UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Modularização

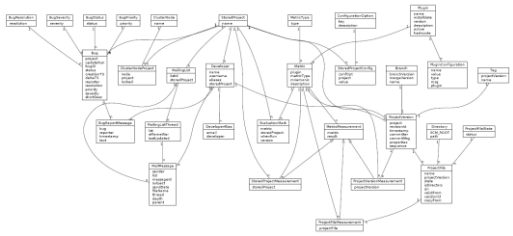
Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Introdução

- Classes
  - Agrupamento local (membros)
  - Problemas
    - Pouca representatividade em níveis mais abstratos
    - Difícil utilizar apenas isso em grandes aplicações
- Aumento no tamanho e complexidade
  - Necessário outra estrutura de organização

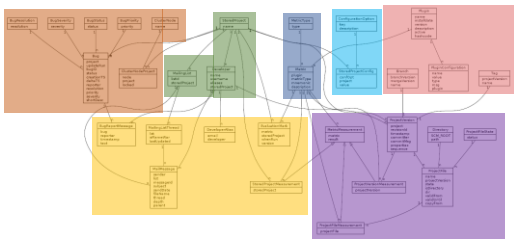
Introdução



Introdução

- "Agrupar para conquistar"
  - Juntar elementos inter-relacionados
  - Manutenção, compreensão, ...
- Programação modular
  - Partes independentes e intercambiáveis
  - Aspectos da funcionalidade do programa

Introdução



Módulo

- Propósito único
- Interface apropriada com outros módulos
- Pode ser compilado separadamente
- Reutilizáveis e Modificáveis

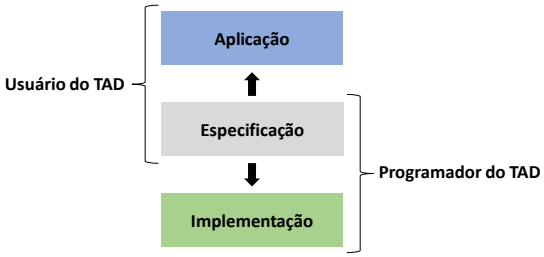
Compilação

- Grandes sistemas
  - Equipes de programadores
  - Código distribuído em vários arquivos fonte
- Não é conveniente recompilar partes (todo) do programa que não foram alteradas
- Princípio do encapsulamento
  - Separar a especificação de como a classe é usada dos detalhes da sua implementação

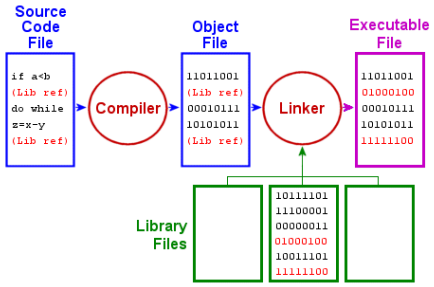
Compilação

- Definição da classe em arquivos que são separados dos programas que usam a classe
- Compile uma vez, use várias outras vezes
  - Bibliotecas → iostream, cstdlib
- Especificação (interface) x Implementação
  - Ao mudar a implementação (.cpp) da classe, somente esse arquivo deverá ser recompilado

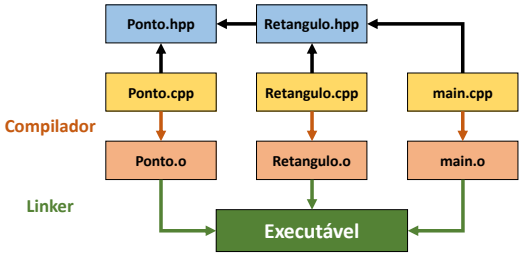
Compilação



Compilação



Compilação



Compilação

- Compilar e Ligar

```
g++ -o Executavel arquivo1.cpp arquivo2.cpp
```

- Compilar apenas

```
g++ -c arquivo.cpp  →  arquivo.o
```

- Ligar apenas

```
g++ -o Executavel arquivo1.o arquivo2.o
```

Compilação
Makefile

- Arquivo de texto especialmente formatado para um programa Unix chamado 'make'
- Contém uma lista de requisitos para que um programa seja considerado 'up to date'
  - O programa make examina esses requisitos, verifica os *timestamps* em todos os arquivos de origem listados no makefile e recompila apenas os arquivos com um registro desatualizado

https://www.gnu.org/software/make/manual/make.html
https://www.cs.bu.edu/teaching/cpp/writing-makefiles/

Compilação
Makefile

- Um makefile contém atribuições de variáveis, comentários e regras (targets)

```
target1 target2 ... : dependencia1 dependencia2 ...
    <TAB> comando1
    <TAB> comando2
    ...
```

```
helloworld: helloworld.cpp
    g++ -o helloworld helloworld.cpp
```


Compilação
Makefile

```
CC=g++
CFLAGS=-std=c++11 -Wall

all: main

Ponto.o: Ponto.hpp Ponto.cpp
    ${CC} ${CFLAGS} -c Ponto.cpp

main.o: Ponto.hpp main.cpp
    ${CC} ${CFLAGS} -c main.cpp

main: main.o Ponto.o
    ${CC} ${CFLAGS} -o main main.o Ponto.o

# Rule for cleaning files generated during compilation.
# Call 'make clean' to use it
clean:
    rm -f main *.o
```


Compilação
Makefile

```
> make
g++ -std=c++11 -Wall -c main.cpp
g++ -std=c++11 -Wall -c Ponto.cpp
g++ -std=c++11 -Wall -o main main.o Ponto.o
> ./main
> make      (Após alteração apenas em Ponto.cpp)
g++ -std=c++11 -Wall -c Ponto.cpp
g++ -std=c++11 -Wall -o main main.o Ponto.o
> ./main
> make
make: Nothing to be done for 'all'.
```


Namespace

- Com um maior número de arquivos, aumenta a chance de conflitos de nomes
- C++: Namespaces
  - Espaço de nomes (escopo) restrito
  - Diferente de classes, a redefinição de um *namespace* continua a definição anterior
- Java: Packages
  - Comparação mais próxima

Namespace

```
namespace Modulo1 {

  class ClasseA {
    public:
      ClasseA() {
        std::cout << "Modulo1::ClasseA" << std::endl;
      }
  };
}
```

```
namespace Modulo2 {

  class ClasseA {
    public:
      ClasseA() {
        std::cout << "Modulo2::ClasseA" << std::endl;
      }
  };
}
```

Namespace

```
int main() {

    Modulo1::ClasseA c1;
    Modulo2::ClasseA c2;

}
```


Namespace

```
namespace Modulo1 {

  class ClasseB {
    public:
      ClasseB() {
        std::cout << "Modulo1::ClasseB" << std::endl;
      }
  };

}
```


Namespace
Organização

- Separação em diferentes diretórios
  - Agrupamento físico de algo lógico

```
. project
├── Makefile
├── build
│   └── [objects]
├── include
│   └── modulo1
│       ├── mod1c1.hpp
│       └── mod1c2.hpp
└── src
    ├── main.cpp
    └── modulo1
        ├── mod1c1.cpp
        └── mod1c2.cpp
    └── test
```


Namespace
Exemplo

```
. project
├── Makefile
├── build
├── include
│   └── automovel
│       ├── Carro.hpp
│       ├── Ferrari.hpp
│       └── Ford.hpp
└── src
    ├── main.cpp
    └── automovel
        ├── Ferrari.cpp
        └── Ford.cpp
```


Namespace
Exemplo

Carro.hpp

```
#ifndef CARRO_H
#define CARRO_H

namespace Carro {

  class CarroAbstrato {

    public:
      virtual void start() = 0;
      virtual void drive() = 0;
      virtual void stop() = 0;
  };
}

#endif
```


Namespace
Exemplo

Ferrari.hpp

```
#ifndef FERRARI_H
#define FERRARI_H

#include "Carro.hpp"

namespace Carro {

  class Ferrari : public CarroAbstrato {

    public:
      void start();
      void drive();
      void stop();

  };

}

#endif
```

Ford.hpp

```
#ifndef FORD_H
#define FORD_H

#include "Carro.hpp"

namespace Carro {

  class Ford : public CarroAbstrato {

    public:
      void start();
      void drive();
      void stop();

  };

}

#endif
```

Namespace
Exemplo

Pode-se omitir o diretório e informar durante a compilação.

Ferrari.cpp

```
#include <iostream>
#include "Ferrari.hpp"

void Carro::Ferrari::start() {
  std::cout << "Starting a Ferrari\n";
}

void Carro::Ferrari::drive() {
  std::cout << "Driving a Ferrari\n";
}

void Carro::Ferrari::stop() {
  std::cout << "Stopping a Ferrari\n";
}
```

Ford.cpp

```
#include <iostream>
#include "Ford.hpp"

namespace Carro {

  void Ford::start() {
    std::cout << "Starting a Ford\n";
  }

  void Ford::drive() {
    std::cout << "Driving a Ford\n";
  }

  void Ford::stop() {
    std::cout << "Stopping a Ford\n";
  }

}
```


Namespace
Exemplo

main.cpp

```
#include "Ferrari.hpp"
#include "Ford.hpp"

int main() {

  Carro::Ferrari ferrari;
  ferrari.start();
  ferrari.drive();
  ferrari.stop();

  Carro::Ford ford;
  ford.start();
  ford.drive();
  ford.stop();

  return 0;
}
```


Namespace
Exemplo

```
CC=g++
CFLAGS=-std=c++11 -Wall
TARGET=program

BUILD_DIR    = ./build
SRC_DIR      = ./src
INCLUDE_DIR  = ./include

${BUILD_DIR}/${TARGET}: ${BUILD_DIR}/Ferrari.o ${BUILD_DIR}/Ford.o ${BUILD_DIR}/main.o
	${CC} ${CFLAGS} -o ${BUILD_DIR}/${TARGET} ${BUILD_DIR}/*.o

${BUILD_DIR}/Ferrari.o: ${INCLUDE_DIR}/automovel/Ferrari.hpp ${SRC_DIR}/automovel/Ferrari.cpp
	${CC} ${CFLAGS} -I ${INCLUDE_DIR}/automovel/ -c ${SRC_DIR}/automovel/Ferrari.cpp -o ${BUILD_DIR}/Ferrari.o

${BUILD_DIR}/Ford.o: ${INCLUDE_DIR}/automovel/Ford.hpp ${SRC_DIR}/automovel/Ford.cpp
	${CC} ${CFLAGS} -I ${INCLUDE_DIR}/automovel/ -c ${SRC_DIR}/automovel/Ford.cpp -o ${BUILD_DIR}/Ford.o

${BUILD_DIR}/main.o: ${INCLUDE_DIR}/automovel/Ferrari.hpp ${INCLUDE_DIR}/automovel/Ford.hpp ${SRC_DIR}/main.cpp
	${CC} ${CFLAGS} -I ${INCLUDE_DIR}/automovel/ -c ${SRC_DIR}/main.cpp -o ${BUILD_DIR}/main.o

# Rule for cleaning files generated during compilation.
# Call 'make clean' to use it
clean:
	rm -f ${BUILD_DIR}/*
```

https://www.rapidtables.com/code/linux/gcc/gcc-i.html

Considerações finais

- Maior reusabilidade
- Melhoria da legibilidade
- Modificações facilitadas (e mais seguras)
- Maior confiabilidade
- Aumento da produtividade