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
    for (n=0; n<HEIGHT; n++)
        for (m=0; m<WIDTH; m++)
        {
            jimmy[n][m]=(n+1)*(m+1);
        }
}
```

		0	1	2	3	4
jimmy	0	1	2	3	4	5
	1	2	4	6	8	10
	2	3	6	9	12	15

## Exemplo: Arranjos pseudo-multidimensionais

```
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
    for (n=0; n<HEIGHT; n++)
        for (m=0; m<WIDTH; m++)
        {
            jimmy[n*WIDTH+m]=(n+1)*(m+1);
        }
}
```

- Podemos precisar passar um arranjo para uma função como parâmetro
- Em C ++, não é possível passar todo o bloco de memória representado por um arranjo para uma função diretamente como um argumento
- Mas o que pode ser passado em vez disso é seu endereço
- Na prática, isso tem quase o mesmo efeito e é uma operação muito mais rápida e eficiente.

- Para aceitar um arranjo como parâmetro de uma função, os parâmetros podem ser declarados como o tipo de arranjo, mas com colchetes vazios

```
void procedure (int arg[])
```

- Para passar para esta função uma matriz declarada como:

```
int myarray [40];
```

- É suficiente escrever uma ligação como:

```
procedure (myarray);
```

# Exemplo

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

void printarray (int arg[], int length) {
    for (int n=0; n<length; ++n)
        cout << arg[n] << '␣';
    cout << '\n';
}

int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
}
```



## Arranjo multidimensional como argumento

- Em uma declaração de função, também é possível incluir arranjos multidimensionais
- O formato de um parâmetro de arranjo tridimensional é:

```
base_type[] [depth] [depth]
```

- Por exemplo, uma função com um arranjo multidimensional como argumento pode ser:

```
void procedure (int myarray[] [3] [4])
```

- Observe que os primeiros colchetes são deixados vazios, enquanto os seguintes especificam os tamanhos de suas respectivas dimensões
- Isso é necessário para que o compilador seja capaz de determinar a profundidade de cada dimensão adicional.
- De certa forma, passar um arranjo como argumento sempre perde uma dimensão

# Ponteiro

- Anteriormente, as variáveis eram explicadas como localizações na memória do computador que podem ser acessadas por seu identificador (seu nome)
- Dessa forma, o programa não precisa se preocupar com o endereço físico dos dados na memória; ele simplesmente usa o identificador sempre que precisa se referir à variável.
- Para um programa C++, a memória de um computador é como uma sucessão de células de memória, cada uma com um byte de tamanho e cada uma com um endereço único
- Essas células de memória de byte único são ordenadas de uma forma que permite que representações de dados maiores que um byte ocupem células de memória que possuem endereços consecutivos.

## Operador de endereço (&)

- O endereço de uma variável pode ser obtido precedendo o nome de uma variável com um sinal de e comercial (&), conhecido como operador de endereço

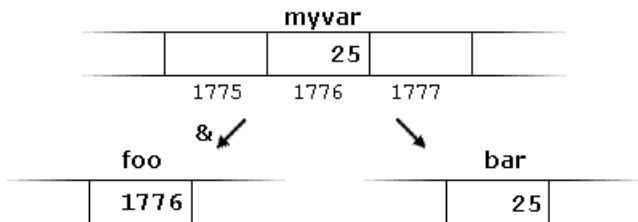
```
foo = &myvar;
```

- Precedendo o nome da variável `myvar` com o operador (&), não estamos mais atribuindo o conteúdo da própria variável a `foo`, mas seu endereço

## Operador de endereço (&) - 2

### ■ Exemplo

```
myvar = 25;  
foo = &myvar;  
bar = myvar;
```



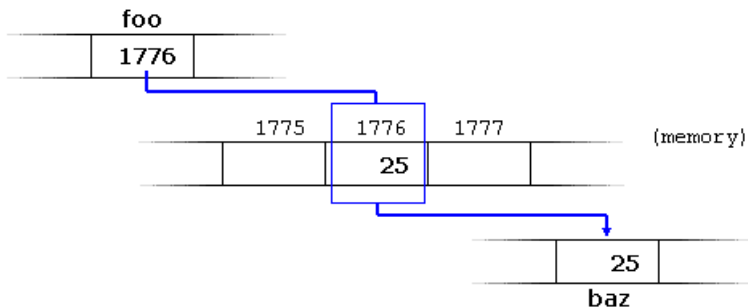
# Operador de desreferência (\*)

- Um ponteiro é uma variável que armazena o endereço de outra variável
- Ponteiros podem ser usados para acessar diretamente a variável para a qual eles apontam
- Isso é feito precedendo o nome do ponteiro com o operador de desreferência (\*)
- O operador pode ser lido como "valor apontado por"
- **Os operadores de referência e desreferência são, portanto, complementares:**
  - **&** é o operador de endereço e pode ser lido simplesmente como "endereço de"
  - **\*** é o operador de desreferência e pode ser lido como "valor apontado por"

# Operador de desreferência (\*) - 2

## ■ Exemplo

```
myvar = 25;  
foo = &myvar;  
baz = *foo;
```



- Devido à capacidade de um ponteiro se referir diretamente ao valor para o qual aponta, um ponteiro tem propriedades diferentes quando aponta para um `char` do que quando aponta para um `int` ou um `float`
- **Uma vez desreferenciado, o tipo precisa ser conhecido**
- E para isso, a declaração de um ponteiro precisa incluir o tipo de dados para o qual o ponteiro vai apontar  
`type * name;`



### ■ Exemplos:

```
int * number;  
char * character;  
double * decimals;
```

- Todos eles são ponteiros e provavelmente vão ocupar a mesma quantidade de espaço na memória
- Cada um destina-se a apontar para um tipo de dados diferente
- Os dados para os quais eles apontam não ocupam a mesma quantidade de espaço nem são do mesmo tipo

# Exemplo

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

int main ()
{
    int firstvalue, secondvalue;
    int * mypointer;

    mypointer = &firstvalue;
    *mypointer = 10;
    mypointer = &secondvalue;
    *mypointer = 20;
    cout << "firstvalue_is_" << firstvalue << '\n';
    cout << "secondvalue_is_" << secondvalue << '\n';
    return 0;
}
```

## Exemplo 2

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

int main ()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;

    p1 = &firstvalue; // p1 = address of firstvalue
    p2 = &secondvalue; // p2 = address of secondvalue
    *p1 = 10;          // value pointed to by p1 = 10
    *p2 = *p1;         // value pointed to by p2 = value pointed to by p1
    p1 = p2;           // p1 = p2 (value of pointer is copied)
    *p1 = 20;          // value pointed to by p1 = 20

    cout << "firstvalue is " << firstvalue << '\n';
    cout << "secondvalue is " << secondvalue << '\n';
    return 0;
}
```

- O conceito de arranjos está relacionado ao de ponteiros
- Na verdade, os arranjos funcionam de forma muito semelhante aos ponteiros para seus primeiros elementos e, na verdade, um arranjo sempre pode ser convertido implicitamente no ponteiro do tipo apropriado

```
int myarray [20];  
  
int * mypointer;  
  
mypointer = myarray;
```

- A principal diferença é que mypointer pode ser atribuído a um endereço diferente, enquanto myarray nunca pode ser atribuído a nada e sempre representará o mesmo bloco de 20 elementos do tipo int

# Exemplo

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

int main ()
{
    int numbers[5];
    int * p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p+4) = 50;
    for (int n=0; n<5; n++)
        cout << numbers[n] << ", " ;
    return 0;
}
```

## Ponteiro e arranjo - 2

- Ponteiros e arranjos suportam o mesmo conjunto de operações, com o mesmo significado para ambos
- A principal diferença é que os ponteiros podem receber novos endereços, enquanto os arranjos não
- Na parte sobre arranjos, os colchetes (`[]`) foram explicados como especificando o índice de um elemento da arranjo
- Na verdade esses colchetes são um operador de desreferenciamento conhecido como operador de deslocamento
- Eles desreferenciam a variável que seguem da mesma forma que `*`, mas também adicionam o número entre colchetes ao endereço que está sendo desreferenciado

```
a[5] = 0; // a [offset of 5] = 0
```

```
*(a+5) = 0; // pointed to by (a+5) = 0
```

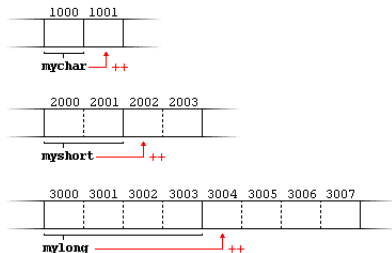
- Os ponteiros podem ser inicializados para apontar para locais específicos no momento em que são definidos:

```
int myvar;  
int * myptr = &myvar;
```

- Quando os ponteiros são inicializados, o que é inicializado é o endereço para o qual eles apontam, nunca o valor sendo apontado

# Aritmética de ponteiro

- Conduzir operações aritméticas em ponteiros é um pouco diferente de conduzi-las em tipos inteiros regulares
- Apenas as operações de adição e subtração são permitidas
- Tanto a adição quanto a subtração têm um comportamento ligeiramente diferente com os ponteiros, de acordo com o tamanho do tipo de dados para o qual eles apontam
- Ao adicionar um a um ponteiro, o ponteiro é feito para apontar para o seguinte elemento do mesmo tipo e, portanto, o tamanho em bytes do tipo para o qual ele aponta é adicionado ao ponteiro





- Essencialmente, essas são as quatro combinações possíveis do operador de desreferência com as versões de prefixo e sufixo do operador de incremento

```
*p++ // increment pointer, and dereference unincremented  
address
```

```
*++p // increment pointer, and dereference incremented  
address
```

```
++*p // dereference pointer, and increment the value it  
points to
```

```
(*p)++ // dereference pointer, and post-increment the value  
it points to
```

# Exemplo

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

int main ()
{
    int tab[2] {1,2};
    int* p_tab = tab;
    cout << p_tab << endl;

    cout << *p_tab++ << " " << p_tab << " " << *p_tab << endl;

    p_tab = tab; tab[0] = 1; tab[1] = 2;
    cout << **p_tab << " " << p_tab << " " << *p_tab << endl;

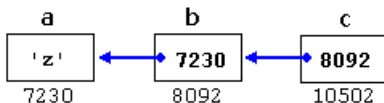
    p_tab = tab; tab[0] = 1; tab[1] = 2;
    cout << ++*p_tab << " " << p_tab << " " << *p_tab << endl;

    p_tab = tab; tab[0] = 1; tab[1] = 2;
    cout << (*p_tab)++ << " " << p_tab << " " << *p_tab << endl;
    return 0;
}
```

# Ponteiros para ponteiros

- C++ permite o uso de ponteiros que apontam para ponteiros, que estes, por sua vez, apontam para dados (ou mesmo para outros ponteiros)
- A sintaxe simplesmente requer um asterisco (\*) para cada nível de indireção na declaração do ponteiro

```
char a;  
char* b;  
char** c;  
a = 'z';  
b = &a;  
c = &b;
```



### change figure

- **c** é do tipo `char **` e um valor de 8092
- **\*c** é do tipo `char *` e um valor de 7230
- **\*\*c** é do tipo `char` e um valor de 'z'

# Ponteiros para funções

- C++ permite operações com ponteiros para funções.
- Um ponteiro de função é uma variável que armazena o endereço de uma função que pode ser chamada posteriormente por meio desse ponteiro de função.
- Ponteiros para funções são declarados com a mesma sintaxe de uma declaração de função regular, exceto que o nome da função é colocado entre parênteses () e um asterisco (\*) é inserido antes do nome

```
void (*foo)(int);
```

## Funções como argumentos para outras funções.

- Para uma rotina de classificação, você pode permitir que o chamador da função escolha a ordem na qual os dados são classificados: ordem crescente ou ordem decrescente.
- Uma maneira de permitir que seu usuário especifique o que fazer é fornecer um sinalizador como argumento para a função, mas isso é inflexível: todos os casos possíveis devem ser incluídos na função.// A função de classificação permite apenas um conjunto fixo de tipos de comparação
- Uma maneira muito melhor de permitir que o usuário escolha como classificar os dados é simplesmente permitir que o usuário passe uma função para a função de classificação.  
Essa função pode pegar dois dados e realizar uma comparação neles.

## Funções Callback

- Outro uso para ponteiros de função é configurar funções de "retorno de chamada" que são invocadas quando um determinado evento acontece. A função é chamada e isso notifica seu código de que algo de interesse ocorreu.
- Um exemplo é quando você está escrevendo código para uma interface gráfica do usuário (GUI). Às vezes, o usuário clica em um botão ou insere texto em um campo. Essas operações são "eventos" que podem exigir uma resposta que seu programa precisa manipular. Sempre que o botão for clicado, a função Callback será invocada.

```
void create_button( int x, int y, function callback_func );
```

# Exemplo

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

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}
```



## Exemplo - 2

```
int main ()
{
    int m,n;
    int (*minus)(int,int) = subtraction;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout <<n;
    return 0;
}
```

- Ponteiro e string
- Ponteiro e const
- Ponteiro e nullptr
- Ponteiro e void

# Memória dinâmica

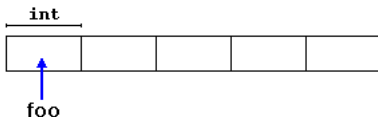
- Nos programas vistos anteriormente, todas as necessidades de memória foram determinadas antes da execução do programa, definindo as variáveis necessárias
- Mas pode haver casos em que as necessidades de memória de um programa só podem ser determinadas durante o tempo de execução
- Por exemplo, quando a memória necessária depende da entrada do usuário
- Nestes casos, os programas precisam alocar memória dinamicamente, para a qual a linguagem C++ integra os operadores `new` e `delete`

- A memória dinâmica é alocada usando o operador `new`
- `new` é seguido por um especificador de tipo de dados e, se uma sequência de mais de um elemento for necessária, o número deles entre colchetes `[]`
- Ele retorna um ponteiro para o início do novo bloco de memória alocado

```
pointer = new type
pointer = new type [number_of_elements]
```
- A primeira expressão é usada para alocar memória para conter um único elemento do tipo `tipo`
- O segundo é usado para alocar um bloco (um arranjo) de elementos do tipo `type`, onde `number_of_elements` é um valor inteiro que representa a quantidade destes

```
int * foo;  
foo = new int [5];
```

- Nesse caso, o sistema aloca dinamicamente espaço para cinco elementos do tipo `int` e retorna um ponteiro para o primeiro elemento da sequência, que é atribuído a `foo` (um ponteiro)
- Portanto, `foo` agora aponta para um bloco válido de memória com espaço para cinco elementos do tipo `int`
- Aqui, `foo` é um ponteiro e, portanto, o primeiro elemento apontado por `foo` pode ser acessado com `foo[0]` ou `*foo` (ambos são equivalentes)
- O segundo elemento pode ser acessado com `foo[1]` ou `*(foo+1)`, e assim por diante ...



- Há uma diferença substancial entre declarar um arranjo normal e alocar memória dinâmica para um bloco de memória usando `new`
- A diferença mais importante é que o tamanho de um arranjo regular precisa ser uma expressão constante e, portanto, seu tamanho deve ser determinado no momento de projetar o programa, antes de ser executado
- A alocação de memória dinâmica realizada por `new` permite atribuir memória durante o tempo de execução usando qualquer valor variável como tamanho

- Na maioria dos casos, a memória alocada dinamicamente só é necessária durante períodos específicos de tempo dentro de um programa
- Uma vez que não seja mais necessário, pode ser liberado para que a memória fique novamente disponível para outras solicitações de memória dinâmica

```
delete pointer;  
delete[] pointer;
```
- A primeira instrução libera a memória de um único elemento alocado usando `new`
- A segunda libera a memória alocada para arranjo de elementos usando `new` e um tamanho entre colchetes (`[]`)



# Exemplo

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

int main ()
{
    int i,n;
    int * p;
    cout << "How many numbers would you like to type? ";
    cin >> i;
    p= new int[i];

    for (n=0; n<i; n++)
    {
        cout << "Enter number: ";
        cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
        cout << p[n] << ", ";
    delete[] p;

    return 0;
}
```

# Estruturas de dados

- Uma estrutura de dados é um grupo de elementos de dados agrupados sob um nome
- Esses elementos de dados, conhecidos como **membros**, podem ter diferentes tipos e comprimentos diferentes

```
struct type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    .  
} object_names;
```
- `type_name` é um nome para o tipo de estrutura
- `object_name` pode ser um conjunto de identificadores válidos para objetos que possuem o tipo desta estrutura
- Entre chaves, há uma lista com os membros de dados, cada um é especificado com um tipo e um identificador válido como seu nome

```
struct product {  
    int weight;  
    double price;  
};  
  
product apple;  
product banana, melon;
```

- Isso declara um tipo de estrutura, denominado produto, e a define como tendo dois membros: peso e preço, cada um de um tipo fundamental diferente
- Essa declaração cria um novo tipo (product), que é então usado para declarar três objetos (variáveis) desse tipo
- Uma vez que o produto é declarado, ele é usado como qualquer outro tipo.

- Os objetos de estrutura apple, banana e melon podem ser declarados no momento em que o tipo de estrutura de dados é definido: beginblock

```
struct product {  
    int weigth;  
    double price;  
} apple, banana, melon;
```

- É importante diferenciar claramente entre o que é o nome do tipo de estrutura (product) e o que é um objeto desse tipo (apple, banana e melon)
- Muitos objetos (como apple, banana e melon) podem ser declarados de um único tipo de estrutura (produto).

- Uma vez que os objetos de um determinado tipo de estrutura são declarados, seus membros podem ser acessados diretamente
- A sintaxe para isso é simplesmente inserir um ponto (.) entre o nome do objeto e o nome do membro

```
apple.weight  
apple.price  
banana.weight  
...
```

- Cada um deles tem o tipo de dados correspondente ao membro ao qual se referem: `apple.weight` é do tipo `int`, enquanto `apple.price` é do tipo `double`

# Exemplo

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

struct movies_t {
    string title;
    int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
    string mystr;
    mine.title = "2001_A_Space_Odyssey";
    mine.year = 1968;

    cout << "Enter_title: ";
    getline (cin,yours.title);
    cout << "Enter_year: ";
    getline (cin,mystr);
    stringstream(mystr) >> yours.year;
```

## Exemplo - 2

```
cout << "My_favorite_movie_is:\n";
printmovie (mine);
cout << "And_yours_is:\n";
printmovie (yours);
return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << "\n(" << movie.year << ")\n";
}
```

- Um dos recursos das estruturas de dados é a capacidade de se referir a seus membros individualmente ou a toda a estrutura como um todo
- Em ambos os casos, usando o mesmo identificador: o nome da estrutura



## Exemplo 2

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

struct movies_t {
    string title;
    int year;
} films [3];

void printmovie (movies_t movie);

int main ()
{
    string mystr;
    int n;
```

## Exemplo 2 - 2

```
for (n=0; n<3; n++)
{
    cout << "Enter title: ";
    getline (cin,films[n].title);
    cout << "Enter year: ";
    getline (cin,mystr);
    stringstream(mystr) >> films[n].year;
}

cout << "\nYou have entered these movies:\n";
for (n=0; n<3; n++)
    printmovie (films[n]);
return 0;
}

void printmovie (movies_t movie)
{
    cout << movie.title;
    cout << " (" << movie.year << ")\n";
}
```

- Como qualquer outro tipo, as estruturas podem ser apontadas por seus próprios tipos de ponteiros

```
struct movies_t {  
    string title;  
    int year;  
};
```

```
movies_t amovie;  
movies_t * pmovie;  
pmovie = &amovie;
```

# Exemplo

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

struct movies_t {
    string title;
    int year;
};

int main ()
{
    string mystr;

    movies_t amovie;
    movies_t * pmovie;
    pmovie = &amovie;
```

## Exemplo - 2

```
cout << "Enter title: ";
getline (cin, pmovie->title);
cout << "Enter year: ";
getline (cin, mystr);
(stringstream) mystr >> pmovie->year;

cout << "\nYou have entered:\n";
cout << pmovie->title;
cout << " (" << pmovie->year << ")\n";

return 0;
}
```

## Operador de seta

- O operador de seta (`->`) é um operador de desreferência usado exclusivamente com ponteiros para objetos que possuem membros
- Este operador serve para acessar o membro de um objeto diretamente de seu endereço

```
pmovie->title
```

equivalente a

```
(*pmovie).title
```

- É algo diferente de:

```
*pmovie.title
```

```
*(pmovie.title)
```

# Input/Output com arquivos

- C++ oferece as seguintes classes para executar a saída e entrada de caracteres de/para arquivos:
  - `ofstream`: Classe de fluxo para escrever em arquivos
  - `ifstream`: Classe de fluxo para ler a partir de arquivos
  - `fstream`: Classe de fluxo para ler e escrever de/para arquivos.
- Estas classes são derivadas direta ou indiretamente das classes `istream` e `ostream`.
- Já utilizamos objetos cujos tipos eram estas classes: `cin` é um objeto de classe `istream` e `cout` é um objeto de classe `ostream`.
- Podemos usar nossos fluxos de arquivos da mesma forma que já estamos acostumados a usar `cin` e `cout`, com a única diferença de que temos que associar estes fluxos com arquivos físicos.



- A primeira operação geralmente realizada sobre um objeto de uma dessas classes é associá-lo a um arquivo real.
- Este procedimento é conhecido como abrir um arquivo.
- Um arquivo aberto é representado dentro de um programa por um fluxo e qualquer operação de entrada ou saída realizada neste objeto de fluxo será aplicada ao arquivo físico associado a ele.
- A fim de abrir um arquivo com um objeto de fluxo, utilizamos sua função de membro `open`:  
`open (filename, mode);`
- Onde `filename` é um `string` que representa o nome do arquivo a ser aberto, e `mode` é um parâmetro opcional com uma combinação de bandeiras.

<code>ios::in</code>	Aberto para operações de entrada
<code>ios::out</code>	Aberto para operações de saída
<code>ios::binary</code>	Aberto em modo binário
<code>ios::ate</code>	Definir a posição inicial no final do arquivo. Se esta bandeira não estiver definida, a posição inicial é o início do arquivo.
<code>ios::app</code>	Todas as operações de saída são realizadas no final do arquivo, anexando o conteúdo ao conteúdo atual do arquivo
<code>ios::trunc</code>	Se o arquivo for aberto para operações de saída e já existir, seu conteúdo anterior é deletado e substituído pelo novo conteúdo

- Todas estas bandeiras podem ser combinadas usando o operador bitwise OR (`||`).

```
ofstream myfile;  
myfile.open ("example.txt", ios::out | ios::app);
```

- Quando terminarmos nossas operações de entrada e saída em um arquivo, fechamos o arquivo para que o sistema operacional seja notificado e seus recursos fiquem novamente disponíveis.
- Para isso, chamamos a função de membro do fluxo `close`.
- Esta função de membro fecha o arquivo:  
`myfile.close();`
- Uma vez chamada esta função de membro, o objeto de fluxo pode ser reutilizado para abrir outro arquivo, e o arquivo fica disponível novamente para ser aberto por outros processos.

- As operações de escrita em arquivos de texto são realizadas da mesma forma que operamos com o cout:

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

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n";
        myfile << "This is another line.\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

- A leitura de um arquivo também pode ser feita da mesma forma que fizemos com o cin:

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

int main () {
    string line;
    ifstream myfile ("example.txt");
    if (myfile.is_open())
    {
        while ( getline (myfile,line) )
        {
            cout << line << '\n';
        }
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```