Objects in C++

Objects, with dynamic lookup of virtual functions

C++ Object System

- Object-oriented features

1. Classes and Data Abstraction
2. Encapsulation
3. Inheritance
   1. Single and multiple inheritance
   2. Public and private base classes
4. Objects, with dynamic lookup of virtual functions
5. Subtyping
   1. Tied to inheritance mechanism

Polymorphism in C++

- Runtime polymorphism
- Virtual functions
- Compile-time polymorfism
- (parametric polymorfism)
- Generic programming
- templates

Run-time Polymorphism

- **Run-time polymorphism**: implemented with **dynamic lookup of virtual functions**
- ***Dynamic lookup***: a method is selected dynamically, at run time, according to the implementation of the object that receives a message
  - not some static property of the pointer or variable used to name the object

- The important property of dynamic lookup is that **different objects may implement the same operation differently**

Virtual functions

- Member functions are either
  - Virtual, if explicitly declared or inherited as virtual
  - Non-virtual otherwise
- Non-virtual functions
  - Are called in the usual way. *Just ordinary functions*.
  - May be redefined in derived classes (overloading through *redefining*)
- Pay overhead only if you use virtual functions

Virtual members

- Must be explicitly declared as “virtual”
- May be *overridden* in derived (sub) classes
- Dynamic binding is activated
- Are accessed by indirection through **ptr** in object
- Explicitly as pointers or using references

```
class A { public: virtual void vi(){...}};
class B : public A{ public: virtual void vi(){ …}};
int main() {
     A* pa = new A; a -> vi(); // VIRTUAL CALL
     A& ra = b; ra.vi(); // VIRTUAL CALL
     A a = b; a.vi(); // NON VIRTUAL CALL
}
```

Sample class: one-dimen. points

```
class Pt {
   public:
      Pt(int xv);
      Pt(Pt* pv);
      int getX();
      virtual void move(int dx);
   protected:
       void setX(int xv);
   private:
      int x;
   };
```

Overloaded constructor

Public read access to private data

Virtual function

Protected write access

Private member data

Sample derived class

```
class ColorPt: public Pt {
   public:
      ColorPt(int xv,int cv);
      ColorPt(Pt* pv,int cv);
      ColorPt(ColorPt* cp);
      int getColor();
      virtual void move(int dx);
      virtual void darken(int tint);
   protected:
      void setColor(int cv);
   private:
      int color;
 };
```

Overloaded constructor

Non-virtual function

Virtual functions

Protected write access

Private member data

Sample derived class

```
/* ----Definitions of Member Functions --------*/

void ColorPt::darken(int tint) { color += tint; }

void ColorPt::move(int dx) {
Pt::move(dx); this->darken(1);
}
```

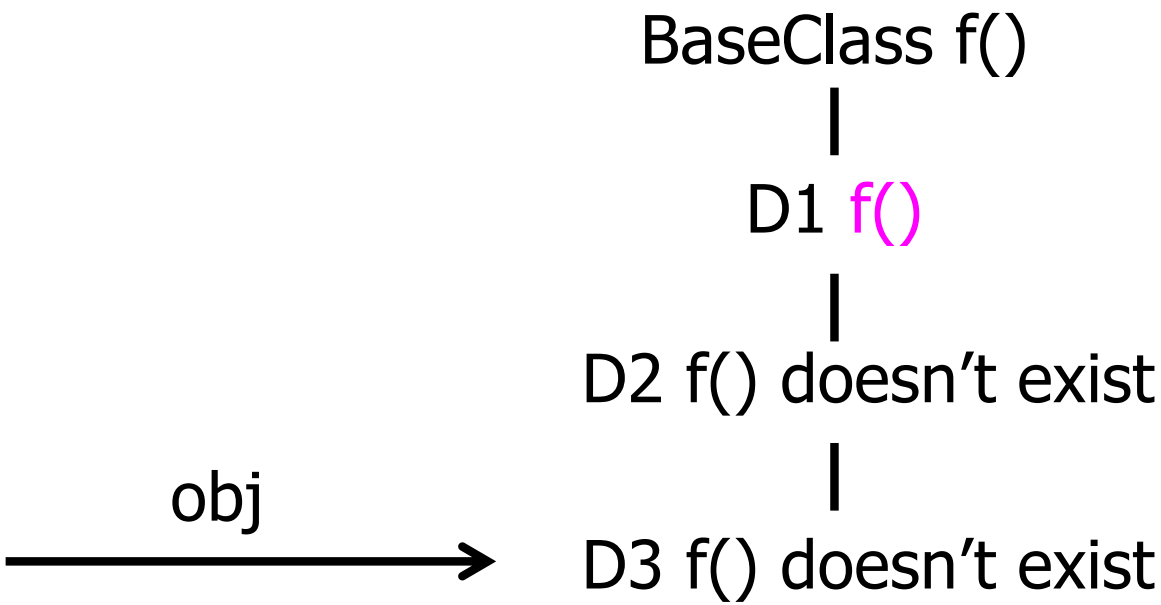
Virtual functions and *indirection (1)*

- C++ allows a base class pointer to point to a (public) derived class object
- Upon method invocation, the method of the derived object is called (**dynamic binding**)
- This leads to generic alghoritms using base class pointers

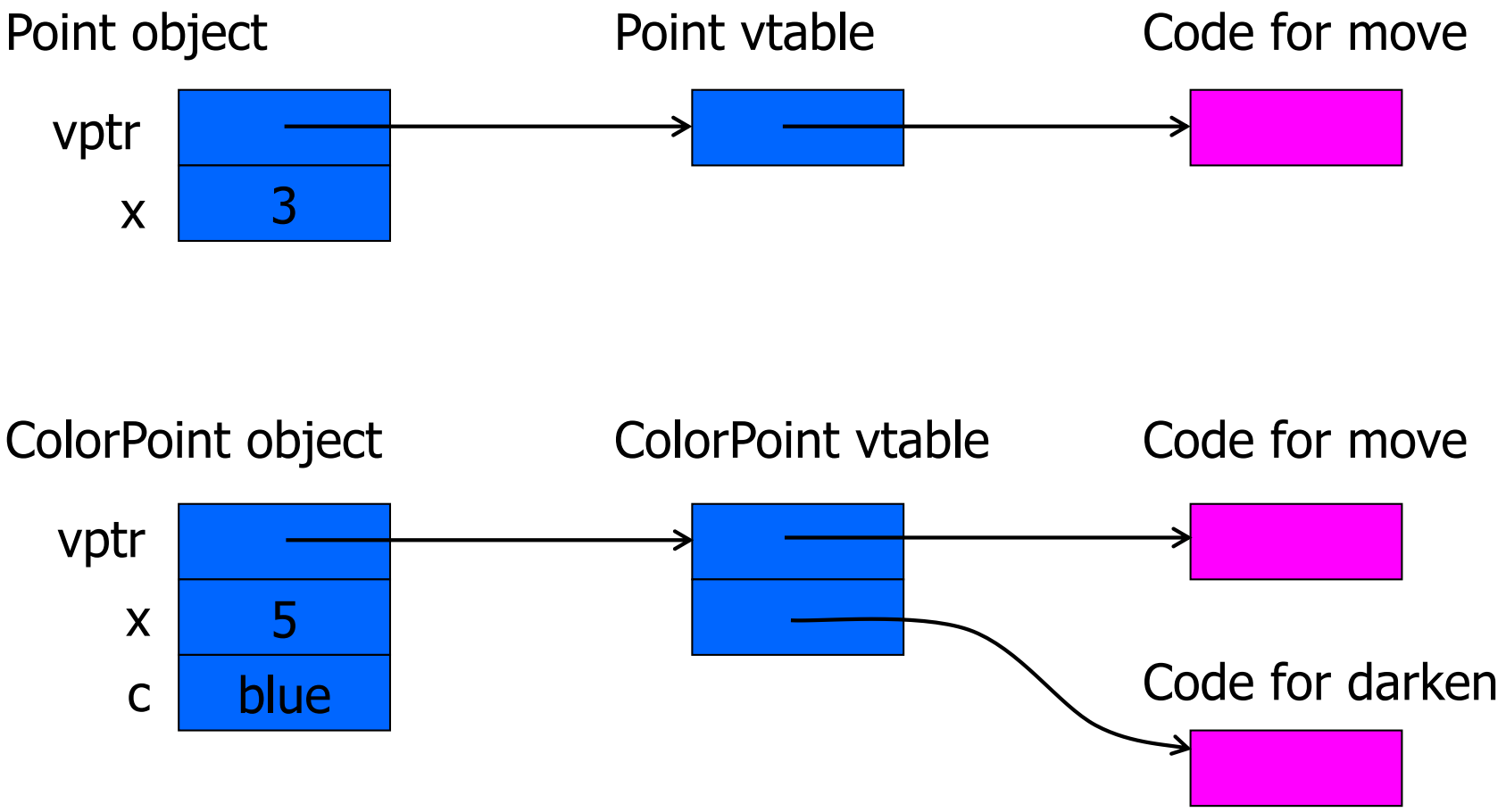
```
Pt*  ptr = new ColorPt;

ptr->move();
```

Virtual functions and *indirection (2)*



Run-time representation



“this” pointer

- Code is compiled so that member function takes “object itself” as first argument

  Code `int A::f(int x) { … g(i) …;}`

  compiled as `int A::f(A *this, int x) { … this->g(i) …;}`

- “this” pointer may be used in member function
- Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Constructors/destructors and inheritance (2)

- destructors
  - always make destructors virtual in base classes
  - there might be cleanup work to be done in derived classes

```
class Employee {
  //...
  public:
   //...
   virtual ~Employee() {}
};
```

Non-virtual functions

- How is code for non-virtual function found?
- Same way as ordinary "non-member" functions:
- Compiler generates function code and assigns address
- Address of code is placed in **symbol table**
- At call site, address is taken from symbol table and placed in compiled code
- *But* some special scoping rules for classes
- Overloading
  - Remember: overloading is resolved at compile time
  - This is different from run-time lookup of virtual function

Overload

- An **overloaded** function is a function that shares its name with one or more other functions, but which has a different parameter list. The compiler chooses which function is desired based upon the arguments used.

Overridden

- An **overridden** function is a method in a descendant class that has a different definition than a **virtual** function in an ancestor class. The compiler chooses which function is desired based upon the type of the object being used to call the function.
  - Regardless the access modifier (private and so on) of the function
  - Si può fare overriding anche di metodi private (se virtual)
    - Not like Java
    - Vediamo un esempio

•redefined

- A **redefined** function is a method in a descendant class that has a different definition than a non-virtual function in an ancestor class. Don't do this. Since the method is not virtual, the compiler chooses which function to call based upon the static type of the object reference rather than the actual type of the object.

Virtual vs redefined Functions

```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");};   };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");};    };
main() {
    parent p;  child c; parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p;  q->printclass(); q->printvirtual();
    q = &c;  q->printclass(); q->printvirtual();
}
```

Output: p p c c p p **p** c

Esercizio

- Definiamo una classe A con un metodo virtual che ridefiniamo (overriding) in una sottoclasse B.
- Proviamo a chiamare quel metodo in diversi casi

Function call binding

- Early binding (C,C++)
  - At compile time
- Late binding (C++)
  - At runtime
- Mighty. But less efficient
- 1 more assembler statement per call
- Slight memory consumption due to the VPTRs

Objects in C++

Subtyping

C++ Object System

- Object-oriented features

1. Classes and Data Abstraction
2. Encapsulation
3. Inheritance

- Single and multiple inheritance
- Public and private base classes

4. Objects, with dynamic lookup of virtual functions
5. Subtyping

- Tied to inheritance mechanism

Subtyping (1)

- **Subtyping** is a relation on types that allows values of one type to be used in place of values of another.
- If some object **`a`** has all of the functionality of another object **`b`**, then we may use **`a`** in any context expecting **`b`**.
- **Inheritance Is Not Subtyping**
- *"Subtyping is a relation on interfaces, inheritance is a relation on implementations."*
- **A typical example is C++,** in which
- A class A will be recognized by the compiler as a **subtype of** B only if B is a public base class of A

Subtyping (2)

- (A<:B = A subtype of B)
- Subtyping in principle
- A <: B if every A object can be used without type error whenever a B object is required

Pt:	int getX();	Public members
	void move(int);	
ColorPt:	int getX();	Public members
	int getColor();	
	void move(int);	
	void darken(int tint);	

- C++: A <: B if class A has public base class B

Sample public derived class

```
class ColorPt: public Pt {
  public:
    ColorPt(int xv,int cv);
    ColorPt(Pt* pv,int cv);
    ColorPt(ColorPt* cp);
    int getColor();
    virtual void move(int dx);
    virtual void darken(int tint);
  protected:
    void setColor(int cv);
  private:
    int color;
 };
```

In C++: public base class gives supertype!

Overloaded constructor

Non-virtual function

Virtual functions

Protected write access

Private member data

Public inheritance and subtyping

```
class ColorPt: public Pt {
  ....
};
```

ColorPt is a subtype of Pt.

I can write

```
Pt * p = new ColorPt;

// not so good
ColorPt cpt;
Pt p = cpt;
```

private derived class are not subtypes

```
class ColorPt: private Pt {
  ....
};
```

ColorPt is not a subtype of Pt.

I cannot write

```
Pt * p = new ColorPt;

ColorPt cpt;
Pt p = cpt;
```

Independent classes not subtypes

```
class Point {
   public:
      int getX();
      void move(int);
    …
};
```

```
class ColorPoint {
   public:
      int getX();
      void move(int);
      int getColor();
      void darken(int);
   …
};
```

- C++ does not treat ColorPoint <: Point as written
- Need public inheritance ColorPoint : public Pt
- Subtyping based on inheritance:
  - An efficiency issue
  - An encapsulation issue: preservation under modifications to base class … *inheritance breaks encapsulation*
    - *We will see "duck subtyping"*

Why C++ design?

- Client code depends only on public interface
  - In principle, if ColorPt interface contains Pt interface, then any client could use ColorPt in place of point
- However -- offset in virtual function table may differ
- Lose implementation efficiency
- Without link to inheritance
  - subtyping leads to loss of implementation efficiency
- Also encapsulation issue:
  - Subtyping based on inheritance is preserved under modifications to base class ...

Sottotipazione e covarianza. Riassunto critico

Rivediamo il concetto di sottotipazione per funzioni (metodi)

http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

Sottotipazione in generale

- A <: B : A sottotipo di B
- Subtyping principle
- A <: B se un espressione A può essere usata in modo sicuro in ogni contesto in cui sarebbe richiesto un B
- In questo modo vale il principio di sostituibilità
- Per variabili (instanze di classe):
- B b = new A() (java)
- B* B = new A; (C++)
- Dove mi aspetto un B posso passare un A.

Sottotipazione per funzioni

- Per le funzioni?
- Posso estendere e dire che una funzione è sottotipo di un'altra se può essere usata al suo posto.
- In teoria potrei ammettere l'overriding e il binding dinamico di tutte le funzioni che siano sottotipo
- Quando una funzione f: W □ Z è sottotipo di un altra?
- f:W->Z vuol dire che prende un W e restituisce un Z

Covarianza tipo restituito

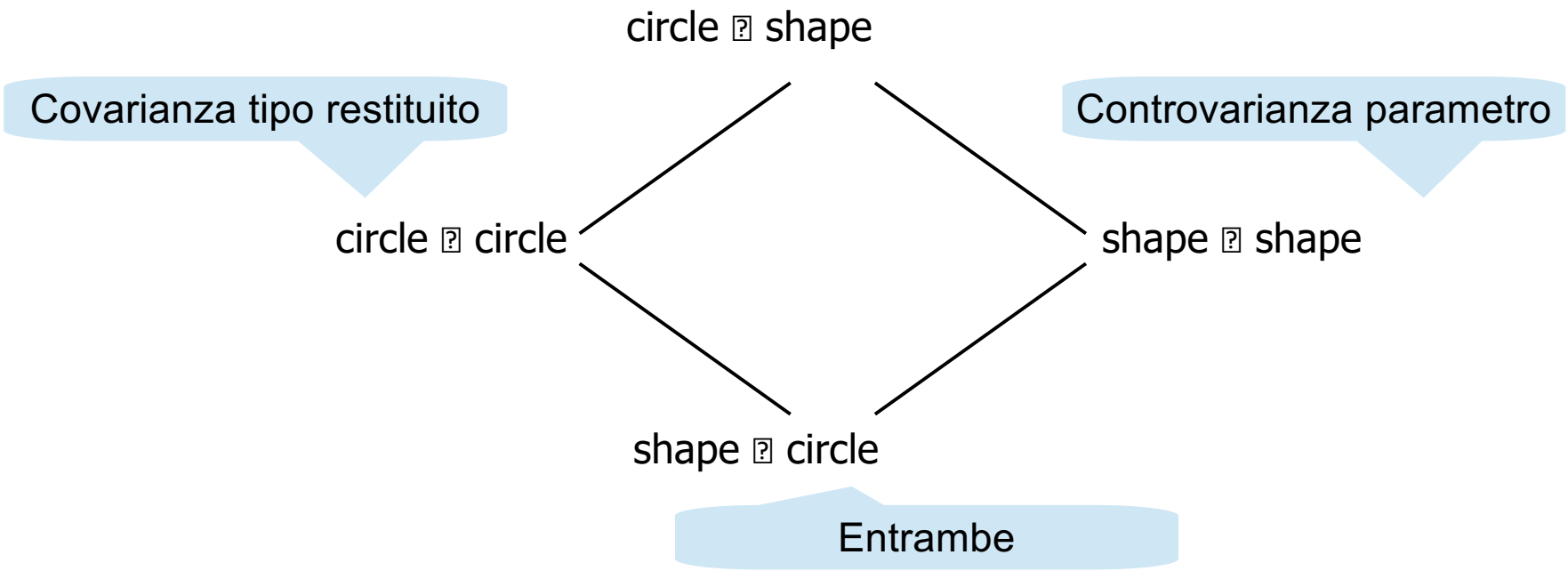
- Rispetto il tipo ritornato, basta che ci sia covarianza:
- If A <: B, then fa:C → A <: fb: C → B
- Cioè fa può essere usata al postodi fb se ritorna un sottotipo (A invece che B)
- Uso
- Al posto di fb(c) [di tipo B]
  posso metterci fa(c) [di tipo A]
- Una sottoclasse quindi potrebbe "sostituire" fb con fa
- Terminology
- Covariance: A <: B implies F(A) <: F(B)

Rispetto ai parametri?

- Rispetto ai suoi argomenti (controvarianza)
- If A <: B, then fb:B → C <: fa: A → C
- Cioè fb può essere usata al posto di fa se prende come argomento un supertipo (B invece che A)
- Uso
- fa(x) [x di tipo A] può essere sostituita da fb(x)
- Una sottoclasse quindi potrebbe "sostituire" fa con fb
- Terminology
- Contravariance: A <: B implies F(B) <: F(A)

Esempio

- Se circle <: shape, then



Nei linguaggi di programmazione

- In pratica i linguaggi di programmazione permettono la "sostituzione"- overriding solo in alcuni casi:
- C++: covarianza tipo ritornato in virtual functions
- Java: covarianza tipo ritornato (da Java 5)
- I parametri devono avere lo stesso tipo (invarianza) altimenti ho overloading
- … altri linguaggi ...

In C++ - from 1998

- C++ supports the covariance of return types
- Only virtual
- Only pointers
- Example

```
class A{
public:
  virtual A * create() ...
};
class B : public A{
public:
  virtual B * create() … // overriding
};
```

Subtyping with functions

```
class Point {
   public:
      int getX();
      virtual Point *move(int);
   protected:    ...
   private:      ...
};
```

```
class ColorPoint: public Point {
   public:
      int getX();          // Inherited, but repeated here for clarity
      int getColor();
      ColorPoint * move(int);
      void darken(int);
   protected:    ...
   private:      ...
};
```

- In principle: can have ColorPoint <: Point
- In practice: some compilers allow, others have not

  This is covariant case; contravariance is another story

In Java

- Covarianza del tipo restituito, già visto

Slicing - attenzione

```
class A {
   int foo;
};
class B : public A
{
   int bar;
};
```

- So an object of type B has two data members, foo and bar

- Polimorfism does not work without pointers, but copy constructor:
- B b;
- A a = b

- a will have only the foo attribute ! The member bar of b is lost

Details, details

- This is legal

```
class Point { …
    virtual Point * move(int);
…   }
class ColorPoint: public Point {   …
    virtual ColorPoint * move(int);
 …  }
```

- But not legal if *'s are removed

```
class Point { …   virtual Point move(int); … }
class ColorPoint: public Point { …virtual ColorPoint move(int);… }
```

Related to subtyping distinctions for object L-values and object R-values

(Non-pointer return type is treated like an L-value for some reason)

Abstract Classes

- Abstract class:
  - A class that has at least one *pure virtual member function*, i.e a function with an empty implementation
- Declare by: **`virtual function_decl = 0;`**
- A class without complete implementation
- Useful because it can have derived classes

  Since subtyping follows inheritance in C++, use abstract classes to build subtype hierarchies.
- Establishes layout of virtual function table (vtable)
- Example
- Geometry classes
- Shape is abstract supertype of circle, rectangle, ...

C++ Summary

- Objects
- Created by classes
- Contain member data and pointer to class
- Encapsulation
- member can be declared public, private, protected
- object initialization partly enforced
- Classes: virtual function table
- Inheritance
- Public and private base classes, multiple inheritance
- Subtyping: Occurs with public base classes only

Duck typing

- deriva dal detto "Se cammina come un'anatra e fa starnazzare come un'anatra, allora deve essere un'anatra".
- Il duck typing è legato alla tipizzazione dinamica
- il tipo o la classe di un oggetto è meno importante dei metodi che definisce.
- Python usa il duck typing, non controlla i tipi (che non sono dichiarati).
- Invece, controlla la presenza di un determinato metodo o attributo.

Esempio

GENERIC ABSTRACTIONS in C++

- C++ Templates
- STL (Standard Template Library)

9.4 *Programming Languages Concepts*

by John Mitchell

Overview

- Motivation
- Template review
  - Function template
  - Class template
- What is the STL?
  - Containers
  - Iterators
  - Algorithms
- Glossed-over stuff

Motivation

- Abstract data types such as stacks or queues are useful for storing many kinds of data
- It is time consuming to write different versions of stacks for different types of elements
- Most typed languages support some form of **type parameterization**
- The **C++ template** is the most familiar type-parameterization mechanism
- The **C++ STL** is a large program library of parameterized abstract data types

C++ Function Template (1)

- A simple swap function:

```
void swap(int& x, int& y){
            int tmp=x; x=y; y=tmp; }
```

- A function template with a type variable **T** in place of **int**:

```
template<class T>
        void swap(T& x, T& y){
             T tmp=x;x=y;y= tmp; }
```

nota:

```
template<typename T> =template<class T>
```

C++ Function Template (2)

- Function templates are instantiated automatically by the program linker using the types of the function arguments

```
int i,j;
...
swap(i,j); // Use swap with T replaced by int
string s,t;
...
swap(s,t); // Use swap with T replaced by String
float a;
...
swap(i,a); // ERROR
```

C++ Function Template (3)

- For each type variable, at least one function argument must depend on the type variable
  - `template<class T> T f(T &); //OK`
  - `template<class T> T f(double); //ERROR`
  - `template<class T> T f(double, T&); //OK`
  - `template<class T, class S> T f(T &, S &); //OK`
  - `template<class T, class S> T f(S &); //ERROR`

C++ Function Template (4)

- Operations on Type Parameters limit the variability of the parameters

- A generic sort function:

```
template <class T>
void sort( int count, T * A[count] ) {
  for (int i=0; i< count-1; i++)
   for (int j=i+1; j< count-1; j++)
        if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- If A is an array of type **T**, then **sort(n, A)** will work only if operator < (possibly *overloaded*) is defined on type **T**

Esercizio

- Definiamo la funzione max tra due elementi generici.

C++ Class Template

```
template <class T> class Complex {
     private:
          T re,im;
     public:
          Complex (const T& r, const T& I)

               :re(r),im(i){}
          T getRe() {return re;}
          T getIm() {return im;}
}
```

- Type variables are fixed explicitly when the object is initialized
  - `Complex <double> x(1.0,2.0) // T = double`
  - `Complex <int> j(3,4) // T = int`
  - `Complex <char*> str("1.0","6") // T = char *`

C++ Class Template

- Type variables can be constant

```
template <class T, int dim> class Message{
private:
T mess[dim];
...
public:
Message (T *str, int n){
 int end = min(n,dim);
 for(int i=0; i<end; i++)
     mess[i]= str[i];
}
Message <char, 80> m (“Message 1”, 8);
// T = char, dim = 80
```

What is the STL?

- "Standard Template Library" by Alex Stepanov in 1976
- Basic motivation:
  - *N* data types, *M* containers, and *K* algorithms
  - Possibly *N* **M* **K* implementations
    - CountIntegerInList(IntList il, int toFind), CountIntegerInSet, CountDoubleInList, etc.
  - STL (with C++ templates): *N* +*M* +*K* implementations
    - algorithms operate over containers of types
    - `set<int> mySet;`
    - `count(mySet.begin(), mySet.end(), 4);`
    - `list<double> myList;`
    - `count(myList.begin(), myList.end(), 3.14);`

Platforms

- STL is part of Standard C++
- Porte din all the major compilers
- Stlport.org
  - Free std C++ implementation (including iostreams), some nice features/performance

STL overview

- Fundamentally, the STL defines *algorithms* that operate over a *range* in a *container*
- Our order:
  - **Containers**: a collection of typed objects
  - **Iterators** (ranges): generalization of pointer or address to some position in a container
  - **Algorithms**

Containers

- **Lists**
  - vector, list, deque
- **Adaptors**
  - queue, priority_queue, stack
- **Associative**
  - map, multimap, set, multiset
  - hash_{above}

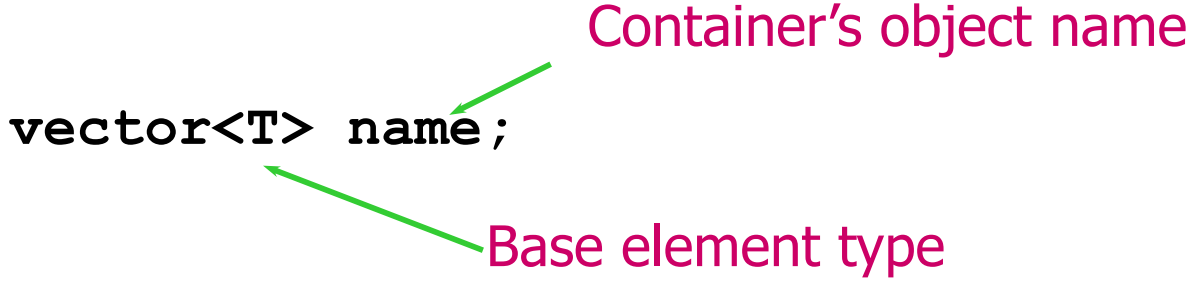
vector<T>

- #include <vector>
- A dynamic array: random-access, grows
- Array-indexing syntax: **`operator[](dim_type n)`**

```
vector<int> v(10); v[0] = 4;
```

Defining a Vector

- Basic definition



- The type can be any type or class!
- Must have: **`#include <vector>`**
- **`Must have: using namespace std;`**
- Creates an empty vector
- Example

```
vector<int> A;              // 0 ints
vector<double> B;           // 0 doubles
vector<string> C;           // 0 strings
```

Modifying a vector object

- Add a new element at the end of the vector
  - **`push_back(const T &val)`**
    - Inserts a copy of **`val`** after the last element of the vector
- Remove one element at the end of the vector
  - **`pop_back()`**
    - Removes the last element of the vector

How many elements?

- **`size_type size()`**
  - Returns the number of elements in the vector

```
cout << A.size();
```

  - Note: size_type is an “alias” name for an unsigned int

- **`bool empty()`**
  - Returns true if there are no elements in the vector; otherwise, it returns false

```
if (A.empty()) {
    // ...
```

Example vector 1

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
  vector<int> A;
  if ( A.empty() ) cout << "A has size zero. ";
  A.push_back(3); // A:  3
  A.push_back(-25); // A: 3 -25
  cout << "Size of A: " << A.size(); // size 2
  A.pop_back();  // A: 3
  cout << "Size of A: " << A.size(); // size 1
}
```

Removing All Elements

- Two member function calls to remove all elements
  - Sometimes we need to “clear out” an existing vector
- **`void resize(size_type s)`**
  - The number of elements in the vector is now **`s`**.
  - Use with zero to remove all elements
  - If you “grow” a vector, default value/constructor used for new items
- **`void clear()`**
  - Removes all elements

```
vector<int> A;
// assume we add elements to A here
A.resize(0);       // A is now empty
A.clear();         // same effect as above
```

Accessing Just One Element

- What if we want to retrieve or change one element?
  - Index value: from 0 to **`size()-1`**
  - Pass index to the **`at()`** member function
- Example:

```
vector<int> A;
// assume we add two or more elements to A
A.at(0) = A.at(1) + 1;
```

- Note: can be used on left-hand side of assignment!
  - E.g. this changes the element stored at index 0
- Example: set last element to value of 1st element

```
A.at( A.size()-1 ) = A.at(0);
```

What's Allowed on the Element?

- When you access one single element using **`at()`**, what are you allowed to do with that element?
  - **Anything** you could normally do with one variable of that type!
- Example: if **`A`** is a vector of **`int`**'s, and the element at index **`i`** exists
  - Element **`A.at(i)`** is an **`int`** just like any other **`int`** variable
    - We can print it, add to it, take its sqrt, pass it as a parameter to a function expecting an int
- Example: if **`S`** is a vector of strings, and **`S.at(i)`** exists
  - Element **`S.at(i)`** is one **`string`** object
    - We can print it, concatenate to it, call **`size`** or **`substr`** on it, pass it as a parameter to a function expecting a string

Vector Bounds Errors

- Elements only exist from index 0 to size()-1
  - Very common error to refer to **`A.at(i)`** where **`i==A.size()`**
  - If there are 10 items, the last one is at index 9
- What if you make such a *vector-bounds error*?
- The **`at()`** member function checks its parameter
  - If not in bounds, throws a run-time exception
  - Your program halts
  - (Heard of arrays? They don't do this check.)

Example 2

```
#include <vector>
#include <string>

int main() {
  int i;
  vector<string> A;
  A.push_back("I"); A.push_back("am");
  A.push_back("me");

  for (i = 0; i < A.size(); ++i) // why not <= ?
     cout << A.at(i) << " ";
  cout << endl;
```

Example 2 continued

```
  // swap 1st and last elements
  string Temp = A.at(0);
  A.at(0) = A.at( A.size()-1 ); // NOTE!!!
  A.at( A.size()-1 ) = Temp;

  A.at( A.size()-1 ) += "!"; // add ! to end

  for (i = 0; i < A.size(); ++i)
      cout << A.at(i) << " ";
  cout << endl;

  return 0;
}
```

Operating on the Whole Vector

- We can do some things on the entire vector
  - Assignment: If two vectors are defined to hold the same kinds of elements
  - Example:
  - ```
    vector<int> A, B;
    // assume we add some elements A
    B = A; // B's old contents gone, now == A
    ```

- Logical equality operators == and != work too

```
if (B == A) { // same size, same (==) elements ?
```
```

Function Examples: Input

```
void GetIntList(vector<int> &A) {
  A.resize(0);
  int Val;
  while (cin >> Val) {
      A.push_back(Val);
  }
}
```

```
vector<int> List;
cout << "Enter numbers: ";
GetIntList(List);
```

Function Example: Output

```
void PutIntList(const vector<int> &A) {
  for (int i = 0; i < A.size(); ++i) {
     cout << A.at(i) << endl;
  }
}
```

```
vector<int> MyList;
// somehow values get into MyList
cout << "Your numbers: ";
PutIntList(MyList)
```

- Question: Why is formal parameter const reference?

Other Useful Functions

- Often we need to search a vector for an item:
  - **`int find (const vector<T> &vect, T target);`**
    - Loops through the elements in the vector, searching for an element equal to **`target`**
    - Returns index of **`target`** if it's found.
    - If not found, return either -1 or **`vect.size()`**
    -
- Defined functions only allow us to add/remove at vector's end
  - By using **`push_back()`** and **`pop_back()`**
  - Could we write functions that take an index value and use it to tell us where to insert or remove an element?

Other Useful Functions (cont'd)

- **`void deleteAt (vector<T> &vect, int idx);`**
  - Remove the element at index **`idx`** (if it exists)
  - How? Must use loop to "shift down" elements, then call **`pop_back()`** to remove unneeded element at the end

- **`void insertAfter (vector<T> &vect,`**
- **`T newItem, int idx);`**
  - Add **`newItem`** after element with index **`idx`**
  - How?
    - Must **`push_back()`** to get one more "space"
    - Must use loop to "shift up" elements
    - Finally do: **`vect.at(idx+1) = newItem;`**

vector<T>

- Time:
  - constant time insertion and removal of elements at the end
  - linear time insertion and removal of elements at the beginning or in the middle.
- The “standard” container

Forward reference: Iterators

- v.begin() and v.end() return iterators
- Like pointers: arithmetic (++, --) and dereferencing (*)

```
for (vector<int>::iterator i =
  v.begin(); i != v.end(); ++i)
  cout << *i;
```

list<T>

- Bidirectional, linear list
- Sequential access only (not L[52])
- Constructors
  - list<T>()
  - list<T>(size_t num_elements)
  - list<T>(size_t num, T init)
- Properties
  - l.empty() // true if l has 0 elements
  - l.size() // number of elements

list<T>

- Adding/deleting elements
  - l.push_back(43);
  - l.push_front(31);
  - l.insert(iterator,4) // insert 4 before the position “iterator”
  - etc..
- Accessing elements
  - l.front() // T &
  - l.back() // T &
  - l.begin() // list<T>::iterator
  - l.end() // list<T>::iterator

list<T>

- Removing elements
  - l.pop_back() // returns nothing
  - l.pop_front() // returns nothing
  - l.erase(iterator i)
  - l.erase(iter start, iter end) // delete a *range*
- Time
  - Amortized constant time insertion and removal of elements at the beginning or the end, or in the middle [because you pass an iterator]

list<T>

- Other operations
  - l.sort(), l.sort(CompFn) // sorts in place
  - l.splice(iter b, list<T>& grab_from)

list<T>

- Example:

```
list<char> l;
for (int i = 0; i < 4; ++i)
{
  l.push_front(i + 'A');
  l.push_back(i + 'A');
}
for (list<char>::iterator i = l.begin();
  i != l.end(); ++i)
  cout << *i; // DCBAABCD
```

Other data structures

- Hashtables / Map
- Queue
- Stack
- Set
- …
- algorithms …

[hash_]map, [hash_]multimap

- A map is an “associative container”
- Given one value, will find another
  - map<string, int> is a map from strings to int’s
  - maps are 1:1, multimap are 1:n
- map, multimap are **logarithmic** when inserting/deleting
  - Needs to maintain sortedness
- hash_map, hash_multimap are amortized **constant time**
  - Not sorted (“hashed”)

Map functions

- m.insert(make_pair(key, value)); // inserts
- m.count(key); // times occurs (0, 1)
- m.erase(key); // removes it
- m[key] = value; // inserts it into the table
- m[key] //retrieves or creates a “default” for it
- i=m.begin(), i=m.end() // iterators
- i->first, i->second // per accedere a chiave e valore della coppia puntata da i

Hash_{…}

- There are hash_map, hash_multimap, hash_set, hash_multiset
- Basically, these are constant time insert/delete instead of log time
  - They don’t maintain sortedness
  - Me: reduced running time from 10 min to 5 min

Hash performance

- Fill with 100,000 random elements
- Lookup 200,000 random elements
  - Same random seed
- map: fill 0.59967s
- map: lookups 1.57483s
- hash_map: fill 0.615407s
- hash_map: lookups 0.872557s
- So, if you don’t need order, go with hash_map

Summary

- map: 1:1, sorted, m[k] = v
- multimap: 1:n, sorted, mm.insert(make_pair(k,v))
- set: unique elements, sorted
- multiset: multiple keys allowed, sorted
- hash_: faster but **not sorted**

Iterators

- Touched on earlier
- An iterator is like a pointer
- You can increment to it to go to the “next” element
- You can [sometimes] subtract or add N
- You can dereference it
- Different kinds of iterators
- Most useful when combined with algorithms

Iterators

- c.begin() = start
- c.end() = 1 past the last element
  - Never dereference end! (*c.end() is bad!)
- Why? Makes loops simpler.
- Prefer ++i because i++ makes a temporary object and returns it, incrementing later.

Different kinds

- Technically:
  - random access (i += 3; --i; ++i)
  - bidirectional (++i, --i), store/retrieve
  - forward (++i), store/retrieve
  - input (++i) retrieve
  - output (++o) store
- But, writing code directly using iterators hurts a lot

Practical iterators

- iterator
  - “Standard”, goes from beginning to end
  - c.begin(), c.end()
- const_iterator
  - Like iterator, but changes can’t be made (prefer!)
  - c.begin() and c.end() are overloaded so you can use them to assign their result to a const_iterator
- reverse_iterator
  - Goes from the end to the beginning with same semantics as iterator
  - Generally, c.rbegin() and c.rend()
  - list, vector, deque, map, multimap, set, multiset, hash_, string

Iterator example

```
vector<int> v;
for (int k = 0; k < 7; ++k) v.push_back(k);
display(v); // 0 1 2 3 4 5 6

for(vector<int>::iterator i = v.begin(); i != v.end();
   ++i)
   *i = *i + 3; // add three to content
display(v); // 3 4 5 6 7 8 9

for(vector<int>::const_iterator ci = v.begin();
 ci != v.end(); ++ci)
   cout << *ci << ' ';// *ci = *ci - 3; won't compile
cout << endl;//    3 4 5 6 7 8 9

for (vector<int>::reverse_iterator ri = v.rbegin();
  ri != v.rend(); ++ri)
{  *ri = *ri - 3;
   cout << *ri << ' ';}
cout << endl; //6 5 4 3 2 1 0
```

Sort Functions

- Just a touch!

```
vector<int> v;
// fill v with 3 7 5 4 2 6
sort (v.begin(), v.end() );
```

Polymorphic STL containers and iterators

- **STL and inheritance do not mix well together**
- A STL container expects to contain its objects directly, so:

```
ellipse e(rect1);
rectangle r(rect2);
list < shape > shapeList;
shapeList.push_back(e);
shapeList.push_back(r);
```

- this code will compile but will do a slicing to shapes.

(2)

- It becomes apparent that one level of indirection is needed to solve the problem. An obvious solution is to change the list of shapes to a list of pointers to shapes:
- list < shape* > shapeList;
- and list would be populated:
- shapeList.push_back(&e);
- shapeList.push_back(&r);
- Other problems arise...

Esempio

- Lista di studenti con due classi derivate LS e IL
  …
- Implementa un metodo (virtual?) calcolo media
- Fai un vector di Studenti
- Inserisci tre studenti
- Chiama per tutti in metodo calcolo media

Conclusion

- The STL has everything
- Let the compiler do the work for you
- Saves time and lines of code
- **Run-time efficiency** of the code that is generated
- Next steps:
  - Buy a good book on STL
    - Schildt's STL Programming from the Ground Up
  - Use it on your homeworks/personal projects
  - Learn about function objects
    - Didn't have time to cover them; another talk??

Resources

- Books
  - Schildt – “STL Programming from the Ground Up” ***
  - Schildt – “C/C++ Programmers Reference”
- URLs
  - http://www.stlport.org/resources/StepanovUSA.html
  - http://www.usenix.org/publications/library/proceedings/coots97/full_papers/sundaresan/sundaresan_html/node2.html
  - MSDN
  - Google: sgi stl <container or algorithm>

C++
Smart pointers

•ANGELO GARGANTINI

•PROGRAMMAZIONE AVANZATA AA 22/23

Cosa sono gli smart pointer

Gli smart pointer cercano di risolvere il problema della gestione della memoria (memory allocation e deallocation)

- Evitare dangling pointers, memory leak etc.

Il principio è RAII o Resource Acquisition Is Initialization.

Obiettivo: quello di dare la proprietà di qualsiasi risorsa allocata nell'heap a un oggetto allocato nello stack il cui distruttore contiene il codice per eliminare o liberare la risorsa.

SMART POINTER-→ STACK + HEAP

- DEALLOCATO → STACK + PULIZIA DELLO HEAP

Come si usano i puntatori intelligenti

```
void UseRawPointer() {
// Utilizzo di un puntatore raw - non
consigliato .
Song *pSong = new Song(" Nothing on You");
// Usa pSong ...
// Non dimenticare di eliminare !
delete pSong;
}
```

I puntatori raw sono passati ad un puntatore intelligente che lo gestirà

```
void UseSmartPointer() {
// Dichiara un puntatore intelligente sullo stack e
// gli passa il puntatore raw.
unique_ptr<Song> song2(new Song(" Nothing on You"));
// Usa song2 ...
//wstring s = song2 -> duration_;
// ...
} // song2 viene cancellata automaticamente qui.
```

Distruzione di un puntatore intelligente

- Il distruttore del puntatore intelligente contiene la chiamata per eliminare anche il puntatore raw
  - poiché il puntatore intelligente è dichiarato nello stack, il suo distruttore viene richiamato quando il puntatore intelligente esce dallo scope
- Sono sicuro che il puntatore verrà cancellato (delete)

- Importante mai utilizzare l'espressione new o malloc sul puntatore intelligente stesso.

Come usare un puntatore intelligente

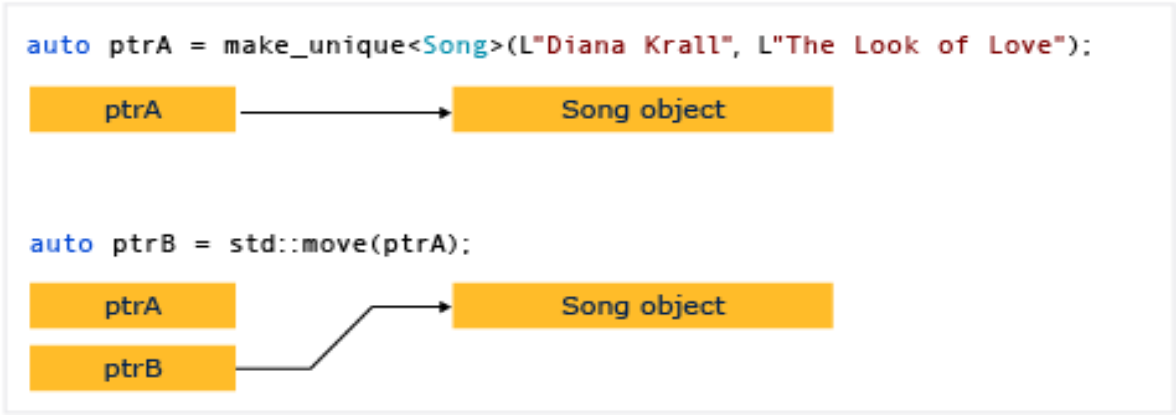
- Per accedere al puntatore incapsulato si utilizzano i soliti operatori del puntatore, -> e *, che la classe del puntatore intelligente sovraccarica per restituire il puntatore grezzo incapsulato.

- L'accesso al puntatore incapsulato utilizzando gli operatori smart pointer sovraccaricati * e -> non è significativamente più lento rispetto all'accesso diretto ai puntatori raw.
- I puntatori intelligenti di solito forniscono un modo per accedere direttamente al loro puntatore grezzo (get) – da usare con molta attenzione

Tipi di puntatori

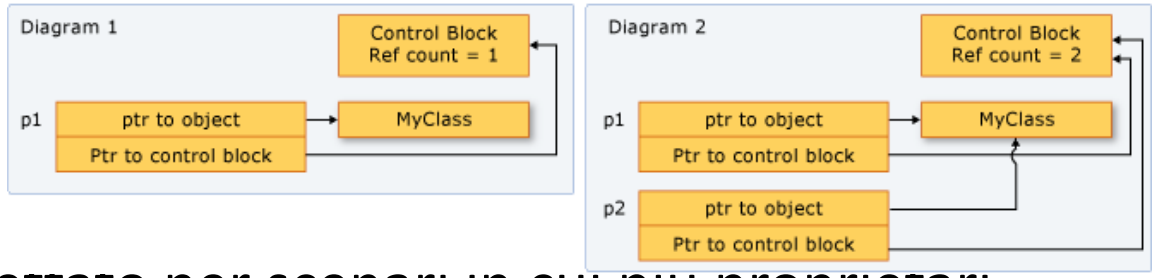
- unique_ptr: Consente esattamente un proprietario del puntatore sottostante. Se lo distruggo per uno viene distrutto per tutti
- shared_ptr Puntatore intelligente con conteggio dei riferimenti. Utilizzare quando si desidera assegnare un puntatore raw a più proprietari, ad esempio, quando si restituisce una copia di un puntatore da un contenitore ma si desidera mantenere l'originale.
- weak_ptr Puntatore intelligente per casi speciali da utilizzare insieme a shared_ptr.

Unique ptr



- Un unique_ptr non condivide il puntatore. Non può essere copiato in un altro , passato per valore a una funzione o usato in qualsiasi algoritmo della libreria standard C++

Shared pointer



- Il tipo shared_ptr è progettato per scenari in cui più proprietari potrebbero dover gestire la durata dell'oggetto in memoria.
- Dopo avere inizializzato un oggetto shared_ptr, è possibile copiarlo, passarlo come valore negli argomenti di funzione e assegnarlo ad altre istanze di shared_ptr.
- Tutte le istanze puntano allo stesso oggetto e condividono l'accesso a un "blocco di controllo" che incrementa e decrementa il conteggio dei riferimenti ogni qualvolta un nuovo oggetto shared_ptr viene aggiunto, esce dall'ambito o viene reimpostato. Quando il conteggio dei riferimenti arriva a zero, il blocco di controllo elimina la risorsa di memoria e se stesso.

C++

Additions not related to objects

Overview

- Additions and changes not related to objects
  - type bool
  - pass-by-reference & the Copy-Constructor
  - user-defined overloading
  - function template and class template
  - exception handling
  - ...

Type bool

- Represents boolean-values
- Conversion rules with the **`int`** type

```
bool b1, b2, b3;
int j, k;
b1 = 3*5; // b1 = true
b2 = 0; // b2 = false
j = b1; //  j = 1
j = b1 || b2; // j = 1
j = b1 && b2; // j = 0
b1 = j == 0; // b1 = true
```

Reference variables (1)

```
int a, *ptr_a;
int &ref_a = a;
```
// **ref_a** is an address, a *reference variable*
```
ref_a = 5;      // or a = 5
ptr_a = &ref_a; // or ptr_a = &a;
```

Reference variables (2)

A reference variable is similar to **a *const* pointer**

```
int a, *ptr_a;
int &ref_a = a;
ref_a = 5;
ptr_a = &ref_a;
```

```
int a, *ptr_a;
int * const ptr_a = &a;
*ptr_a = 5;
ptr_a = ptr_a;
```

This implies:

- a reference variable must be initialized when defined
- must refer always to the same variable, reassignment is not allowed

Call-by-reference (1)

```
int f(int& t_in) {
 t_in = 99;
…
}
```

A good style

```
int f(const int& t_in) {
 t_in = 99; // ERROR
…
}
```

Call-by-reference (2)

- Two possible realizations in C++
  - void doSomething(Data * data);
    - pointer-based
    - Advantages and drawbacks of pointer approach
  - void doSomething(Data & data);
    - Reference based
    - Non null-checking necessary

Return-by reference

```
int & g(int &x){
 return x
}
```

A C++ program can be made easier to read and maintain by using references rather than pointers. A C++ function can return a reference in a similar way as it returns a pointer. When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement (unless const).

```
g(j) = 99   ... j = 99
const int & h(int x);
…
h(j) = 99; // ERROR
```

Summary

- Variables hold values
  - `int v = 5`
- Pointers hold addresses of variables
  - `int* p = new(int);`
  - `*p = 5;`
- References refer to contents of another variable
  - `int& r = a;`

The Copy Constructor

- **`ChewingGum(const ChewingGum& rhs);`**
- Default copy constructor
  - Automatically generated if not present
  - Produces a complete **shallow** copy of the passed object (e.g. pointers are copied equal)
- User-defined copy constructor
  - Can take arbitrary mesaures to provide a copy of the rhs object (e.g. deep copies)

The Assignment Operator

```
ChewingGum& operator
      =(const ChewingGum& rhs);
```

- Sets an object to be a copy of a passed object
- Default behavior: shallow copies
- Example
  - ChewingGum g1;
  - ChewingGum g2 = g1;

The Assignment Operator & Inheritance

- When assigning to a base class, the = is used.
- Example

```
class A{}; class B: public A{}
A a;
B b;
```

`a = b` --> assignment of a is used.

  - Note that fields of B that are not in A are not copied (slicing)

The Copy Constructor and the Assignment Operator

- copy constructors and assignment operators
  - automatically generated in each class
  - no inheritance

```
class Employee {
  //...
  Employee(const Employee&);
  Employee& operator=(const Employee& );
};

Manager m("Homer", 3);
Employee e = m;        // Slicing!!
```

The Copy Constructor and dynamic memory

- **always** declare a user-defined **copy constructor** for classes with dynamically allocated memory
- the default implementation leads to
  - undefined behaviour (probably an access violation)

An example (1)

```
#include <iostream.h>
#include <string.h>
class Message {
   char *subject;
   char *message;
   //A function to initiale data members
   init_message(const char *,const char *);
 public:
   //A constructor
   Message(const char *, const char * = "");
   //The copy-constructor
   Message(const Message & m);
   //Overloading of the assignment operator =
   const Message& operator=(const Message &);
   //The destructor
   ~Message(;
};
```

An example (2)

```
//A function to initiale data members
Message::init_message(const char *s, const char *m){
    subject = new char [strlen(s)+1];
    strcpy(subject,s);
    message = new char [strlen(m)+1];
    strcpy(message,m);
}

//A constructor
Message(const char *s, const char * m){
   init_message(s,m);

//The destructor
~Message(){
   delete subject;
   delete message;
}
```

An example (3)

```
//The copy constructor
Message::Message(const Message & m){
   init_message(m.subject,m.message);
}


//Overloading of the assignment operator =
const Message& Message::operator=(const Message & m){
   // always check for self-assignment
   if (this == &m) return *this;
   // clean current object
   delete subject;
   delete message;
   init_message(m.subject,m.message);
   return *this; //the left element is returned
}
```

Assignment sequences

- C++ allows for

```
int a, b, c, d;
a = b = c = d = 5;
```

- Objects should allow this as well
- assignment operator needs to return a reference to `(*this)`

References (1)

- C++ as a language and programming guidelines
  - Stroustrup, B. (1999). *The C++ Programming Language*, Addison-Wesley.
  - Meyers, S. (1998). *Effective C++,* Addison-Wesley
  - Meyers, S. (1995). *More Effective C++,* Addison-Wesley
  - Meyers, S. (2000). *Effective STL*, Addison-Wesley
  - Alexandrescu, A. (2002). *Modern C++ Design*, Addison-Wesley

References (2)

- Process memory layout, etc.
  - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*, Chapters 3 and 6
  - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, Chapter 2
  - Santa Cruz Operation, Inc. (1997), *System V Application Binary Interface Intel386(tm) Architecture* Processor Supplement, Fourth Edition

String in C++

- C++ provides a simple, safe alternative to using char*s to handle strings. The C++ string class, part of the std namespace, allows you to manipulate strings safely.
- Declaring a string is easy:

```
using namespace std;
string my_string;
```

or

```
std::string my_string;
```

- Vedi syllabus

References (3)

- **C++ Applications** by the creator of C++
- Bjarne Stroustrup
  - http://www.research.att.com/~bs/applications.html