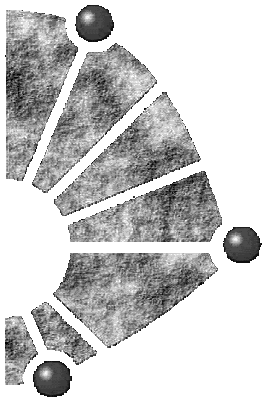
Programação Orientada a Objetos em C++



Ivan Luiz Marques Ricarte

FEEC/UNICAMP

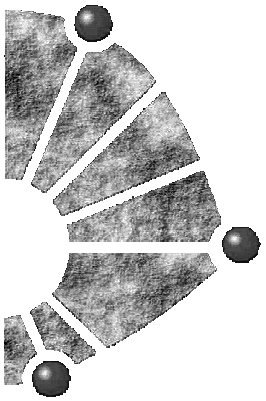
Visão geral

- A programação orientada a objetos no contexto geral do desenvolvimento de software
- Do projeto de software para a implementação em C++
- Técnicas de implementação inerentes à linguagem de programação C++

Formas de contato

•e−mail
`ricarte@fee.unicamp.br`
•URL
`http://www.dca.fee.unicamp.br/~ricarte/`
•endereço postal
`DCA/FEEC/UNICAMP`
`Caixa Postal 6101`
`13083−970 Campinas, SP`
•telefone
`(19) 3788 3771`

O desenvolvimento de software orientado a objetos



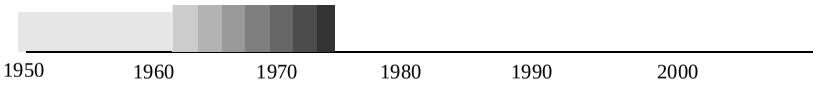
Uma breve visão da evolução da Engenharia de Sofwtare

Por que tanta ênfase nas técnicas de desenvolvimento de software?

- Sistemas computacionais onipresentes
  - software é componente essencial nesses sistemas
- Software é um produto
  - resultado do desenvolvimento entregue junto ao sistema computacional
- Software é um veículo para disponibilizar o produto

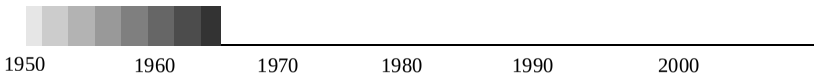
A segunda era do desenvolvimento de software

- Software como um produto
  - primeiras bibliotecas de software
- Alto custo de manutenção
  - em muitos casos, manutenção era virtualmente impossível



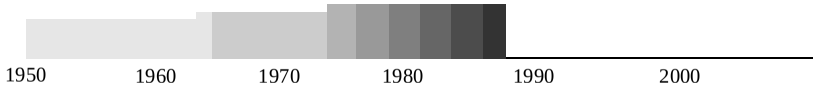
Os primórdios do desenvolvimento de software

- Desenvolvimento de software dirigido pelo desenvolvimento de hardware
  - software como um *afterthought*
  - “custom software”
  - quase nenhum método sistemático de desenvolvimento



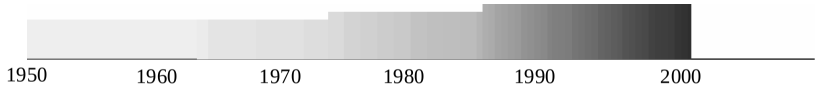
A terceira era

- Complexidade de software crescente
  - sistemas distribuídos e concorrentes
- Amplo uso de microprocessadores
  - sistemas embarcados
  - custo de hardware baixo



A era atual no desenvolvimento de software

- Sistemas decentralizados e incorporando novas tecnologias
  - Inteligência artificial, computação paralela, realidade virtual, multimídia
    - software mais caro que hardware
    - OO como tecnologia de integração



Mitos de software

- *Precisa desenvolver software?*
  - *Vamos comprar um computador da última geração*
- *O que é preciso para começar a programar?*
  - *Basta uma descrição genérica de objetivos*
- *O desenvolvimento está atrasado?*
  - *Vamos acrescentar alguns programadores a mais para agilizar e entrar em fase*

Problemas recorrentes no desenvolvimento de software

- Maior parte do software ainda é desenvolvido sob medida
  - como explorar potencial do hardware?
  - como atender às demandas por novas funcionalidades e programas?
  - como desenvolver software confiável?
  - como manter software existente?
- Os mitos de software persistem

E mais mitos

- *Houve algumas mudanças nos requisitos do projeto; algum problema?*
  - *Não, software é flexível*
- *Como saber se o projeto obteve sucesso?*
  - *Basta ver se o programa "rodou"*
- *O programa "rodou"?*
  - *Fim do trabalho do desenvolvedor do software*
- *Como certificar a qualidade do software?*
  - *Pode−se analisar agora que o programa está rodando*

Demanda pela Engenharia de Software

- *The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*

  Fritz Bauer, 1969

- *(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
- *(2) The study of approaches as in (1).*

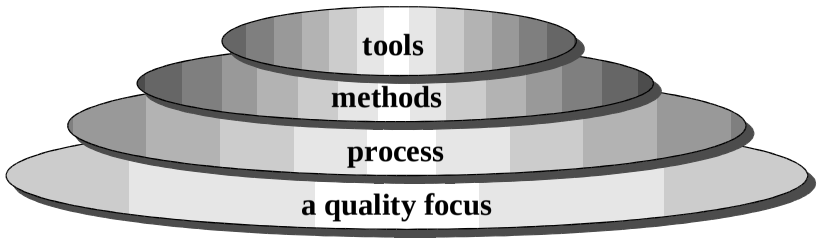
  IEEE, 1993

Camadas da Engenharia de Software

- Processos
  - estabelecem a base para o controle administrativo de projetos de software
- Métodos
  - aspectos técnicos (os “how to’s”) da construção de programas
- Ferramentas
  - apoio automatizado a processos e métodos

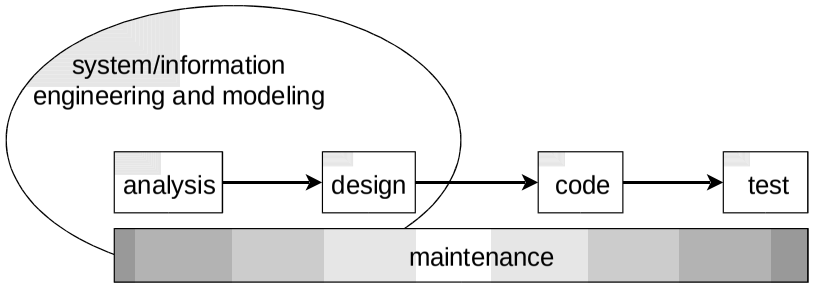
O que é a engenharia de software?

- A tecnologia que deveria ser utilizada por aqueles que desenvolvem software



Modelo de processo linear

- Modelo em etapas seqüenciais
  - t.c.c. modelo clássico ou *waterfall*



Etapas genéricas do processo: Análise

- Análise de requisitos
  - ponte entre engenharia de sistemas e o projeto de software
  - reconhecimento do problema, avaliação, síntese, modelagem, especificação e revisão
- Análise de software
  - primeira representação técnica do sistema
  - modelagem dos dados, seus atributos e relacionamentos, e das funções que os manipulam

Etapas genéricas no processo: Codificação

- Tradução do projeto de software em programas
  - uma ou mais linguagens de programação
- Incorporação de opções de implementação
  - fidelidade ao projeto, eficiência, reutilização de código, adaptação a recursos disponíveis

Etapas genéricas do processo: Projeto

- Estabelecimento das opções de desenvolvimento
  - dados, procedimentos, interfaces e arquitetura
  - deve contemplar “requisitos implícitos”
- Guia para implementador
  - oferece visão completa do software, tanto no aspecto estrutural como no comportamental

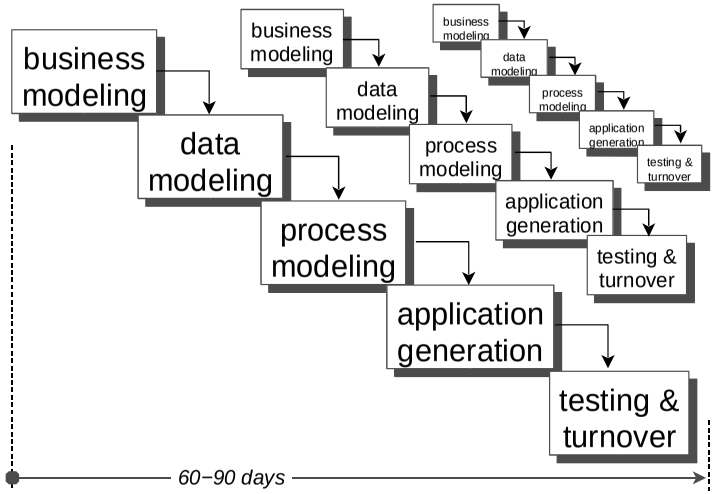
Etapas genéricas do processo: Teste

- Processo “destrutivo”
  - deve demonstrar que programa não atende aos requisitos estabelecidos
- Estratégias
  - Depuração de código
  - Teste de unidade
  - Teste de integração
  - Teste de sistema e validação

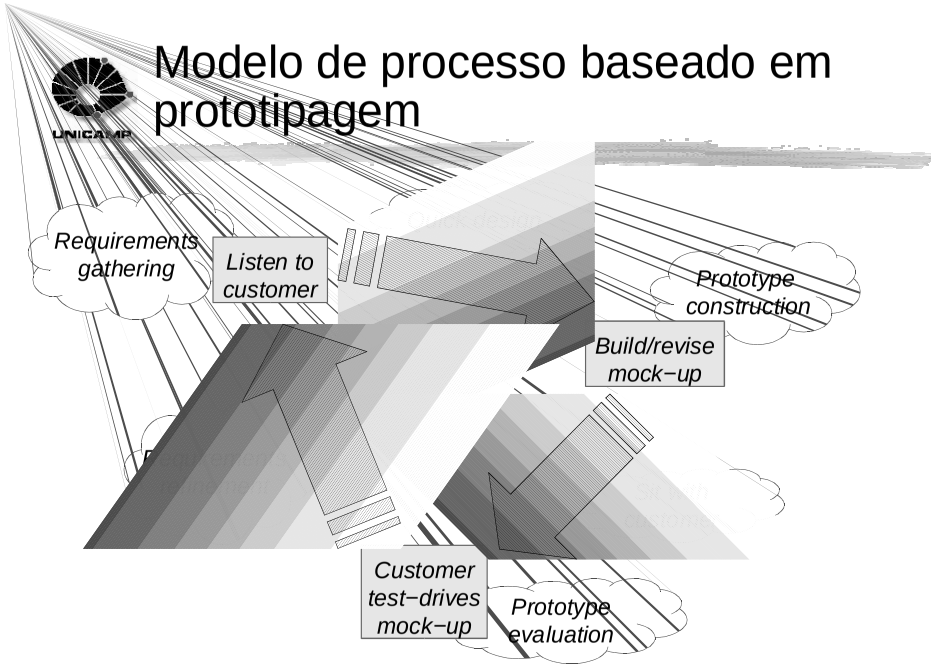
Etapas genéricas do processo: Manutenção

- Atividade de correção ou atualização de parte do software
  - em decorrência do resultado de testes ou de novas demandas colocadas para o sistema
- Requer um ciclo completo de desenvolvimento
  - análise, projeto, codificação e teste

Modelo de processo RAD



Modelo de processo baseado em prototipagem



A metodologia de desenvolvimento de software estruturado

- Modelos resultantes da análise
  - Diagramas de fluxo de dados
- Atividades no projeto
  - Estabelecimento de arquitetura do sistema
  - Seleção das estruturas de dados
- Codificação
  - Linguagem de programação estruturada

Características de um bom projeto de software estruturado

- Abstração
  - de dados, de procedimentos e de controle
  - refinamento *top−down*
- Arquitetura modular
  - solução como integração de módulos com alta coesão e baixo acoplamento
- Ocultamento da informação
  - reduz impacto de alterações e propagação de erros

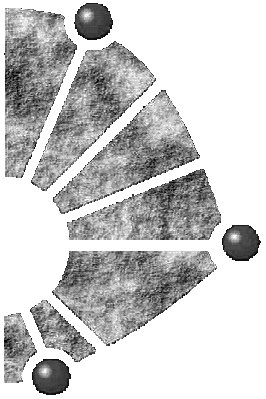
A construção de software orientado a objetos

- Uso de linguagem de programação que suporte as mesmas primitivas da análise e projeto
  - classes, objetos, herança, relações, ...
- O modelo de orientação a objetos permite levar naturalmente às boas características de software
  - mas maus hábitos de programação podem levar tudo a perder!

A metodologia de desenvolvimento de software orientado a objetos

- A mesma “linguagem” na análise e projeto
  - Análise: modelos conceituais
    - casos de uso
    - classes, atributos no domínio do problema
    - diagramas de seqüência e contratos
  - Projeto: opções de desenvolvimento
    - uso de componentes e interfaces
    - visibilidade e organização em pacotes

Programação Orientada a Objetos



Do projeto para o programa

Resultados da atividade de projeto

- Especificação detalhada do sistema
  - Diagramas, descrições funcionais e estruturais
- Projeto estruturado
  - procedimentos, módulos, interface de programação para módulos, estruturas de dados internas
- Projeto orientado a objetos
  - classes, métodos, atributos, colaborações

Evolução dos paradigmas de programação: orientada a objetos

- Foco na definição de classes
  - definição estrutural de classes
    - classes básicas, atributos, declaração de métodos
    - classes derivadas
  - definição do comportamento de classes
    - definição de métodos
- Programas de teste e programa principal
  - após definição completa de classes

Evolução dos paradigmas de programação: tradicional

- Foco nas descrições de procedimentos
  - Abordagem *top−down*
    - primeira descrição: programa principal
    - procedimentos usados no programa principal são refinados sucessivamente até que procedimentos primitivos sejam alcançados
  - Arquitetura modular
    - estrutura interna de organização
    - mecanismo para abstração

Programação orientada a objetos

- Um paradigma de programação
  - Técnica para escrever bons programas para um determinado conjunto de problemas
- Linguagem de programação orientada a objetos
  - Linguagem de programação que oferece mecanismos adequados ao estilo de programação orientada a objetos

Características de linguagens de programação orientada a objetos

- Definição de classes
  - criação de tipos definidos pelo programador
  - conjunto de atributos e métodos associados
- Herança
  - definição de classes a partir da especialização de classes existentes
- Criação e remoção de objetos
  - manipulação da região de memória definida pelo objeto

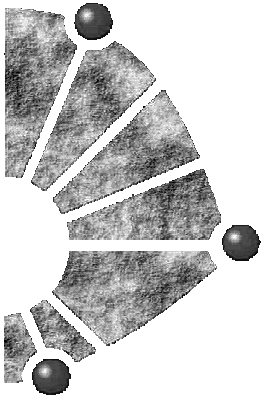
Linguagens recentes de programação orientada a objetos

- C++
  - extensão da linguagem C para incorporar mecanismos da programação orientada a objetos
    - meados da década de 1980
- Java
  - uma nova linguagem orientada a objetos usando uma sintaxe no estilo da linguagem C
    - meados da década de 1990

Primeiras linguagens de programação orientada a objetos

- Simula 67
  - aplicações em simulação
  - introduziu conceito de classes e objetos
- Smalltalk 80
  - linguagem “completamente orientada a objetos” desenvolvida na Xerox PARC
    - baixa eficiência
    - considerada uma “linguagem de brinquedo” por “desenvolvedores de verdade”

Programação orientada a objetos



Introdução à

linguagem C++

Introduction to C++

- Programming language to improve the C programming language supporting:
  - data abstraction
  - object−oriented programming
- Designed by Bjarne Stroustrup
  - AT&T Bell Labs
  - initial motivation
    - development of efficient simulation programs

C++ evolution

- C++ predecessor: C with classes (1980)
  - Stroustrup approach to support development of event−driven simulation programs from C
- C++ first public appearance
  - July 1983
  - since then, use of C++ has grown explosively
- C++ standardization effort
  - Starting 1987
  - ANSI committee, 1989−1996

C++ evolution

- C++ is mostly derived from C
  - originally a systems programming language
    - Unix operating system project
  - generated code is time and space efficient
  - derived from BCPL and B
- Concepts from many other languages
  - Simula67: classes
  - Algol68: operator overloading
  - ADA and Clu: templates, exception handling

C++: Class definition

`class` *class_name* **`{`**
`private:`
p*rivate−member−declarations*
`public:`
p*ublic−member−declarations*
`protected:`
p*rotected−member−declarations*
`};`

C++: Class definition

- Members are class attributes (variables) or class methods (functions)
  - private members can only be accessed from member methods
  - public members accessible from any function
  - protected members accessible from derived classes

C++: Class definition

- Static member
  - member of a class that is shared by all objects of the class
    - keyword **`static`**
  - static attribute member
    - same value viewed by all objects
  - static method member
    - applied to the class, not to objects

C++: Class definition

- Attribute members are defined within the scope of class definition
- Method members are declared within class definition but can be defined outside
  - method definition within class definition: inline
  - to define outside class definition: scope operator

```
returntype classname::methodname(args){body}
```

C++: Objects

- Instances of classes
  - access to class members: operator **`.`**
- can be handled statically
  - created by object declaration
  - deleted by end of scope
- can be handled dynamically
  - created by operator **`new`**
  - deleted by operator **`delete`**

C++: Objects

- Object initialization and finalization
  - constructors:
    - same name of class
    - no return value (not even void), can have arguments
    - automatic invocation when object is created
  - destructors:
    - name of the class prefixed by ~
    - no return value, no arguments
    - automatic invocation when object is destroyed

C++: Polymorphism

- Abstract classes
  - contain at least one *virtual function*
  - concrete derived classes should provide implementation for these functions
  - member functions bound dynamically, during run time, to the function that will be called
    - when function invoked through reference to object
    - in C++, reference to object: pointer

C++: Inheritance

- Inheritance

```
class derived : public base {
// only additions to derived class are specified
  ...
}
```

- Multiple inheritance

```
class derived : public base1, public base2 {
...
}
```

C++ syntax

- Minimal C++ program:

```
main( ) {  }
```

  - defines a function `main` that takes no arguments and does nothing
  - { and } express grouping in C++
    - block delimiters
- Every C++ program has a **`main`** function
  - starting point for program execution

C++ syntax

- main function can present arguments and return a value

```
int main(int argc,char *argv[]){
   return 0;
}
```

  - argc: argument count
  - argv: argument values
  - return value: to the invoking program (operating system)

C++ syntax

- Fundamental types
  - `char`
  - `short int` or simply `short`
  - `int`
  - `long int` or simply `long`

  Integral types

  - `float`
  - `double`
  - `long double`

  Floating−point numbers

  - `void`

C++ syntax

- Variables
  - declaration

```
type  var_name[, var_name];
```

  - also
    - initialization
    - **`const`** modifier
    - storage modifiers **`static`** and **`extern`**
      - alter scope and lifetime of variables

C++ syntax

- Arithmetic operators

  `+  −  *  /  %  ++  −−`
- Comparison operators

  `==  !=  <  >  <=  >=`
- Boolean connectives: **`&&  ||  !`**
- Bit operators: **`&  |  ~  ^`**
- Assignment operator: **`=`** op**`=`**
- **`sizeof`** operator

C++ syntax

- Arrays
  `char  v[10];`
  - v has elements indexed from 0 to 9
- Pointers
  `char *p;`
- Operator address−of
  `p = &v[3];`

C++ syntax

- Loops
  `while` (logical condition) {
  ...
  }
  - also
    - **`do while`** and **`for`** statements
    - **`break, goto, continue`** jump statements
- Logical condition is an integer expression

C++ syntax

- Tests
  `if` (logical condition) {
  ...
  }
  `else` {
  ...
  }
  - also
    - **`switch−case`** statement
    - ternary operator **`? :`**

C++ syntax

- Functions
  `return_type function(param_list) {`
  function body
  }
  - also
    - **`return`** statement
    - function declarations
  - argument passing
    - default: by value

C++ syntax

- Functions
  - argument passing by reference is possible
    - pointer manipulation is implicit
    - operator **&**
  - different signatures for same function name
    - function signature determines which implementation is selected
    - static (compile−time) checking

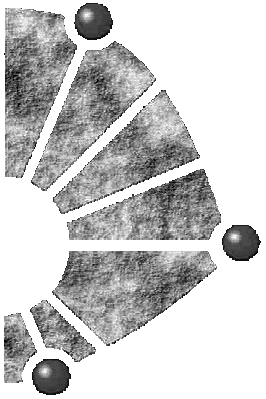
C++ programming

- Common design pitfalls
  - Ignore classes, using C subset only
  - Ignore derived classes and virtual functions
    - use data abstraction subset
  - Ignore static type checking
    - design constraining implementers to dynamic type checking only

C++ programming

- It is possible to use C++ and do exclusively procedural programming
  - C++ can be used as a dialect of C
- Development of good object−oriented programs in C++ depends on programming discipline
  - supported by application of object−oriented practices in analysis and design

Programação orientada a objetos com C++



Trabalhando com objetos

A criação de objetos

- Criação de um objeto de uma classe dá-se por meio da invocação de um de seus métodos construtores
  - construtor default
    - invocado sem argumentos
  - construtor de cópia
    - invocado com um argumento da mesma classe
  - construtor com argumentos
    - invocado em outros casos

Exemplos de construtores default

```
class Xyz {
...
}

main() {
   Xyz o1;
   ...
}
```

```
class Xyz {
...
public:
   Xyz(int a);
}
main() {
   Xyz o1;
   ...
}
```

erro!

```
class Xyz {
...
public:
  Xyz(int a=0);
}
main() {
   Xyz o1;
   ...
}
```

Construtor default

- Automático
  - construtor sem argumentos é disponibilizado pelo compilador se nenhum outro construtor for definido
- Definido pelo programador
  - sem argumentos
  - com argumentos, desde que todos possam receber algum valor por default

Construtor de cópia

- Permite criação de objetos a partir de outro objeto existente
  - recebe como argumento uma referência para um objeto da mesma classe

```
class Xyz {
   ...
public:
   Xyz(Xyz& a);
}
```

Construtor de cópia automático

- Construtor de cópia é automaticamente definido pelo compilador
  - quando programador não define explicitamente um construtor de cópia
- Comportamento do construtor automático
  - cria novo objeto da classe
  - faz cópia membro a membro

Construção de objetos

- Estaticamente
  - declaração de objeto
  - escopo válido até o fim do bloco onde ocorre declaração
- Dinamicamente
  - declaração de ponteiro para objeto
  - criado com operador new
  - válido até invocação do operador delete para o ponteiro

Construtor de cópia pelo programador

- Situações onde é necessário definir explicitamente um construtor de cópia
  - quando parte do objeto faz uso de recursos externos aos seus membros
    - áreas alocadas de memória, arquivos
- Opções de cópia com recursos externos
  - cópia rasa
    - com compartilhamento
  - cópia profunda
    - com alocação de novos recursos

Exemplo de construção de objetos

```
#include <iostream.h>
class Xyz {
  int x;  int y;
public:
  Xyz(int a=0, int b=0);
  Xyz(Xyz& a);
};
Xyz::Xyz(int a, int b) {
  x = a;  y = b;
  cout << "Xyz criado: (" << x << "," << y << ")" << endl;
}
Xyz::Xyz(Xyz& a) {
  x = a.x;  y = a.y;
  cout << "Xyz copiado: (" << x << "," << y << ")" << endl;
}
int main(int argc, char *argv[]) {
  Xyz o1;
  Xyz o2(1,1);
  Xyz o3 = o2;
  Xyz o4 = Xyz(2,2);
  Xyz o5 = 1;
  Xyz* o6 = new Xyz(3,3);
  return 0;
}
```

Resultado da execução para construção de objetos

- Xyz criado: (0,0)
- Xyz criado: (1,1)
- Xyz copiado: (1,1)
- Xyz criado: (2,2)
- Xyz criado: (1,0)
- Xyz criado: (3,3)

Operando com arranjos de objetos

- Para a criação de a1 e a4, construtor sem argumentos é utilizado
  - deve estar presente
- A remoção de arranjos alocados dinamicamente deve ser explícita
  - delete [] a4;
  - de outro modo, apenas o primeiro elemento é liberado
    - seria o comportamento padrão pelo fim do escopo

Criação de arranjos de objetos

- Xyz a1[4];
  - cria arranjo com quatro objetos (0,0)
- Xyz a2[] = {1, 2};
  - cria arranjo com dois objetos, (1,0) e (2,0)
- Xyz a3[4] = {Xyz(1,1), Xyz(2,2)};
  - cria arranjo com (1,1), (2,2), (0,0) e (0,0)
- Xyz* a4 = new Xyz[4];
  - cria arranjo com quatro objetos (0,0)

O operador new

- Operador global e static
  - Obtém espaço da área livre
- Pode ser redefinido na classe
  - void *operator new(size_t s);
- Pode incluir parâmetros
  - void *operator new(size_t s, params);
  - Uso: Xyz* x = new (args) Xyz(args_cons);
- Pode ser sobrecarregado
  - para usar o original: ::new

Lista de inicializadores

- Complementa informação para a construção do objeto

  Classe::Classe(params) : lista_inicializadores {...}
- Possíveis usos:
  - iniciar valores de membros constantes
  - iniciar objetos membros com construtores diferentes do default
  - invocar construtores diferentes do default nas superclasses

Destruição de objetos

- Quando estado de objeto é descrito exclusivamente por seus membros “locais”, comportamento padrão de remoção funciona bem
  - liberar a área de memória ocupada
- Quando outros recursos externos estão alocados ao objeto, uso de destrutores é necessário

Seqüência de construção

- Quando um objeto de classe T é criado
  - 1. construtores das classes bases são chamados
    - pela ordem de declaração, se não alterada na lista de inicializadores
  - 2. se classe T contém membros que sãso objetos, estes são criados
    - usando construtor default, se não especificado de outra forma na lista de inicializadores
  - 3. corpo do construtor de T é executado

Destrutores

- Método destrutor para classe T é T::~T()
  - sem argumentos
  - não pode ser sobrecarregado
  - objetos criados estaticamente
    - invocado automaticamente ao fim do escopo
  - objetos criados dinamicamente
    - invocado automaticamente pelo operador **`delete`**
- Exemplo de uso
  - remoção de área alocada:
    - cópia rasa vs. profunda

Invocação explícita de destrutor

- É um método static da classe:

```
Abc* pa;
...
pa -> Abc::~Abc();
```

  - realiza um “clean−up” sem remover o objeto
  - neste caso, destrutor precisa ter sido declarado explicitamente
    - apenas um destrutor pode ser definido
    - assinatura é fixa, não permite sobrecarga

Cópia de objetos

- Duas maneiras de copiar objetos
  - por atribuição
    - uso do operador de atribuição =
  - por inicialização
    - na inicialização de objetos
      - incluindo declaração com valor inicial
    - na passagem de argumentos por valor
      - portanto não deve existir construtor T(T) para classe T
    - como retorno de funções e métodos

Seqüência de destruição

- Quando um objeto da classe T é destruído
  - 1. o destrutor da classe T é invocado
  - 2. os destrutores dos objetos membros de T são invocados
  - 3. os destrutores das classes bases são invocados
- Processo inverso ao da criação de objetos

Operador de atribuição

- Comportamento padrão
  - cópia membro a membro
  - pré−definido
  - pode não ser adequado em alguns casos
    - membros que são ponteiros para áreas alocadas
- Sobrecarga do operador =

```
Abc&  Abc::operator =(Abc& rhs)
```

  - operador pode ser desabilitado sobrecarregando−o como **`private`**

Sobrecarga de operadores: nem tudo é permitido

- Não é possível inventar novos operadores
  - restrito aos existentes em C++
- Não é possível redefinir alguns operadores
  - . .* :: ? :
  - # ##
- Não é possível mudar associatividade ou precedência de operadores
- Não é possível mudar número de operandos de operadores

Exemplo: sobrecarga do operador +

```
class Xyz {
  int x;  int y;
public:
  ...
  Xyz operator +(Xyz a);
};
...
Xyz Xyz::operator +(Xyz a) {
  Xyz z;
  z.x = x + a.x;
  z.y = y + a.y;
  return z;
}
int main() {
  Xyz o1(2,2);
  Xyz o2(1,1);
  Xyz o3 = o1 + o2;
  Xyz o4 = o2 + 1;
  //  Xyz o5 = 1 + o2;  <=== Situação de erro de compilação
}
```

Sobrecarga de operadores: casos especiais

- Operadores incremento (++) e decremento (−−)
  - para diferenciar implementação pré−fixada da pós−fixada, nesta utiliza−se um parâmetro extra do tipo int
  - para classe T,
    - T operator ++(); // pré−fixada
    - T operator ++(int); // pós−fixada

Resultado da execução com sobrecarga do operador +

- Xyz criado: (2,2)
- Xyz criado: (1,1)
- Xyz copiado: (1,1)
- Xyz criado: (0,0)
- Xyz copiado: (3,3)
- Xyz criado: (1,0)
- Xyz criado: (0,0)
- Xyz copiado: (2,1)

Forma funcional equivalente ao uso do operador +

- Xyz o3 = o1 + o2;
  - o3 = o1.operator+(o2);
    - todos os argumentos de acordo com assinatura
- Xyz o4 = o2 + 1;
  - o4 = o2.operator+(1);
    - sabe como criar Xyz a partir de um inteiro
- Xyz o5 = 1 + o2;
  - o5 = 1.operator+(o2);
    - não pode aplicar operador a um valor inteiro...

Exemplos de amizade

```
class X {
   friend void f();
}

void f() {
   ...
}
```

métodos com direito de acesso aos membros internos de X

```
class Y;
class X {
   friend void Y::f();
}
class Y {
void f();
}
void Y::f() {
   ...
}
```

```
class Y;
class X {
   friend void Y;
}
class Y {
...
}
```

todos os métodos de Y têm direito de acesso aos membros internos de X

Friends

- Funções que são amigas de uma classe recebem direito de acesso aos membros internos (private e protected) de objetos dessa classe
  - Quando uma classe é declarada como amiga dentro de outra, todos os métodos da classe declarada recebem esse direito

Características de funções e operadores amigos

- São globais
  - podem pertencer a mais de uma classe
- Não têm o ponteiro de auto−referência (this)
  - não fazem referência a um objeto específico
  - não têm o parâmetro implícito
- Se declaração está na parte pública ou privativa da definição da classe é irrelevante

Uso de funções/operadores amigos: classes acopladas

- Quando acesso a membros internos de mais de uma classe é desejado ou necessário
  - operação conjunta das classes está fortemente relacionada
- Exemplos
  - Matriz e Vetor (aplicação em álgebra linear)
  - Container e Iterator (coleções de objetos)

Mecanismos de conversão de tipos

- Conversão pode ocorrer por meio de construtores e por meio de operadores

```
class Xyz {
  ...
public:
   Xyz (OutroTipo ot);       // de OutroTipo para Xyz
   operator OutroTipo( );    // de Xyz para OutroTipo
}
...
Xyz x;  OutroTipo w;
w = OutroTipo(x);      // OK
w = (OutroTipo) x;     // Também OK
w = x;                 // OK, conversão implícita
```

Uso de funções/operadores amigos: uso de conversão

- Função amiga não é aplicada a um objeto
  - recebe objeto como argumento
- Pode ser “aplicada” tanto ao objeto da classe especificada no parâmetro como a qualquer instância de outro tipo que possa ser convertido para ela
  - argumentos de funções e operadores amigos são tratados simetricamente

Exemplo de uso de operador amigo: +

```
class Xyz {
  ...
public:
friend Xyz operator +(Xyz a, Xyz b);
   ...
};
Xyz operator +(Xyz a, Xyz b) {
  Xyz z;
  z.x = a.x + b.x;
  z.y = a.y + b.y;
  return z;
}
int main() {
  Xyz o1(2,2);
  Xyz o2(1,1);
  Xyz o3 = o1 + o2;
  Xyz o4 = o2 + 1;
  Xyz o5 = 1 + o2;  // Agora OK
  ...
}
```

Uso de funções/operadores amigos: extensão streams

- Forma de uso desejada dos operadores << e >>

```
Xyz x;
cin >> x;
cout << x;
```

  - não devem ser membros de ostream e istream
    - não seria possível prever todas as situações
  - não podem ser membros de Xyz
    - lado esquerdo não é objeto Xyz

Quando não usar friend?

- Quando não houver nenhum motivo que justifique o uso de função amiga, dê preferência ao método
  - reduz uso do espaço global de nomes
- Se não há conversão definida
- Quando operador requer um lado esquerdo que seja do tipo de objeto
  - = , += , −=, ..., ++ , −−

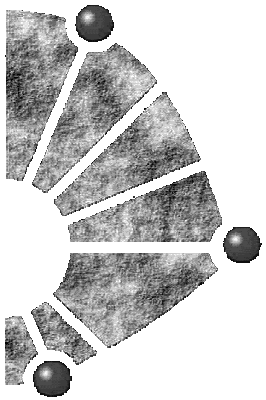
Solução para extensão de streams

```
class Xyz {
...
   friend ostream& operator<<(ostream&, Xyz& a);
   friend istream& operator>>(istream&, Xyz& a);
}
ostream& operator<<(ostream& os, Xyz& a) {
os << “(“  << a.x << “”,””  << a.y << “”)”;
return os;
istream& operator>>(istream& os, Xyz& a) {
is >> a.x >> a.y;
return is;
```

Quando usar friend?

- Duas ou mais classes fortemente acopladas
  - praticamente impossível pensar que uma seria independente da outra
- Operador pode necessitar da conversão implícita para qualquer um de seus operandos
  - +, − , *, ...

Programação Orientada a Objetos com C++



Reaproveitando funcionalidades definidas em outras classes

Quando usar herança?

- Objetos da nova classe constituem um subconjunto dos objetos da classe base
  - associação do tipo “é−um”
  - Princípio da substituição de Liskov
- Relacionamento hierárquico é estabelecido entre as classes
  - classe base: funcionalidades genéricas
  - classe derivada: funcionalidades especializadas

Projeto com reuso de classes

- Mecanismos para definir novas classes usando classes existentes:
  - Derivação de classes
    - implementação do conceito de herança
  - Composição de classes
    - implementação do conceito de associações entre classes

Estratégias de herança

- Extensão
  - classe derivada acrescenta atributos e funcionalidades à definição da classe base
- Especificação
  - classe derivada oferece implementação a funcionalidades declaradas na classe base
- Herança polimórfica
  - classe derivada implementa especificação e reaproveita funcionalidades da classe base

Mecanismos de herança em C++

- Especificação de classe derivada:

  class Derivada : especificador_acesso Base {
  ...
  };

  - objetos de Derivada podem manipular diretamente membros public e protected definidos em Base
    - mas não têm acesso a membros private

Exemplo de classe Base privativa

```
class Base {
  int a;
protected:
  int b;
public:
  int c;
};
```

```
class Deriv1 : Base {
public:
  void fd();
};
void Deriv1::fd() {
  ...
}
```

private implícito

Base::a is not accessible in function Deriv1::fd()

O especificador de acesso na derivação de classes

- private (default)
  - todos os membros public e protected da classe Base tornam−se membros private de Derivada
- protected
  - todos os membros public e protected da classe Base tornam−se membros protected de Derivada
- public
  - membros public e protected da classe Base mantêm visibilidade na classe Derivada

Restrições em funções externas à hierarquia Base−>Derivada

```
int main() {
  Base b1;
  Deriv1 d1;
  b1.c++;
  d1.c++;
  d1.fd();
  Base* bp = &d1;
  bp−>c++;
}
```

Base::c is not accessible in function main()

Cannot convert ‘Deriv1 *’ to ‘Base *’ in function main()

Reajuste de permissões de acesso a membros

- Visibilidade da classe base pode ser individualmente restaurada na classe derivada

```
class Deriv1 : private Base {
protected:
Base::b;
public:
  void fd();
};
```

O que era público na base continua público na derivada

```
int main() {
  Deriv2 d1;
  d1.c++;
  d1.fd();
  Base* bp = &d1;
  bp->c++;
}
```

Agora OK.

Também OK.

Exemplo de classe Base pública

```
class Base {
  int a;
protected:
  int b;
public:
  int c;
};
```

```
class Deriv2 : public
   Base {
public:
  void fd();
};
void Deriv2::fd() {
   ...
}
```

Base::a is not accessible in function Deriv1::fd()

O que se herda da classe base?

- A classe derivada recebe
  - Todos os membros de dados
    - mesmo que não tenha acesso a eles
  - Todos os métodos (funções membros)
- Mas não recebe de herança
  - Construtores
  - Destrutores

Objetos derivados e construtores

- Construção de objeto começa pela parte derivada da classe base
  - Construtor da classe derivada sempre invoca construtor da classe base
  - Construtor *default* (sem argumentos) é invocado automaticamente
  - Caso não exista construtor default na classe base, invocação explícita deve estar na lista de inicializadores do construtor derivado

Especificando a construção da parte base

- Incluir construtor default na classe base
  - se tiver acesso à classe base
  - se fizer sentido
- Referenciar construtor válido na lista de inicializadores

```
Deriv1::Deriv1( ) : Base(0) {
...
}
```

A busca pelo construtor *default*

```
class Base {
public:  Base(int i);
};
Base::Base(int i) { ... }
class Deriv1 : public Base {
 ...
};
int main() {
  Deriv1 d1;
}
```

Cannot find default constructor to initialize base class 'Base' in function Deriv1::Deriv1()

Definindo o construtor de cópia

- Construtor de cópia automático
  - invoca construtor de cópia da classe base
  - complementa cópia com parte específica
- Construtor de cópia definido pelo programador
  - responsabilidade de invocar cópia da base

```
Deriv::Deriv(Deriv& d) : Base(d) {
...
}
```

Atribuição entre objetos de classes bases e derivadas

- De classe derivada para classe base
  - ocorre sem problemas
  - descarta parte especializada
- De classe base para derivada
  - não é permitida (parte indefinida)

```
int main() {
  Deriv d1; Base b1;
  b1 = d1;
  d1 = b1;
}
```

Ok

Cannot convert 'Base' to 'Deriv' in function main()

Usando operador de atribuição automático

```
class Base {
public:
  Base& operator=(Base& b);
};
Base& Base::operator=(Base& b) {
  cout << "Atribuicao base" << endl;
  return b;
}
class Deriv1 : public Base {...};
int main() {
  Deriv1 d1, d2;
  d2 = d1;
}
```

Atribuição base é invocada

O operador de atribuição em classes derivadas

- Semântica para operador definido automaticamente:
  - invoca operador de atribuição para classe base
  - executa atribuição membro a membro para a parte específica da classe derivada
- Semântica para operador é redefinido:
  - responsabilidade do programador

Usando operador de atribuição especificado

```
class Base { public:  Base& operator=(Base& b); };
Base& Base::operator=(Base& b) { ... }
class Deriv : public Base {
public:  Deriv& operator=(Deriv& d); };
Deriv& Deriv::operator=(Deriv& d) {
  cout << "Atribuicao da parte derivada" << endl;
  return d;
}
int main() {
  Deriv d1, d2;
  d2 = d1;
}
```

Atribuição base não é invocada

Usando todos operadores de atribuição especificados

```
class Base { ... };
Base& Base::operator=(Base& b) { ... }
class Deriv : public Base { ... };
Deriv& Deriv::operator=(Deriv& d) {
  (Base&) (*this) = d;
  cout << "Atribuicao da parte derivada" << endl;
  return d;
}
int main() {
  Deriv d1, d2;
  d2 = d1;
}
```

Atribuição base é invocada e, depois, atribuição derivada

Cuidados com uso de ponteiros

- Porquê dos ponteiros
  - permitem programação mais flexível
    - definição do objeto manipulado apenas no momento da execução
  - permitem aplicação do princípio da substituição de Liskov
    - onde houver ponteiro para classe base, objeto de qualquer classe derivada pode ser referenciado

Objetos derivados e destrutores

- O processo de destruição de objeto ocorre na ordem inversa ao processo de criação

```
Base::Base(int i) { cout << "Construtor Base" << endl; }
Base::~Base() { cout << "Destrutor Base" << endl; }
Deriv1::Deriv1() : Base(0) { cout << "Construtor Deriv1" << endl; }
Deriv1::~Deriv1() { cout << "Destrutor Deriv1" << endl; }
int main() {
  Deriv1 d1;
  ...
}
```

Construtor Base
Construtor Deriv1

Destrutor Deriv1
Destrutor Base

Destruição de objetos derivados e ponteiros

- Destrutores não são redefinidos:

```
int main() {
  Base* pb = new Deriv1;
  ...
  delete pb;
}
```

Construtor Base
Construtor Deriv1

Destrutor Base

Destruição de objetos derivados e ponteiros

- Se destrutor base for declarado **virtual**

```
class Base {   ...
   virtual ~Base();
}
int main() {
  Base* pb = new Deriv1;
   ...
  delete pb;
}
```

Construtor Base
Construtor Deriv1

Destrutor Deriv1
Destrutor Base

Métodos de mesmo nome na seção pública

- Definição derivada esconde original mesmo que assinatura seja diferente

```
class Base {public:  void f(); };
void Base::f() { ... }
class Deriv : public Base {public:  void f(int x); };
void Deriv::f(int x) { ... }
int main() {
  Deriv d1;
  d1.f();
  d1.f(1);
}
```

Erro de compilação:
Too few parameters in call to
Deriv1::f(int) in function main()

Definição de membros com mesmo nome

- Membro de classe derivada com mesmo nome de membro da classe base "esconde" o membro original

```
class Base {public:  void f(); };
void Base::f() {  cout << "Base::f()" << endl; }
class Deriv : public Base {public:  void f(); };
void Deriv::f() {  cout << "Deriv::f()" << endl; }
int main() {
  Base b1;    b1.f();
  Deriv d1;   d1.f();
}
```

Base::f( )

Deriv::f( )

Acesso a membros de mesmo nome

- O operador de escopo permite acesso às definições originais

```
...
void Deriv::f() {
  Base::f();
  cout << "Deriv::f()" << endl;
}
int main() {
  Deriv d1;
  d1.f();
}
```

Referência à definição original

Base::f()
Deriv::f()

Acesso a membros de mesmo nome por ponteiros

- No caso geral, o tipo do ponteiro é que predomina

```
...
int main() {
  Base *b1, *b2;
  b1 = new Base;
  b1 −> f( );
  b2 = new Deriv;
  b2 −> f( );
}
```

Base::f()

Base::f()

Conversão entre classes bases e derivadas

- Ponteiro para objeto de classe derivada é automaticamente convertido para ponteiro de classe base
  - apenas membros da classe base são referenciáveis através do ponteiro
  - para métodos declarados como virtuais, se um método redefinido com mesma assinatura estiver presente na classe derivada então será usado. Senão, usa método da classe base

Polimorfismo: usando funções membros virtuais

- Quando método é virtual, vale a definição para o tipo efetivo do objeto apontado

```
class Base {
public:
  virtual void f();
};
int main() {
  Base *b1, *b2;
  b1 = new Base;
  b1−>f();
  b2 = new Deriv;
  b2−>f();
}
```

Base::f()

Deriv::f()

Funções virtuais puras

- Funções virtuais que devem obrigatoriamente ser redefinidas em classes derivadas
  - não contém nenhuma implementação

```
class Base {
public:
   virtual void f( ) = 0;
};
```

Classes abstratas

- Contêm funções virtuais puras
- Só podem ser utilizadas como base para a definição de outras classes
  - não é possível criar objetos de classe abstrata

```
class Abst {
public:
  virtual void f() = 0;
};
int main() {
  Abst a1;
}
```

```
Cannot create instance of abstract
class 'Abst' in function main()
Class 'Abst' is abstract because of
'Abst::f( ) = 0'
```

Herança múltipla com base repetida

```
class Base { ... };
class Base1 : public Base { ... };
class Base2 : public Base { ... };
class Deriv : public Base1, public Base2 {
...
};
int main() {
  Deriv d1;
  ...
}
```

```
construtor de Base
construtor de Base1
construtor de Base
construtor de Base2
construtor de Deriv
```

```
destrutor de Deriv
destrutor de Base2
destrutor de Base
destrutor de Base1
destrutor de Base
```

Herança múltipla

- Mesmos princípios da herança simples

```
class Base1 { ... };
class Base2 { ... };
class Deriv : public Base1, public Base2 {
...
};
int main() {
  Deriv d1;
  ...
}
```

```
construtor de Base1
construtor de Base2
construtor de Deriv
```

```
destrutor de Deriv
destrutor de Base2
destrutor de Base1
```

Herança múltipla com base compartilhada

```
class Base { ... };
class Base1 : public virtual Base { ... };
class Base2 : public virtual Base { ... };
class Deriv : public Base1, public Base2 {
...
};
int main() {
  Deriv d1;
  ...
}
```

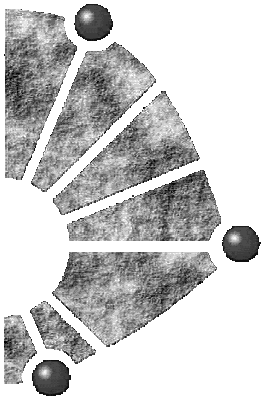
```
construtor de Base
construtor de Base1
construtor de Base2
construtor de Deriv
```

```
destrutor de Deriv
destrutor de Base2
destrutor de Base1
destrutor de Base
```

Quando não usar herança

- Hierarquia de classes contém contração
  - redução de funcionalidades nas classes derivadas
- Relação entre as classes pode ser melhor descrita por composição
  - “é parte de”, “tem um”

Programação Orientada a Objetos com C++



Tratamento de exceções

Usando composição em C++

```
class Part1 { ... };
class Part2 { ... };
class Comp {
  ...
  Part1*  p1;
  Part2*  p2;
public:
  ...
};
```

Comp não oferece os mesmos métodos que Part1 e Part2

Mas os métodos de Comp têm acesso a essas funcionalidades através dessas referências, possivelmente inicializadas na construção do objeto Comp

Se Part1 e Part2 declararem Comp como classe amiga, métodos de Comp terão acesso às suas estruturas internas

Mecanismo para manipulação de erros

- Motivação
  - separar código para tratamento de erros do “código normal”
  - evitar verificação desnecessária por condições de erro
  - desatrelar ocorrência do erro de seu tratamento

O modelo padrão de exceções

- Modelo baseado em dois eventos síncronos
  - sinalização da ocorrência da situação de exceção
    - throwing an exception
  - resposta à situação de exceção sinalizada
    - catching an exception
- Ocorrência do evento de sinalização transfere controle do ponto de execução para um tratador de exceções previamente cadastrado

Aplicando uma exceção

- Uso do comando throw

  throw *expressão;*
  - Comando executado quando a situação de erro é detectada
    - Encerra a execução do código corrente
    - Similar a um comando return

Por que exceções?

- Código pode encontrar uma condição que não pode suportar
  - Retorna a algum código que o chamou e sabe como tratar o erro.
- Eventos ocorrem em partes distintas do código
  - Podem até mesmo ter sido definidos por programadores distintos

Identificando a exceção ocorrida

- O tipo resultante da expressão throw identifica a exceção
  - Objeto estático temporário do tipo da expressão é criado
  - Seu conteúdo é inicializado para o valor da expressão
- Objeto estático: garante que mesmo erros associados à manipulação da área livre podem ser tratados

Reconhecendo a ocorrência da exceção

- Instrução try−catch

```
try {
   ... // código que pode gerar a exceção
   ... // em algum método utilizado
}
catch (tipo_de_exceção e) {
   ... // código de tratamento para esse
   ... // tipo de exceção
}
```

Múltiplos catches

```
try {
  ...
}
catch (tipo1 e1) {
  ...
}
catch (tipo2 e2) {
  ...
}
catch (...) {
...
}
```

Catch genérico

Exemplo de geração e captura de exceção

- Aplicação

```
...
 throw “Socorro!”;
...
```

- Captura e tratamento

```
try {
// ...
}
catch(char *p) {
// tratamento
}
```

Efeito da exceção

- Quando exceção é gerada, controle de execução passa para o bloco try−catch mais recentemente definido
  - A pilha é "esvaziada"
    - Destrutores são invocados para todos os objetos automáticos introduzidos desde a introdução do bloco try
    - Mesmo para objetos parcialmente construídos o compilador deve tratar corretamente a destruição parcial

Tratando a exceção

- Ativa código do catch correspondente ao tipo da exceção gerada
  - Correspondente:
    - Catch e throw têm mesmo tipo
    - Catch tem classe que é base para tipo de throw
    - Catch é ponteiro e throw gera ponteiro que pode ser convertido para o ponteiro capturado por uma conversão padrão de ponteiros

Continuando a execução

- Após código do tratamento, execução continua após o bloco try
  - Não há retorno para ponto de ocorrência do erro
- Se execução das instruções no corpo do bloco try não causa a geração de nenhuma condição de exceção, blocos catches são ignorados

Definindo múltiplos catches

- Os tipos de tratadores são testados na ordem de aparecimento
  - Os mais genéricos devem aparecer após casos especializados
  - A declaração genérica (...), se presente, deve ser a última
  - Se não houver nenhuma combinação, propaga exceção
    - Equivale a ter "catch(...) { throw; }"

Especificação de exceções

- Indicação presente na declaração e definição de funções
  - Que exceções podem ocorrer no corpo da função

```
tipo func(param)  throw (lista_tipos_exceções);

tipo func(param) throw (lista_tipos_exceções) {
   ... // código que pode gerar ou propagar
   ... // as exceções indicadas
}
```

Unexpected

- Função unexpected() é invocada quando uma função gera uma exceção não especificada
  - Efeito default de unexpected(): invocar terminate()
  - Efeito default de terminate(): invocar abort()
- Reflete um erro sério de projeto

Propagação de exceções

- No tratamento de uma exceção, o código de manipulação pode optar por repassar a exceção que o ativou:

```
...
throw;
...
```

  - encerra execução do manipulador e de seu correspondente bloco try−catch
    - repassa controle ao próximo bloco try−catch

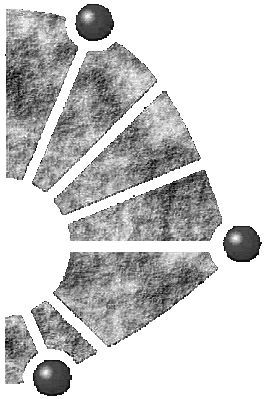
Variantes de especificação

- Função sem lista de especificação pode aplicar qualquer exceção
- Função com lista de especificação vazia, throw(), não pode aplicar exceções
- Função com lista de especificação contendo a classe B pode gerar exceções associadas a objetos de qualquer classe derivada publicamente de B

Comentários sobre exceções

- Exceções são objetos
  - podem carregar informação extra sobre a condição de erro além da simples sinalização de sua ocorrência
  - podem ser agrupadas em hierarquias de exceções
    - permite tratamento genérico para um grupo de exceções de mesma raiz

Programação Orientada a Objetos com C++



Streams

Saída

- Classe ostream
  - Sobrecarrega operador <<
    - inserção ou “put to”
    - Definido na classe base com lado direito correspondente a todos os tipos básicos da linguagem
    - qualquer outro tipo: deve ser definido como apresentar valores

Classes de entrada e saída

- Funcionalidades que não fazem parte do núcleo da linguagem
  - Definidas através do arquivo de cabeçalho iostream.h
  - declara os objetos associados aos dispositivos padrões de entrada e saída
    - cin − entrada padrão
    - cout − saída padrão
    - cerr −saída padrão de erros (sem buffer)
    - clog − saída padrão de erros (com buffer)

Entrada

- Classe istream
  - Sobrecarrega operador >>
    - extração ou “get from”
    - Definido na classe base com lado direito correspondente a todos os tipos básicos da linguagem
      - qualquer outro tipo: deve ser definido como obter valores
    - por default, ignora espaços em branco
      - se não quiser ignorar, usar métodos da classe
        - get(), getline(), read()

Manipuladores de streams

- ostream& flush(ostream&)
  - efetiva qualquer pendência nos buffers de saída
- ostream& endl(ostream&)
  - escreve nova linha e *flush*
- setw(int)
  - define máxima quantidade de elementos na leitura
    - char name[MAX];
    - cin >> setw(MAX−1) >> name;

Formatação

- métodos da classe ios
  - afetam próxima operação apenas
  - width(int w) e int width()
    - largura de campo de apresentação
  - fill(char) e char fill()
    - caráter de preenchimento da largura
  - precision(int) e int precision()
    - precisão de apresentação de valores reais

Estados de streams

- Definidos na classe base ios
  - comum a istream e ostream
- Métodos
  - int eof()
    - fim de arquivo
  - int fail()
    - próxima operação falhará
  - int bad()
    - stream corrompido

Formatação

- métodos da classe ios
  - flags(long) e long(flags)
    - flags para controle de formato:
      - left, right
      - dec, oct, hex
      - showbase
      - uppercase
      - scientific, fixed
    - associados a manipuladores
      - cout << hex << 27 << endl;

Arquivos de leitura

- Declarações em fstream.h
  - arquivo seqüencial de entrada: ifstream
    - construtor recebe um argumento char* (nome):

```
class ifstream : public fstreambase, public istream {
  ifstream(const  char *nome, int mode=ios::in, int
    prot=0664) : fstreambase(nome, mode, prot);
...
}
```

  - modos alternativos de abertura
    - segundo argumento com valores de constantes definidas em ios (ios::binary)

Arquivos de entrada e saída

- Classe fstream
  - Derivada de fstreambase e de iostream
- Todas as classes incorporam um método open() com a mesma assinatura do construtor
  - Método close() definido na classe base (virtual) ios

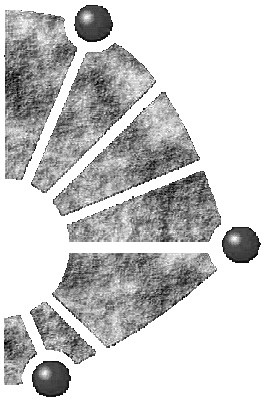
Arquivos de saída

- Arquivo seqüencial de saída: ofstream
  - construtor recebe um argumento char* (nome)

```
class ofstream : public fstreambase, public ostream {
  ofstream(const  char *nome, int mode=ios::out, int
    prot=0664) : fstreambase(nome, mode, prot);
...
}
```

  - segundo argumento com valores de constantes definidas em ios
    - ios::binary, ios::nocreate, ios::ate, ios::trunc

Programação orientada a objetos com C++



Templates

O que é um template?

- Mecanismo específico de C++ para definição de classes e algoritmos que manipulem qualquer tipo
  - originalmente, baseado em macros e pré−processamento
  - posteriormente, incorporado ao compilador de C++
- Reuso de código−fonte
  - classes ou funções

Motivação

- Considere a definição de uma classe ArrayInt que verifica se índice de acesso está entre limites válidos
  - internamente mantém um arranjo de inteiros
  - tamanho do arranjo definido no construtor
  - método size() retorna tamanho do arranjo
  - operador de indexação [ ] sobrecarregado

Em outras linguagens orientadas a objetos...

- Hierarquia de classes com raiz comum (Object)
  - Smalltalk, Java
- Estruturas de dados e algoritmos genéricos manipulam objetos da classe raiz
  - uso de polimorfismo
  - manipulação do tipo efetivo do objeto em tempo de execução

Classe ArrayInt

```
class ArrayInt {
  int* A;
  const int limite;
public:
  ArrayInt(int n=100);
  int& operator[](int index);
  int size() { return limite; }
};
```

Métodos de ArrayInt

```
ArrayInt::ArrayInt(int n) : limite(n) {
  A = new int[n];
}
int& ArrayInt::operator[](int index) {
  if (index < 0 || index >= limite)
    throw new
   IndexOutOfBounds(index,limite);
  return A[index];
}
```

Uso de ArrayInt

```
int main() {
   ArrayInt ia(20);
   try {
      for (int i=0; i<ia.size(); ++i)  ia[i] = i+1;
      cout << "ia[5] = " << ia[5] << endl;
      cout << ”ia[35] = " << ia[35] << endl;
   }
   catch(IndexOutOfBounds* iof) {
     cout << ”Acesso a posicao " << iof−>pos() <<
            " em array de "<< iof−>max() << " posicoes.”
            << endl;
   }
}
```

Classe auxiliar: IndexOutOfBounds

```
class IndexOutOfBounds {
  int index;
  int limite;
public:
  IndexOutOfBounds(int i, int l) {
    index = i;    limite = l;
  }
  int pos() {    return index;  }
  int max() {    return limite;  }
};
```

Reuso de código

- Como oferecer a mesma funcionalidade para outros tipos de arranjos?
  - definir classes similares para cada tipo básico: ArrayChar, ArrayShort, ..., ArrayDouble
    - repetir procedimento para classes de outros tipos: ArrayOstream, ArrayIstream, ArrayComplex, etc.
  - usar o mecanismo de template

Definição de template de classes

- Definição da classe é precedida pela palavra−chave **template**
- Após a palavra−chave template, indicação do tipo de parâmetros para a definição entre < e >
  - class ou typename
- Segue a definição da classe, usando o parâmetro indicado no lugar do “tipo variável”

Métodos da classe parametrizada

- Método size( ) foi definido inline
  - nenhuma indicação especial foi necessária
- Para os métodos definidos externamente à classe, é preciso indicar que definição é de classe parametrizada
  - usar “template<class *tipo*>”
  - mesmo que parâmetro *tipo* não esteja sendo utilizado no método

Classe parametrizada Array

```
template<class T>
class Array {
  T* A;
  const int limite;
public:
  Array(int n=100);
  T& operator[](int index);
  int size() { return limite; }
};
```

Métodos de Array

```
template<class T>
Array<T>::Array(int n) : limite(n) {
  A = new T[n];
}
template<class T>
T& Array<T>::operator[](int index) {
  if (index < 0 || index >= limite)
    throw new IndexOutOfBounds(index,limite);
  return A[index];
}
```

Uso de Array

```
int main() {
  Array<int> ia(20);        // arranjo de 20 ints
Array<double> da(30); // arranjo de 30 doubles
  try {
    for (int i=0; i<ia.size(); ++i)  da[i] = ia[i] = i+1;
    cout << ”ia[35] = " << ia[35] << endl;
  }
  catch(IndexOutOfBounds* iof) {
   cout << ”Acesso a posicao " << iof−>pos() <<
         " em array de "<< iof−>max() << " posicoes.”
         << endl;
  }
}
```

Templates e arquivos de cabeçalho

- Definições de classes parametrizadas e seu código podem normalmente estar presentes em um arquivo de cabeçalho
  - código de template não é a definição ainda, que ocorrerá com a instanciação do template no código da aplicação
  - pode haver necessidades especiais em colocar as definições de código de template em arquivos fonte; checar compilador

Constantes em templates

- Constantes podem ser passadas como argumentos para template

```
template<class T, int limite=100>
class Array {
  T A[limite];
public:
  T& operator[](int index);
  int size() { return limite; }
};
```

Constantes em templates

```
template<class T, int limite>
T& Array<T,limite>::operator[](int index) {
  if (index < 0 || index >= limite)
    throw new IndexOutOfBounds(index,limite);
  return A[index];
}
int main() {
  Array<int,20> ia;
  Array<double,30> da;
  ...
```

Templates de funções

- Definição de métodos de uma classe parametrizada especifica uma família de métodos
  - um membro para cada instanciação do template
- Mesmo mecanismo pode ser aplicado a funções globais com corpo similar exceto pelos tipos manipulados
  - funções parametrizadas

Uso das funções parametrizadas

- Seleção da função instanciada ocorre de acordo com os tipos dos argumentos da função
- Exemplo 1: swap de inteiros

```
int i=10, j=11;
swap(i,j);
```

- Exemplo 2: swap de complexos

```
complex<double> c(1,1), h(2,2);
swap(c,h);
```

Definindo uma função parametrizada

- Exemplo: troca de valores entre duas variáveis de mesmo tipo

```
template<class T> void swap(T& a, T& b) {
  T temp = a;
  a = b;
  b = temp;
}
```

Sobrecarga de funções parametrizadas

- Se função parametrizada coexiste com outras funções de mesmo nome
  - Inicialmente, procura por função com tipos na lista de argumentos idênticos à invocação
  - Caso falhe, procura por função parametrizada que possa gerar função com tipos na lista de argumentos idênticos à invocação
  - Caso falhe, aplica resolução padrão de sobrecarga com conversão de tipos
  - Caso falhe, sinaliza erro

Resolução de sobrecarga

- Exemplo

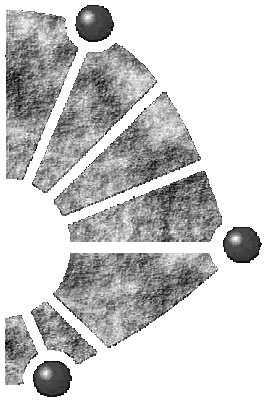
```
int main( ) {
  int i=10;
  double d=21;
  swap(i,d);
}
```

- Erro: não é possível gerar swap(int,double);

O que é a STL

- C++ Standard Template Library
  - conjunto de classes e algoritmos parametrizados
    - containers
    - algoritmos genéricos
    - iterators
    - objetos função
    - adaptadores
    - alocadores

Programação Orientada a Objetos com C++



Uma visão geral da biblioteca de *templates* padrões

(STL)

Containers

- Um container é um objeto que armazena coleções de outros objetos
  - eventualmente, outros containers
- Dois tipos de containers em STL
  - containers de seqüências
  - containers associativos ordenados

Containers de seqüências

- Organizam uma coleção de objetos de mesmo tipo em uma estrutura linear de tamanho variável
- Podem ser de três tipos:
  - vector<T>
  - deque<T>
  - list<T>

deque<T>

- Oferece acesso direto a uma seqüência de comprimento variável
  - tempo de leitura de um elemento em qualquer posição não depende da posição
- Tempo de inserção e de remoção de elementos é constante para o início e para o final da seqüência

vector<T>

- Oferece acesso direto a uma seqüência de comprimento variável
  - tempo de leitura de um elemento em qualquer posição não depende da posição
- Tempo de inserção e de remoção de elementos é constante apenas para o final da seqüência

list<T>

- Tempo de acesso a uma posição na seqüência varia linearmente com o comprimento da seqüência
- Tempo de inserção e de remoção de elementos é constante para qualquer posição da seqüência

Containers associativos ordenados

- Definem coleções de objetos de tamanho variável com rápida recuperação baseada em valores de chaves
  - set<Key>
  - multiset<Key>
  - map<Key, T>
  - multimap<Key, T>

map<Key, T>, multimap<Key, T>

- map<Key, T>
  - permite a recuperação rápida de um tipo T com base no valor da chave de tipo Key
  - sem repetição de valores de chaves
- multimap<Key, T>
  - como map<Key, T>, porém permitindo a replicação de valores de chaves

set<Key>, multiset<Key>

- set<Key>
  - define coleções de objetos (chaves) sem repetição de valores
  - rápido tempo de recuperação
- multiset<Key>
  - define coleções de objetos onde a repetição de chaves é permitida
  - rápido tempo de recuperação

Algoritmos genéricos

- Complementam a definição de containers
  - oferecem funcionalidade genérica (algoritmo)
- Trabalham igualmente com containers, arranjos e strings
  - classe string
    - definida com uma seqüência de caracteres

Exemplos de algoritmos genéricos

- fill
  - preenche um trecho de uma coleção com o mesmo valor repetido múltiplas vezes
- generate
  - preenche um trecho de uma coleção com o valor de retorno de uma função especificada
- count
  - produz o número de elementos que, em um trecho da coleção, satisfaz um predicado

Exemplos de algoritmos genéricos

- search
  - tenta localizar a ocorrência de uma faixa de valores em uma faixa da coleção
- min_element
  - localiza o menor valor em uma faixa da coleção
- max_element
  - localiza o maior valor em uma faixa da coleção

Exemplos de algoritmos genéricos

- copy
  - copia trechos de uma coleção especificada para outra coleção
- copy_backward
  - como copy, mas em ordem inversa
- reverse
  - reverte a ordem de um trecho da coleção
- rotate
  - troca dois trechos de uma coleção

Exemplos de algoritmos genéricos

- replace
  - varre uma faixa da coleção trocando o valor especificado (se ocorrer) pelo novo valor
- replace_if
  - como replace, mas em função de um predicado especificado ser verdadeiro para o valor armazenado na coleção
- equal
  - verifica se duas faixas têm os mesmos valores na mesma ordem

Exemplos de algoritmos genéricos

- remove
  - rearranja a coleção de forma que os elementos “removidos” estejam no final, retornando a indicação da nova posição final
- unique
  - remove elementos duplicados da coleção
- sort, stable_sort
  - ordenam elementos na faixa especificada da coleção

Iteradores

- Base para operação dos algoritmos genéricos
  - oferecem referências a posições nas coleções
- Cinco categorias
  - Input iterators
  - Output iterators
  - Forward iterators
  - Bidirectional iterators
  - Random access iterators

Exemplos de algoritmos genéricos

- binary_search
  - realiza busca binária em faixa ordenada da coleção
- merge
  - combina elementos de duas coleções ordenadas, com resultado ordenado
- set_union, set_intersection, set_difference
  - operações de conjuntos sobre coleções ordenadas

Input e output iterators

- Input iterator
  - Requer a implementação dos operadores
    - igualdade: ==, !=
    - conteúdo de (leitura): *
    - avanço: ++ (forma pré−fixa e pós−fixa)
- Output iterator
  - Requer a implementação dos operadores
    - conteúdo de (escrita): *
    - avanço: ++ (forma pré−fixa e pós−fixa)

Forward, bidirectional e random access iterators

- Forward iterator
  - avanço: ++ (forma pré−fixa e pós−fixa)
- Bidirectional iterator
  - engloba forward iterator
  - retrocesso: −− (forma pré−fixa e pós−fixa)
- Random access iterator
  - engloba bidirectional iterator
  - +, − , +=, −=, <, >, <=, >=

Adaptadores

- Componente que modifica a interface de outro componente
  - exemplos:
    - reverse_iterator, para inverter a ordem de varredura de um iterador
    - stack_adaptor ou queue_adaptor, para restringir a política de acesso a um container
    - binder, para converter um objeto função de binário para unário fixando o valor de um dos operandos

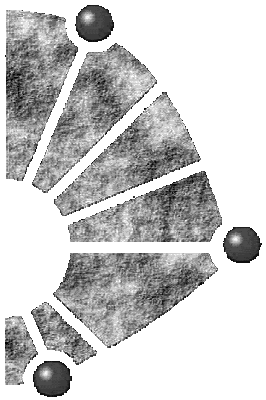
Objetos função

- Funções que podem ser passadas como argumentos para os algoritmos genéricos
  - especificando operações binárias, predicados
- Genericamente, pode ser também qualquer objeto de uma classe cujo operador de chamada de função, ( ), tenha sido sobrecarregado

Alocadores

- Classes associadas a containers com o objetivo de isolar os detalhes internos de armazenamento
  - manipulação de ponteiros e referências é oculta pelos métodos dos alocadores
  - eventualmente (raramente), classe poderia ter implementação alternativa ao default de STL para suportar outro modelo de alocação

Programação Orientada a Objetos com C++



Comentários finais

Para onde ir daqui?

- Prática
  - não se domina uma linguagem de programação na teoria
- Domínio da linguagem de programação não garante sucesso de desenvolvimento
  - aplicação de técnicas sólidas de projeto e de implementação
    - design patterns, extreme programming
  - trabalho em equipes
    - com forte gerência de projetos

C++ e outras linguagens

- Tempo de desenvolvimento de aplicações relativamente longo
- Uso eficiente de memória
  - metade do uso de código equivalente em linguagem script, um terço de Java
- Execução mais rápida
  - cerca de duas vezes mais rápido que Java
    - *Fonte: An empirical comparison of seven programming languages (Lutz Prechelt), IEEE Computer, October 2000*