Contemporary C++: Learning Modern C++ in a Modern Way

آلفا ALPHA

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 8/24

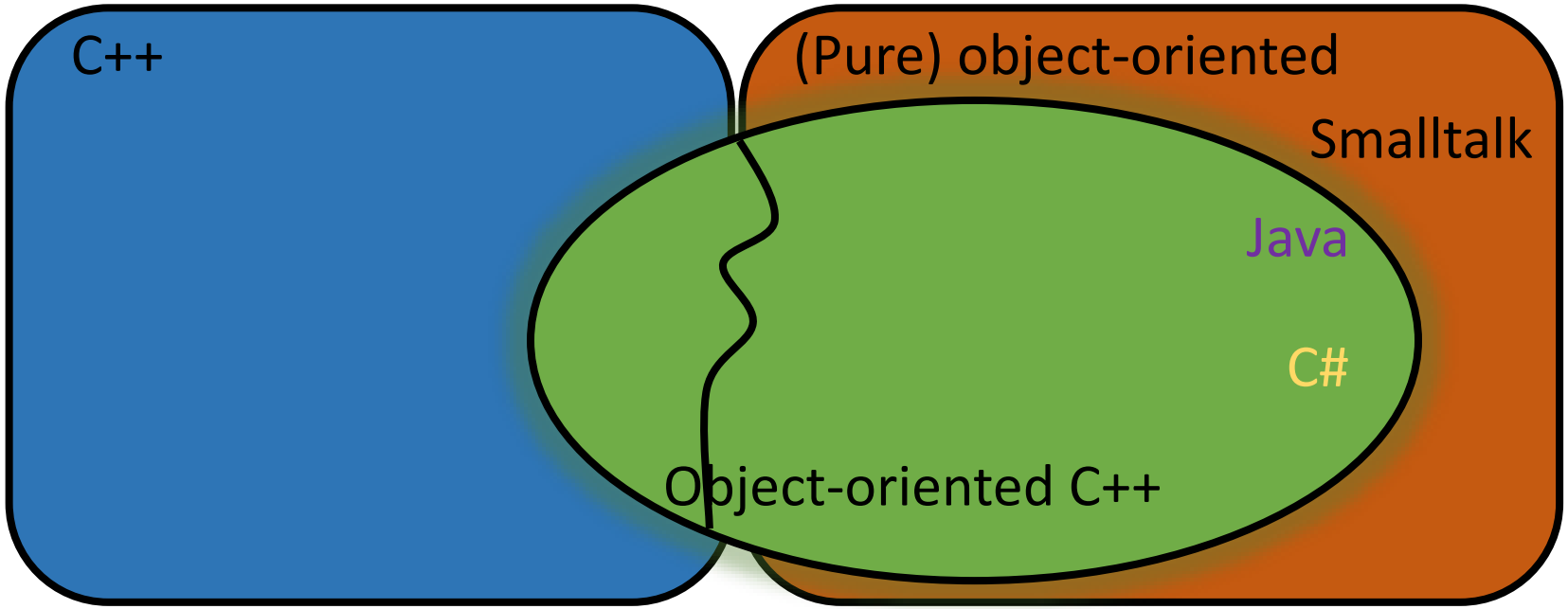
Session 8. Classes: fundamental concepts (part I)

- Fundamental types vs. User-defined types
- The C++ classes classification: Concrete and Handle classes
- Classes: Data members and Member functions
- Access control: public and private
- Introduction to memory allocation and free store
- Special member functions: Constructors
- Objects
- Const member functions
- Static members: data members and member functions
- Classes vs. structs
- the this pointer
- Q&A

150 min (incl. Q & A)

C++ and Object-orientation

- C++ is a multi-style/paradigm language.



- The concepts and principles of class design from the point of pure object-oriented is beyond the scope of this course.

Types and user-defined types

Types
- Fundamental types or Built-in types: bool, char, int, double
- User-defined types

- A type is a concrete representation of a concept.

  Ex. C++ built-in type float ≈ real numbers

- The designer of a general-purpose programming language cannot foresee the detailed needs of every application.

C++ class ⟶ Creating new types

> *"Those types are not "abstract"; they are as real as* `int` *and* `float`*."*
>
> *- Doug McIlroy*

- Ideally, such types should not differ from built-in types in the way they are *used*, only in the way they are *created*.

User-defined types: benefits

- C++ is a type-full programming language

- Defining a new type:

  - Separate implementation details from interface

  - Better expressiveness: Problem domain vs. Solution domain

  - Maintainability: Easier to understand and easier to modify programs

Sentence, String, Paragraph, … → Test processing

Point, Line, Rectangle, Circle, Shape, … → Graphics

Button, Label, Textbox, Combobox, Icon, … → GUI

Explosion, Orc warrior, … → Video games

Classes: definitions

Class

C++ standard working draft:

A class is a type.

Simula → C++

A Class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform.

GOF: Design Patterns, Elements of Reusable Object-Oriented Software

Bjarne Stroustrup:

A class is the mechanism for representing concepts.

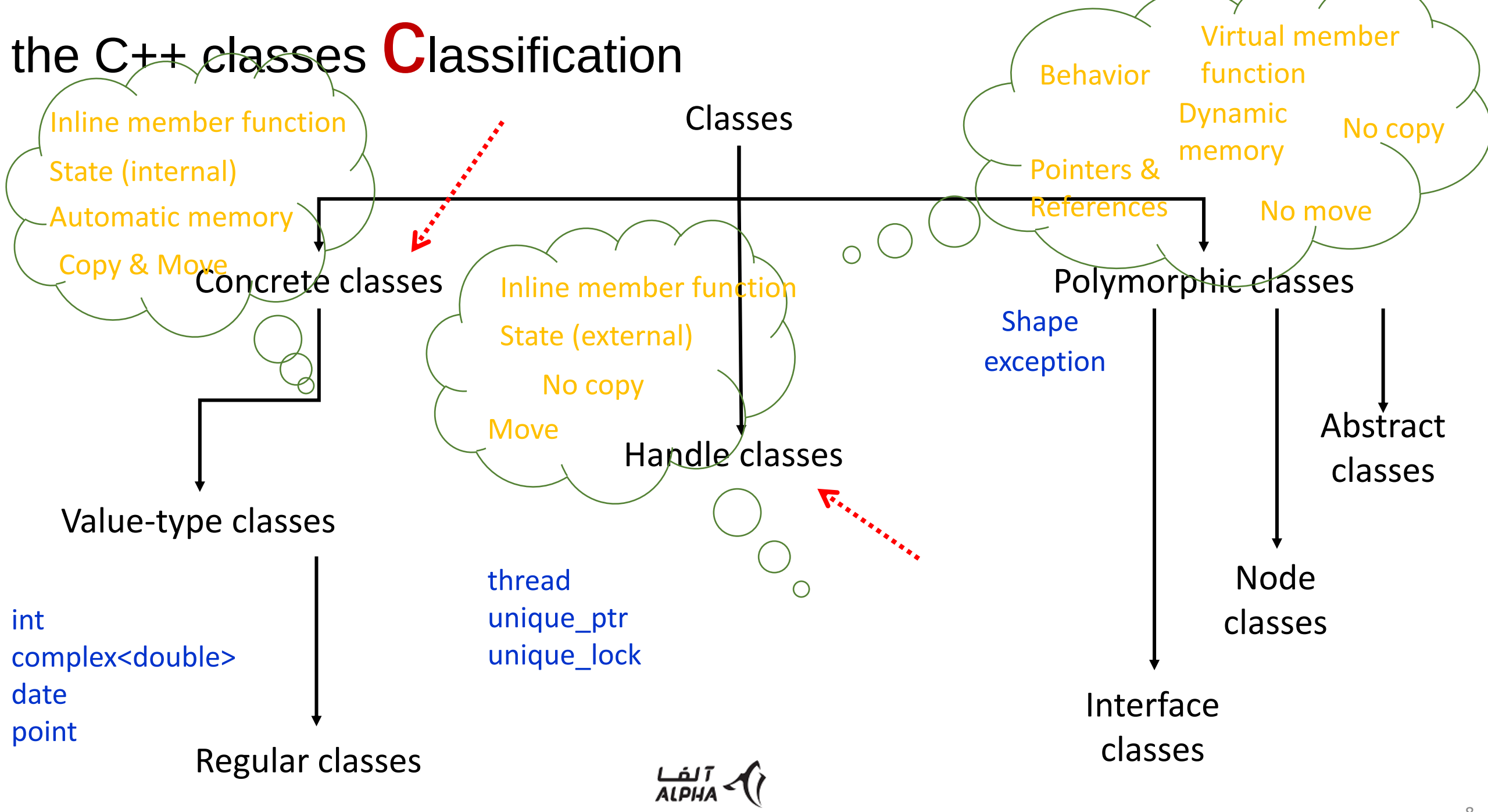
A class is a set of objects that share a common structure and a common behavior.

Grady Booch: Object-Oriented Analysis & Design with Applications

Classes: benefits

- A program that provides types that closely match the concepts of the application tends to be:
  - easier to understand
  - easier to reason about
  - easier to modify them
- A well-chosen set of user-defined types makes:
  - a program more concise
  - many sorts of code analysis feasible
- Fundamental idea: separating incidental details of the implementation from essential properties.
- Better expressiveness
- Software reuse

the C++ classes Classification



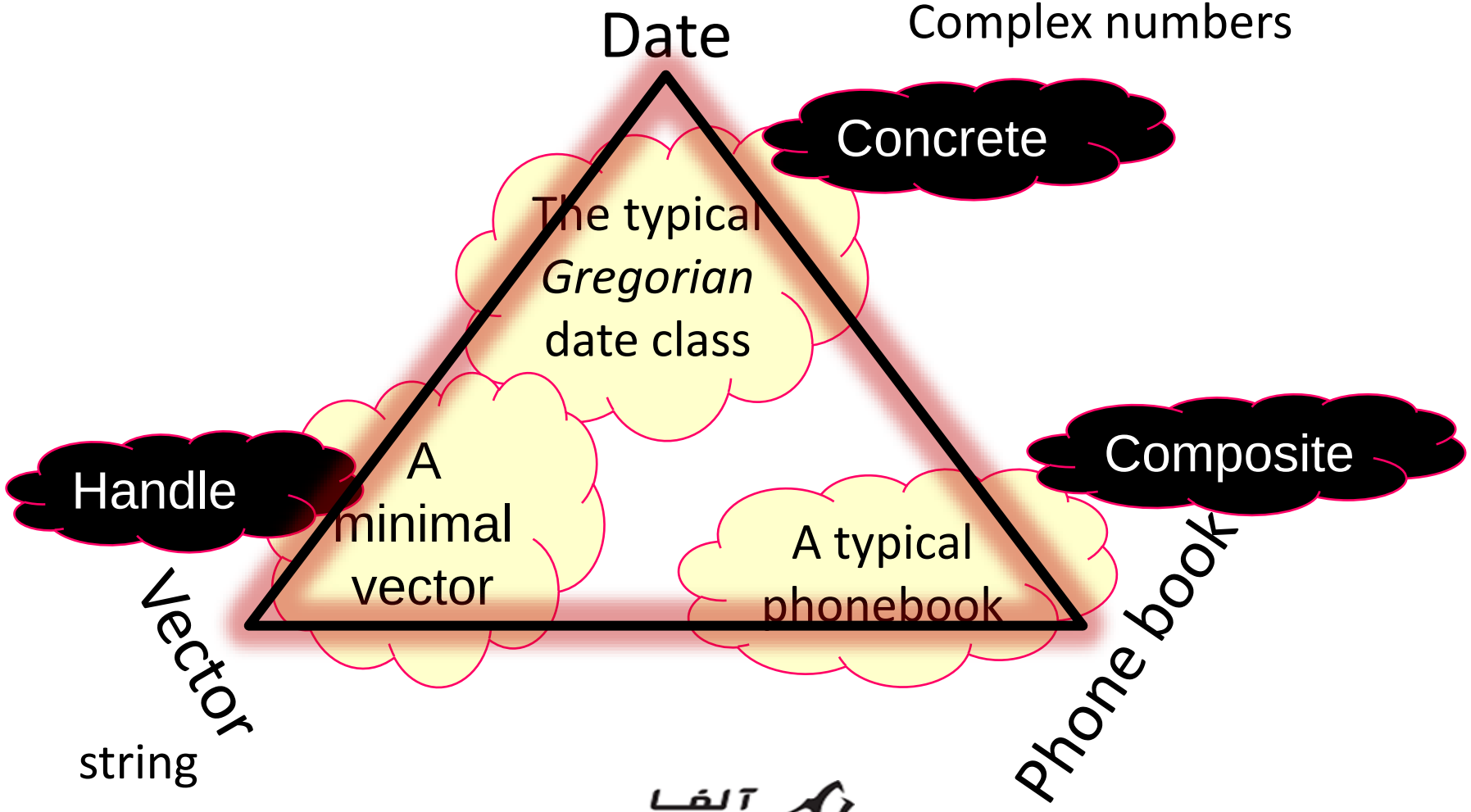
the Classes we want to present

- The iterative and incremental design
- The gradual refinement process

Rational number

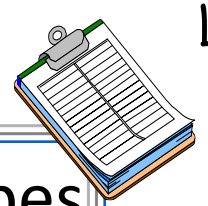
Point

Complex numbers



Concrete classes: introduction

- a concrete type resembles a built-in type.



Language-technical rule

> Provide as good support for user-defined types as for built-in types.

```
void f()
{
  int i; // int variable declaration/definition
  int j = i; // initialize an int variable
  j = i; // assign new value to already defined int

  int* pi = &i; // take the address of int variable
  int* p = new int(0); // make –unnamed- int in heap
  // …
  delete p; // release the memory allocated by var.
}
```

The Date Class

1. Separated states and behaviors

- Data members

```
struct Date { // representation
  int day, month, year;
}; // declaration

// operations
void init_date(Date& d, int, int, int); // initialize d
void add_year(Date& d, int); // add n years to d
void add_month(Date& d, int n); // add n months to d
void add_day(Date& d, int n); // add n days to d
// …
```

```
void init_date(Date& dd, int d, int m, int y)
{
  dd.Day = d; dd.Month = m; dd.Year = y;
}
// other functions
```

Member functions

Object inversion

```
struct Date {
  int day, month, year; // data members
  // member functions
  void init(int, int, int);
  void add_year(int);
  void add_month(int n);
  void add_day(int n);
  // …
};
```

- struct is a kind of class.
- There is no explicit connection between the data type and these functions.

2. Member Functions

```
class X {
   data members
   member functions
};
```

Member → Data Member

Member → Member Function

- Old terminology: Object inversion

The Vector Class

- The very 1st try: fixed-size and struct-based vector

The Date Class cont.

- Function declared within a class definition are called *member functions*.
- Scope resolution operator: ::

```
void Date::init(int d, int m, int y)
{
  day = d; month = m; year = y;
}
// use date
Date birthday; // uninitialized object
Date today; // another object
Date unix_time_epoch; // Unix epoch time
today.init(1, 11, 2018); // initialize today
unix_epoch_time.init(1, 1, 1970); // initialize epoch time
```

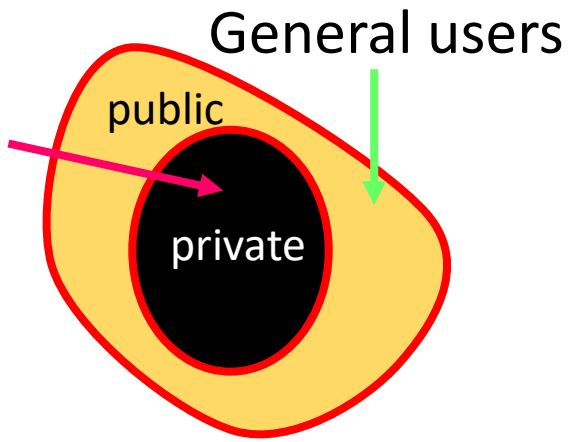
3. Access Control and Information Hiding

```
// date.h
class Date {
  // data members are encapsulated
  int Day, Month, Year;
public:
  // member functions
  void init(int, int, int);
  void add_year(int);
  void add_month(int n);
  void add_day(int n);
  // …
};
```

Implementation

Interface

own member functions and friends



class declaration/definition

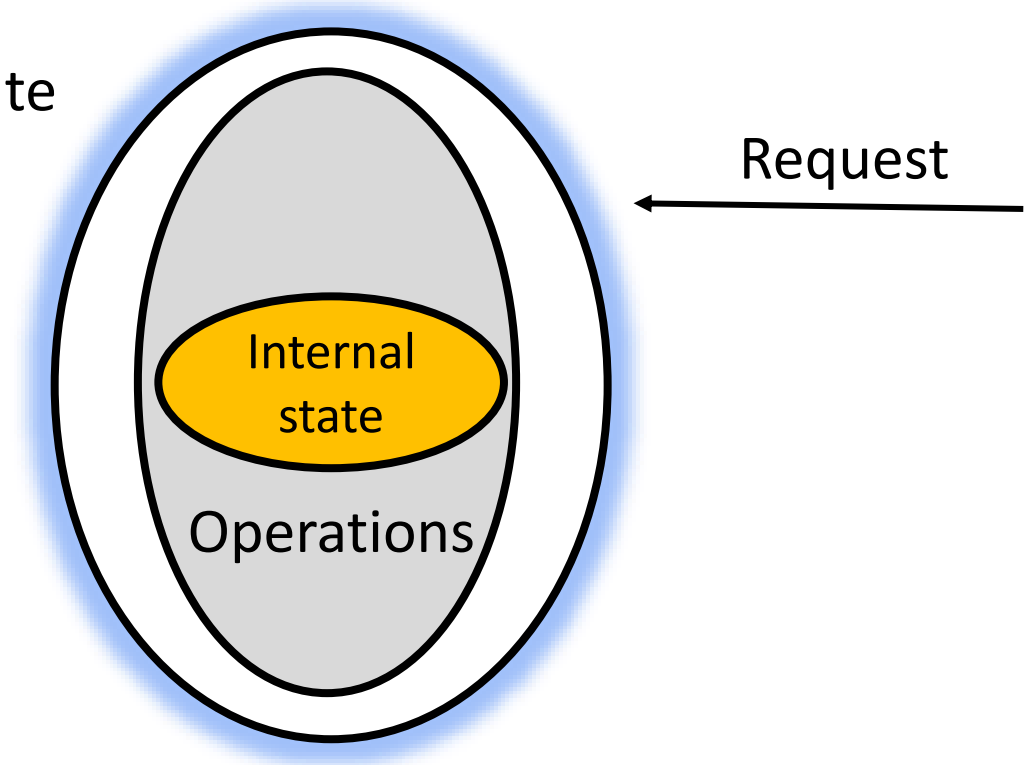
```
class X { … };
```

- The class definition comes in a header file

→ component fundamental rule

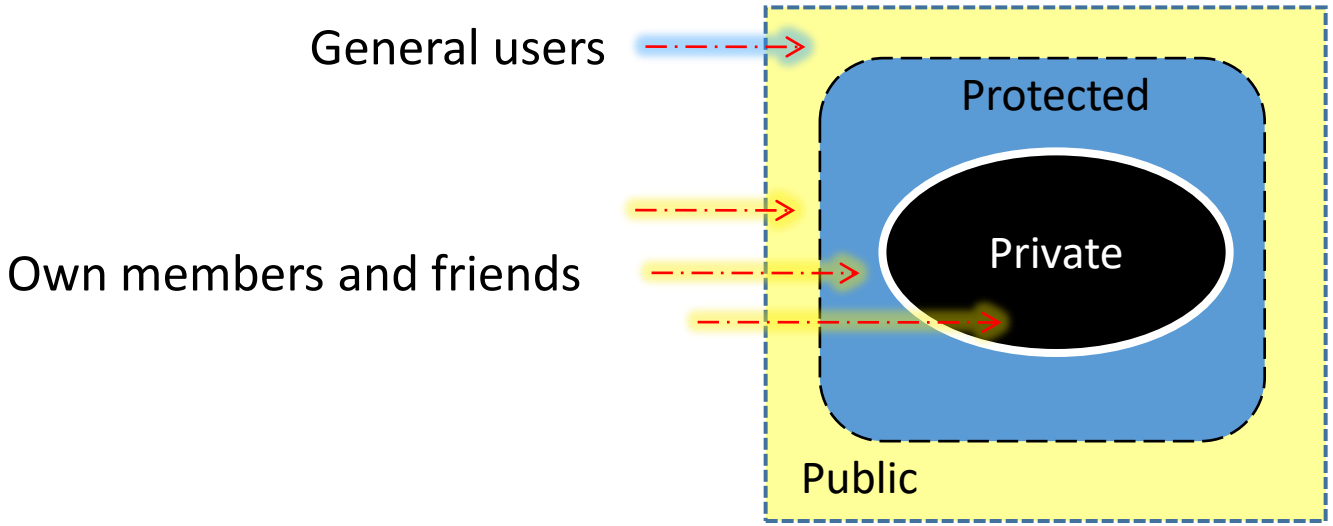
Class/object Design: fundamental model

- The egg model
  - Requests are the *only* way to get and object to execute an operation.
  - Operations are the *only* way to change an object's internal data (state).
- Example: The car, the stack



Access control

- A member of a class can be *private*, *protected*, or *public*.
- If it is *private*, its name can be used only by members and friends of the class in which it is declared.
- If it is *public*, its name can be used anywhere without access restriction.



- Access to members vs. Member visibility

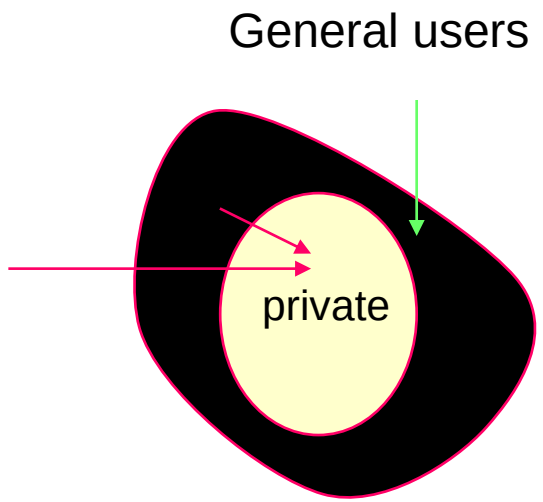
Access Control, class scope and resolution operator

- (Non-inline) member functions definitions comes in source file.

```
// date.cpp
void Date::add_year(int n)
{
    year += n;
}
void Date::add_month(int m)
{
    // …
}

void Date::add_day(int d)
{
    // …
}
```

Scope resolution operator



- Nonmember functions are barred from using private members.

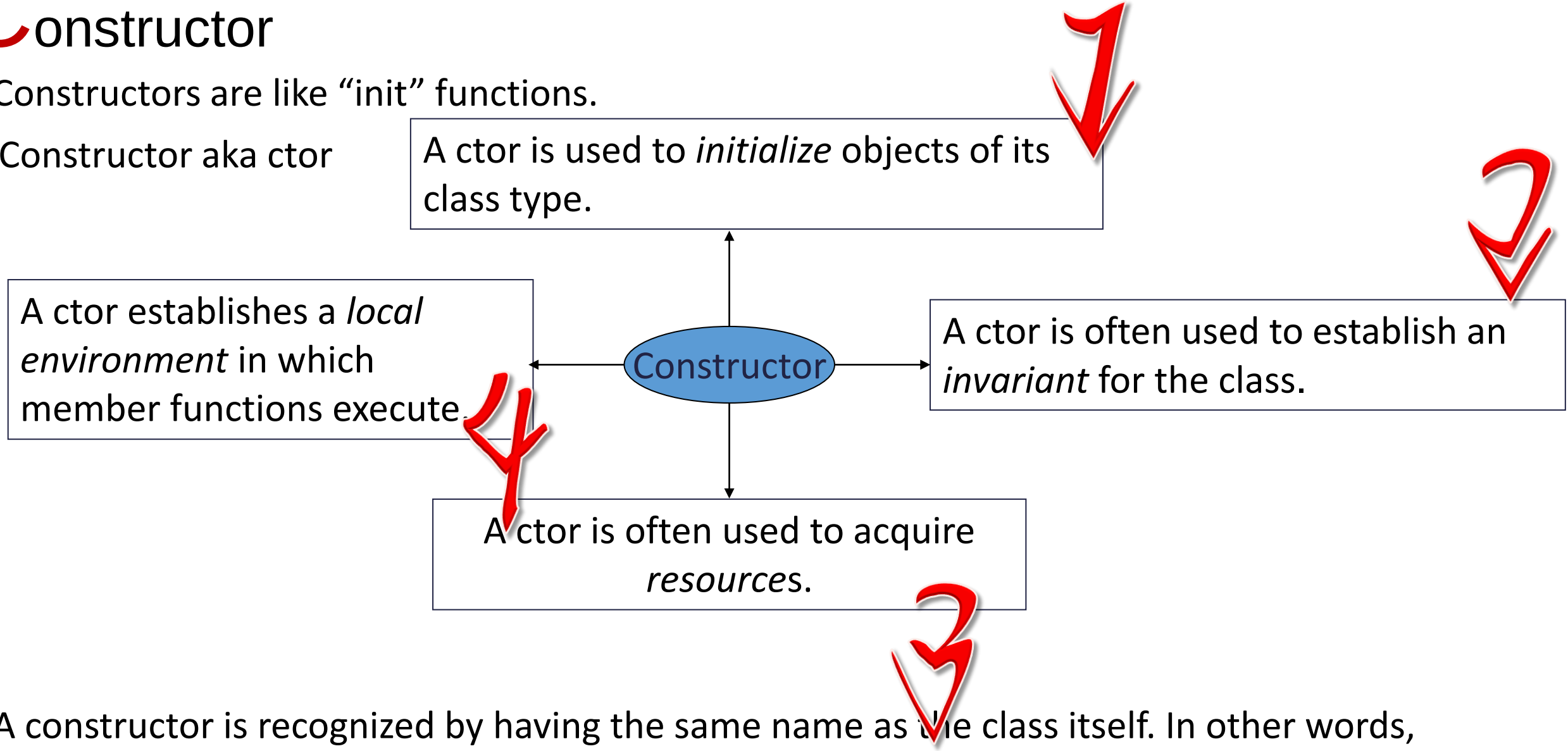
```
void f()
{
  Date d;
  d.init(3, 12, 1967);
  d.day++; // error: day is private
  Date dd;
  dd.init(36, 13, 1380); // invalid date
}

void Hijack(Date& d)
{
  d.month--; // error: can't access to private member
}
```

- Local scope
- Function Scope
- Class Scope: A class is a namespace containing its members.
- Namespace Scope

Constructor

- Constructors are like “init” functions.
- Constructor aka ctor



- A constructor is recognized by having the same name as the class itself. In other words, constructors do not have names. Constructors do not return value.

Constructor: Initializing objects

```
class Date {
  // …
public:
  // ctor declaration
  Date(int, int, int); // constructor
};
Date d1 = Date(1, 1, 2019);
Date d2(1, 1, 2019);
Date d3; // error
```

- Multiple constructors: Several ways to initialize an object
- Overloaded constructors: Function name overloading

```
class Date {
  // …
  Date(int, int, int); // day, month, year
  Date(int, int); // day, month, today's year
  Date(int); // day, today's month and year
  Date(); // default Date: today (global)
  Date(const char*); // Date in string rep.
};
```

```
class Date {
  // …
  Date(int = 0, int = 0, int = 0); // day, month, year
  Date(const char*); // Date in string rep.
};
```

- Default arguments

Date class: constructor

```
Date today(7, 7, 1387); // today is a global object
Date::Date(int d, int m, int y) // use a global object
{
  Day = (d ? d :today.Day);
  Month = (m ? m : today.Month);
  Year = (y ? y : today.Year);
}
```

- Bad design: The `date` class is dependent on global `today` object.

In-class function definition

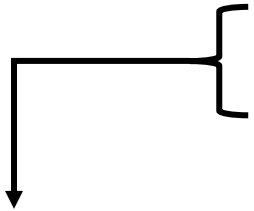
- A member function defined within the class definition – rather than simply declared there – is taken to be an inline member function.

```
class Date { // Perfectly good C++ code but confusing
public:
  int day() const { return Day; } // return Date::d
  // ...
private:
  int Day, Month, Year;
};
```

```
class Date { // better
public:
  int day() const;
private:
  int Day, Month, Year;
};
inline int Date::day() const { return Day; }
```

Another example: Point

- Point
  - two coordinates: x, y
  - two constructors
  - Move
- Concrete class

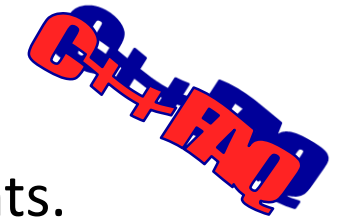


In-class function definition

```
class Point { // 2D point concept
public: // Constructor(s)
  Point(int xx, int yy) { x = xx; y = yy; }
  Point() { x = y = 0; }
  void move(int xx, int yy) { x = xx; y = yy; }
private: // Implementation
  int x, y;
};
```

Default constructor

- A default constructor, is a constructor that *can* be called without supplying an argument.
- If a user has declared a default constructor, that one will be used.



Q *Is the default constructor for Fred always Fred::Fred()?*

A No. A "default constructor" is a constructor that *can be called* with no arguments. One example of this is a constructor that takes no parameters:

```
class Fred {
public:
        Fred();   // Default constructor: can be called with no args
        ...
};
```

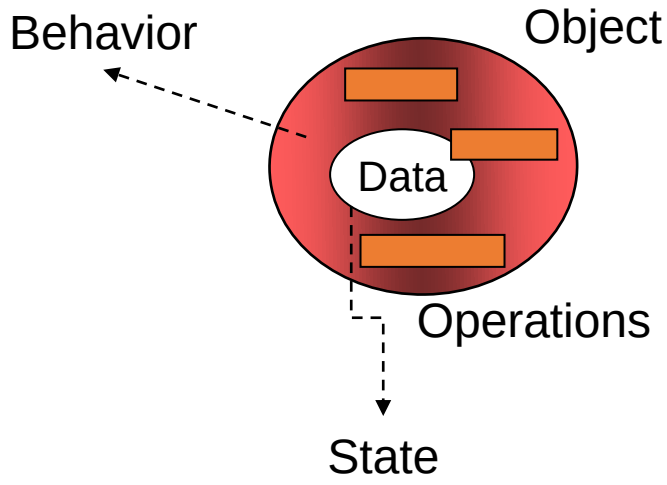
Another example of a "default constructor" is one that can take arguments, provided they are given default values:

```
class Fred {
public:
        Fred(int i=3, int j=5);   // Default constructor: can be called with no args
        ...
};
```

Examining State: Const member functions

- No modifying member functions

```
class Date {
  int Day, Month, Year;
public:
  int day(); // note: "day" not "Day"
  int month();
  int year();
};
int Date::day() { return Day; }
int Date::month() { return Month++; } // change state by mistake
```



- Constant objects

```
const char blank = ' ';
blank = '\0'; // error
```

```
const Date my_birthday(1, 5, 1350);
```

- Constant functions do not modify the state of objects of a given *type*.

```
class Date {
  int Day, Month, Year;
public:
  // functions for examining the Date (Selectors)
  int day() const  { return Day; }
  int month() const { return Month; }
  int year() const; // just declaration
};
```

Const member functions cont.

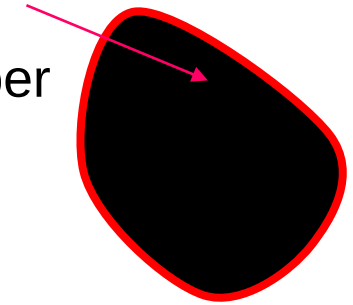
```
inline int Date::year() const {
  return ++Year; // error: attempt to change member value in const function
}
```

- When a const member function is defined outside its class, the const suffix is required.

```
inline int Date::year() {
  return Year; // error: const missing in member function definition
}
```

- Constant objects

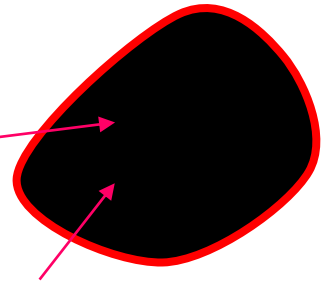
constant member functions



- Non-constant objects

Non-constant member functions

constant member functions



Classes without Virtual functions

- … I always maintained a clear view of what an object looked like in memory.

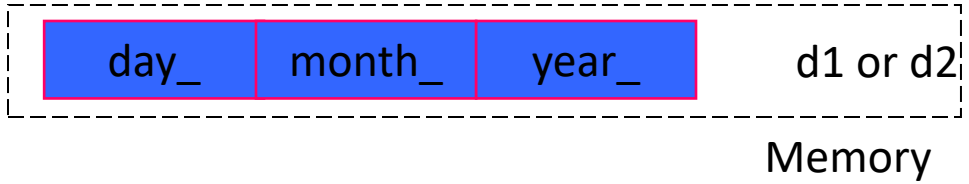
  - Bjarne Stroustrup

- A class without a virtual function is a simple C struct.
- Each class object maintains its copy of the class data members.

```
class Date {
  int day_, month_, year_;
public:
  Date(int = 0, int = 0, int = 0);

  void add_year(int);
  void add_month(int);
  void add_day(int);
  // Other member function(s)
} d1;
```

```
struct Date {
  int day_, month_, year_;
} d2;
```



- Later members have higher addresses within a class object.

- A class without a virtual function requires exactly as much space to represent as a struct with the same data members.
- A compiler may add some “padding” between and after the members for alignment.

Concrete classes: characteristics

- It allows us to

  - place objects of concrete types on the stack, in statically allocated memory, and in other objects.

  - refer to objects directly.

  - initialize objects immediately and completely.

  - copy objects.

  - move objects.

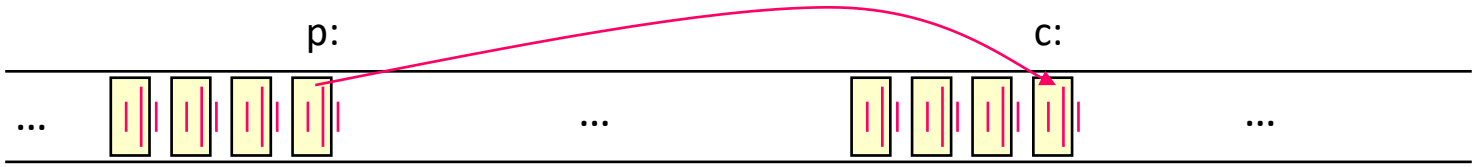

Memory management

the C Memory model

- Programmer-controlled memory management

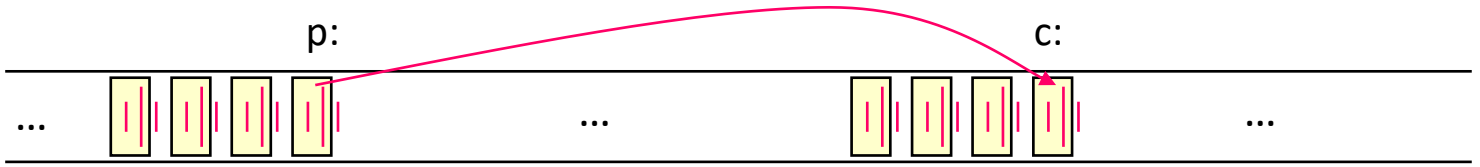
the C Memory model

- Programmer-controlled memory management
- A typical machine has an array of consecutively numbered or addressed memory cells.



the C Memory model

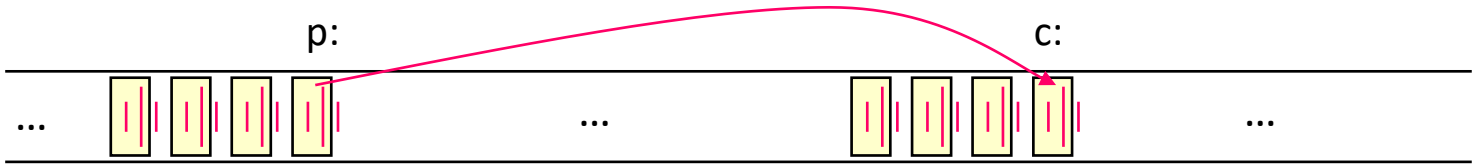
- Programmer-controlled memory management
- A typical machine has an array of consecutively numbered or addressed memory cells.



- A C/C++ pointer is simply an address of a memory locations.
- A C/C++ array is simply a sequence of memory locations.

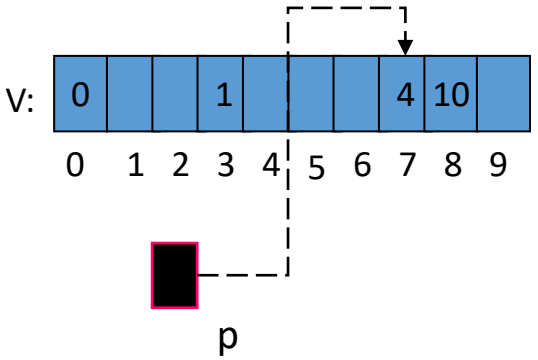
the C Memory model

- Programmer-controlled memory management
- A typical machine has an array of consecutively numbered or addressed memory cells.



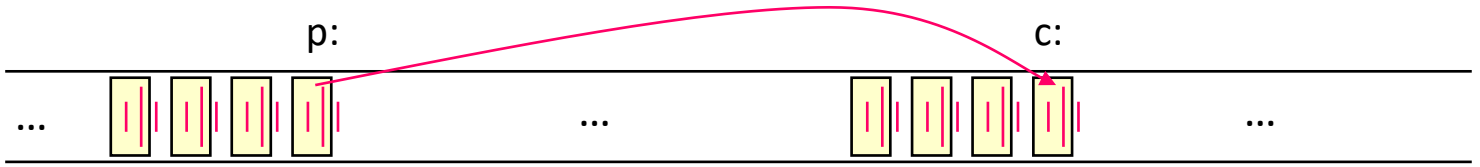
- A C/C++ pointer is simply an address of a memory locations.
- A C/C++ array is simply a sequence of memory locations.

```
int v[10]; // an array of 10 ints
int* p; // p is a pointer to an int
p = &v[7]; // assign the address of v[7] to p
*p = 4; // write to v[7] through p
*(p++) = 10;
*v = 0; // the name of array points to first element
```



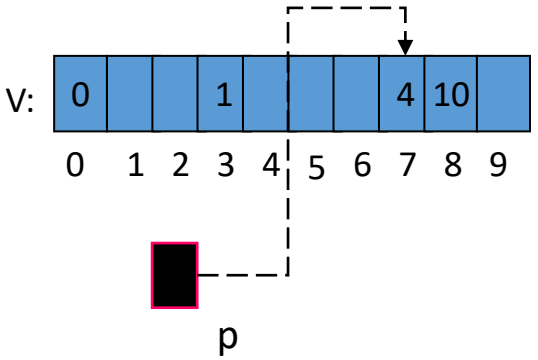
the C Memory model

- Programmer-controlled memory management
- A typical machine has an array of consecutively numbered or addressed memory cells.



- A C/C++ pointer is simply an address of a memory locations.
- A C/C++ array is simply a sequence of memory locations.

```
int v[10]; // an array of 10 ints
int* p; // p is a pointer to an int
p = &v[7]; // assign the address of v[7] to p
*p = 4; // write to v[7] through p
*(p++) = 10;
*v = 0; // the name of array points to first element
```



- Pointer arithmetic
- No initialization, no bound checking

Programmer-controlled memory management

- C Memory model: Three kinds of memory
- The concept of storage duration

Programmer-controlled memory management

- C Memory model: Three kinds of memory
- The concept of storage duration
- Automatic memory: This kind of memory is *automatically* created and destroyed. The storage for these objects lasts until the block in which they are created exits.

    - Function arguments, local variables

Programmer-controlled memory management

- C Memory model: Three kinds of memory
- The concept of storage duration
- Automatic memory: This kind of memory is *automatically* created and destroyed. The storage for these objects lasts until the block in which they are created exits.

  - Function arguments, local variables

- Static memory: An object allocated in static memory is constructed once and *persists* to the end of the program.

  - The storage for these objects shall last for the duration of the program.

  - Global and namespace variables, static class members, static variables in functions,

  - the keyword static

Programmer-controlled memory management

- C Memory model: Three kinds of memory
- The concept of storage duration
- Automatic memory: This kind of memory is *automatically* created and destroyed. The storage for these objects lasts until the block in which they are created exits.

  - Function arguments, local variables

- Static memory: An object allocated in static memory is constructed once and *persists* to the end of the program.

  - The storage for these objects shall last for the duration of the program.

  - Global and namespace variables, static class members, static variables in functions,

  - the keyword static

Free store: Objects can be created dynamically during program execution using new, and destroyed using delete. The storage for these objects shall last for the duration of the program, or executing delete operator.

  - the keywords: new and delete

  - Free memory = dynamic memory = heap

New and delete operators

New and delete operators

- You *request* memory “to be allocated” “on the free store” by the new operator.
- The new operator returns a pointer to the allocated memory.

New and delete operators

- You *request* memory “to be allocated” “on the free store” by the new operator.
- The new operator returns a pointer to the allocated memory.
- You *request* sequence of memory by the new [] operator.

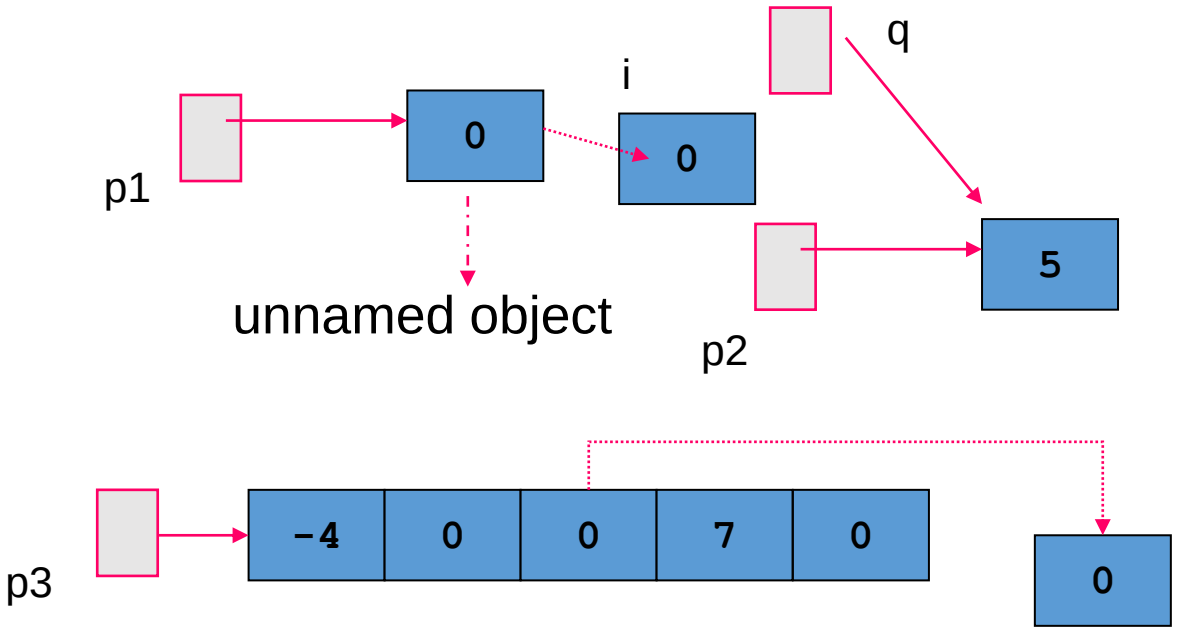
New and delete operators

- You *request* memory “to be allocated” “on the free store” by the new operator.
- The new operator returns a pointer to the allocated memory.
- You *request* sequence of memory by the new [] operator.
- You *release* the memory by the delete and delete [] operators.

New and delete operators

- You *request* memory “to be allocated” “on the free store” by the new operator.
- The new operator returns a pointer to the allocated memory.
- You *request* sequence of memory by the new [] operator.

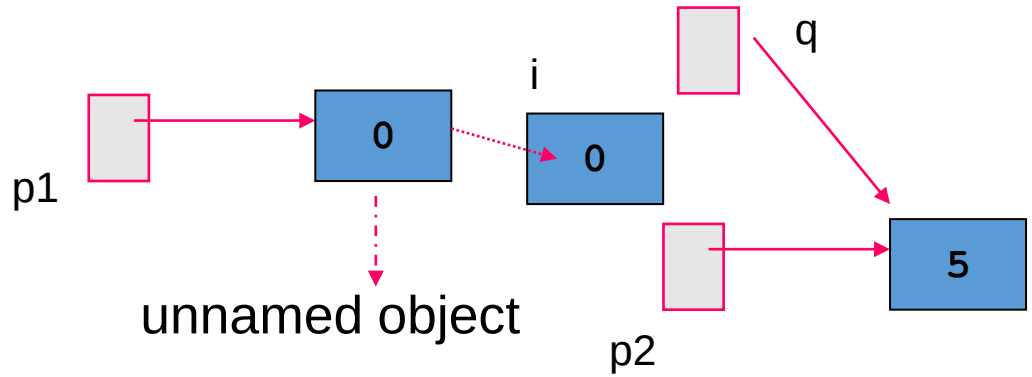
- You *release* the memory by the delete and delete [] operators.

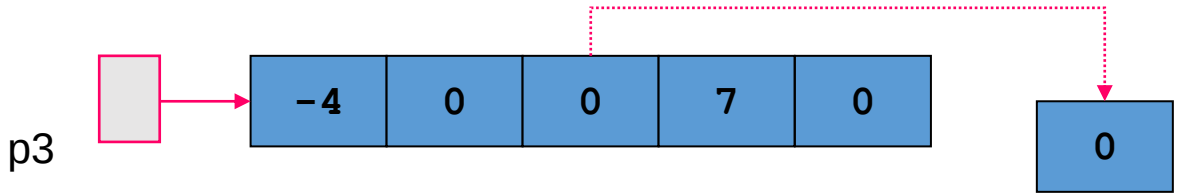



```
int* p1 = new int;
int* p2 = new int(5);
int* q = p2;
int i = *p1;
++*p1; // means ++(*p1)
(*p2)--; // () is necessary
int* p3 = new int[5]; // get (allocate) 5 ints and
                      // initialized to 0
int x = p3[2];
p3[0] = -4;
p3[3] = 7;
// from last page
delete p1; // deallocate an individual object.
           // return memory to OS
delete p2; // deallocate an individual object.
delete [] p3; // deallocate an array
```

New and delete operators

- You *request* memory “to be allocated” “on the free store” by the new operator.
- The new operator returns a pointer to the allocated memory.
- You *request* sequence of memory by the new [] operator.
- You *release* the memory by the delete and delete [] operators.






```
int* p1 = new int;
int* p2 = new int(5);
int* q = p2;
int i = *p1;
++*p1; // means ++(*p1)
(*p2)--; // () is necessary
int* p3 = new int[5]; // get (allocate) 5 ints and
                      // initialized to 0
int x = p3[2];
p3[0] = -4;
p3[3] = 7;
// from last page
delete p1; // deallocate an individual object.
           // return memory to OS
delete p2; // deallocate an individual object.
delete [] p3; // deallocate an array
```

- Memory leak
- Garbage collector

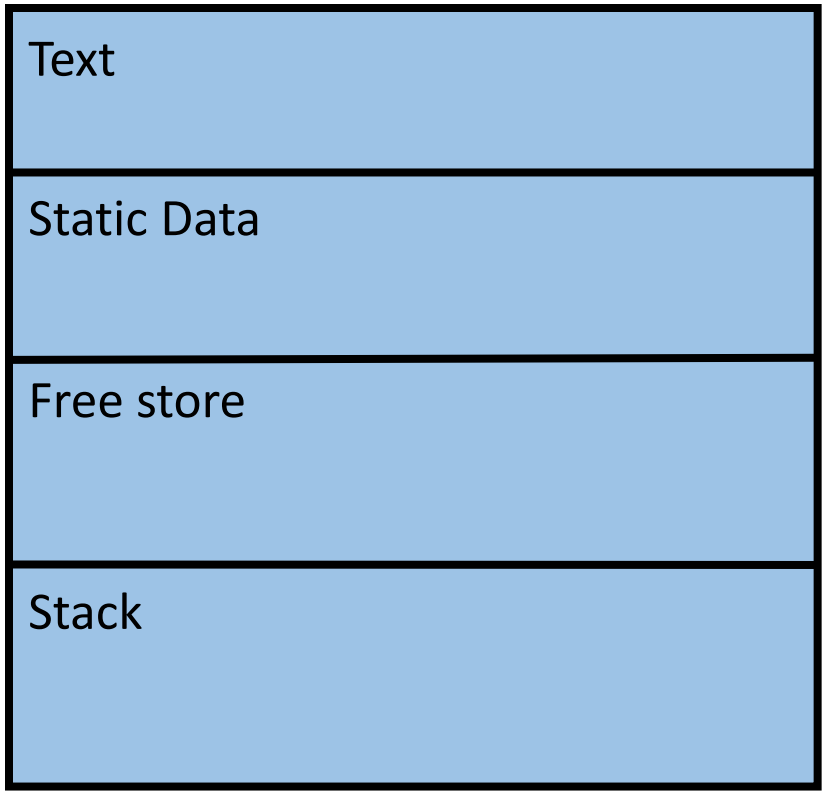
Process structure & three kinds of memories

Process structure & three kinds of memories

- Typical structure of UNIX/Windows process

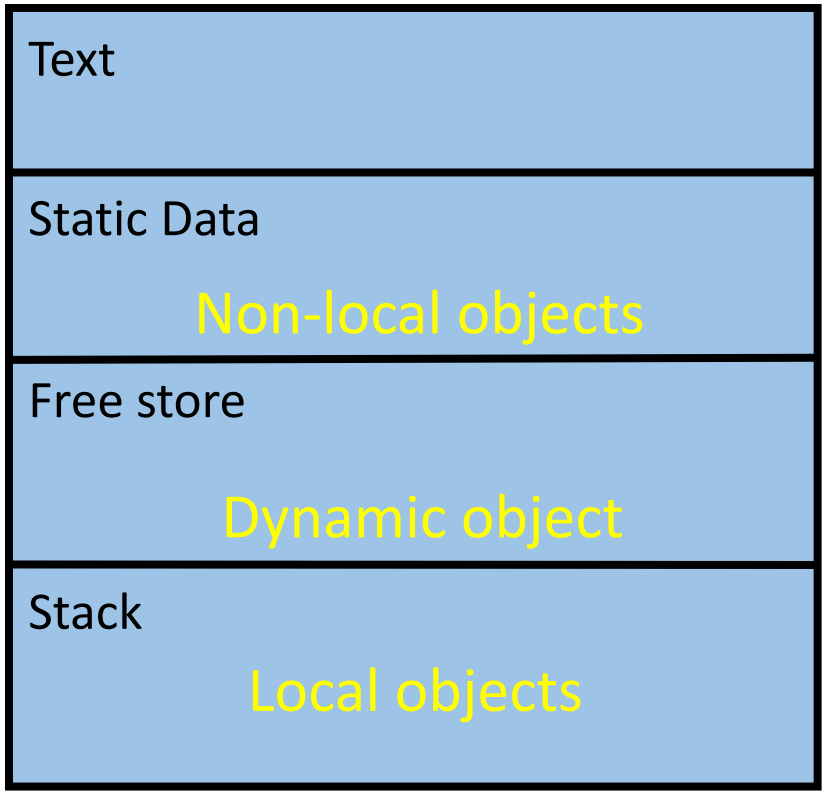
Process structure & three kinds of memories

- Typical structure of UNIX/Windows process



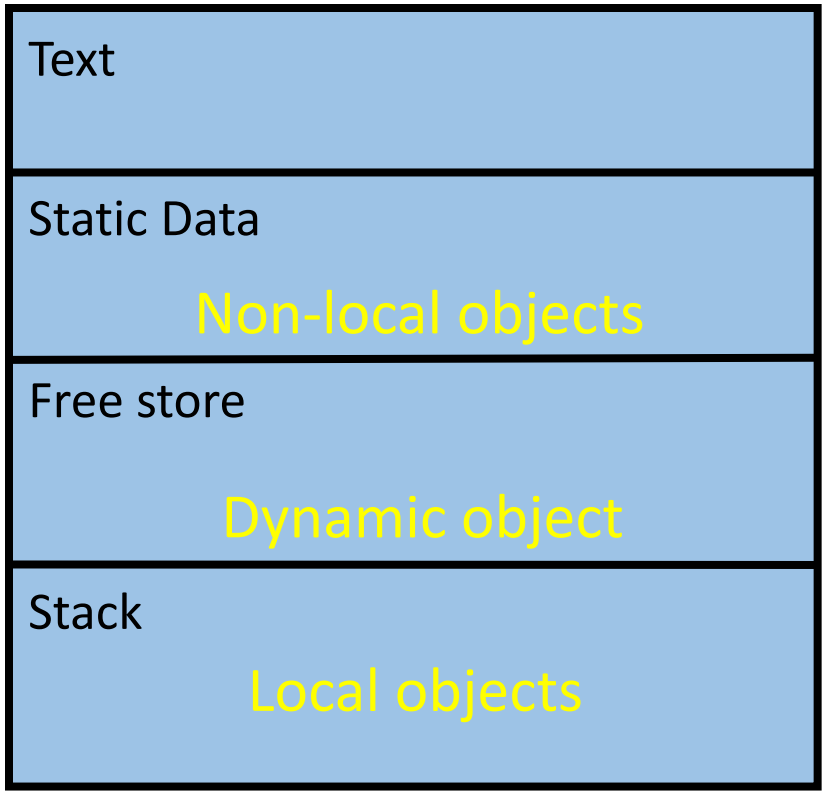
Process structure & three kinds of memories

- Typical structure of UNIX/Windows process



Process structure & three kinds of memories

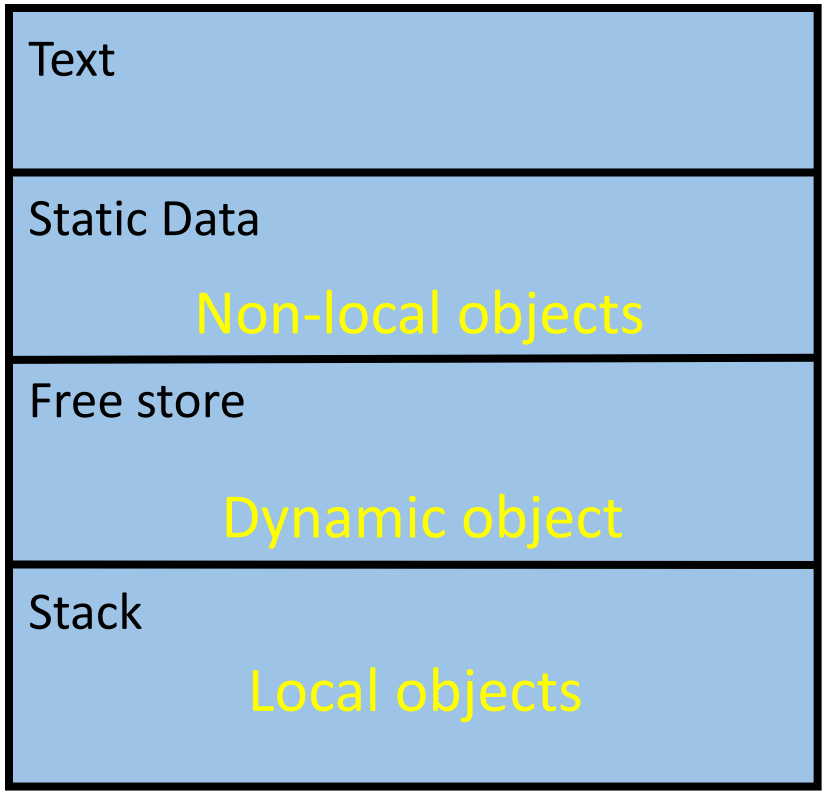
- Typical structure of UNIX/Windows process




```
void f(auto int i, double j)
{
  string s = "ABC";
  vector<int> v(10);
  // …
}
```

Process structure & three kinds of memories

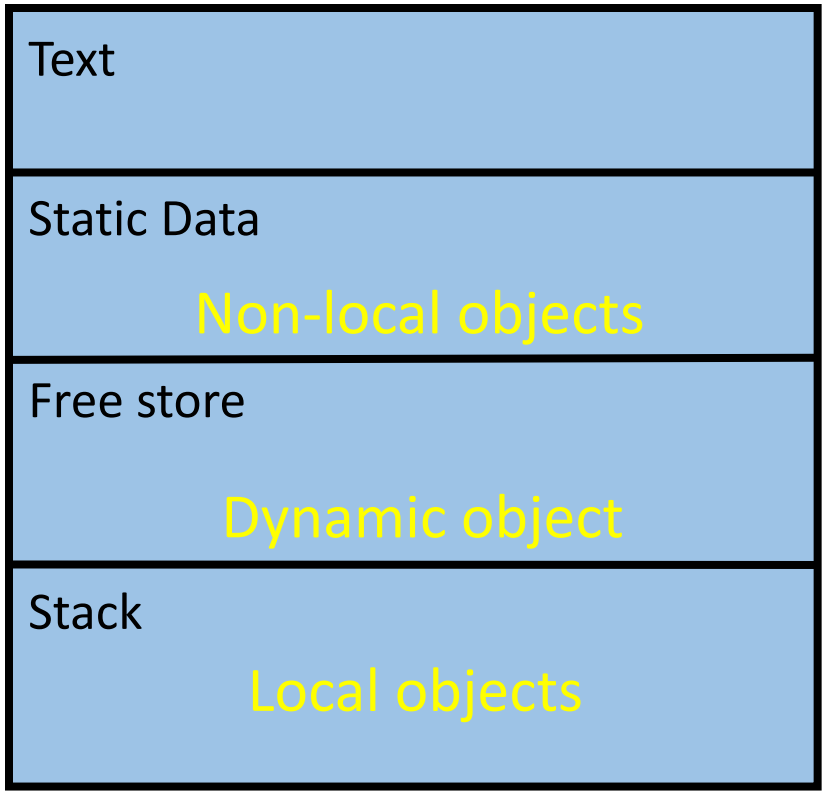
- Typical structure of UNIX/Windows process




```
void f(auto int i, double j)
{
  string s = "ABC";
  vector<int> v(10);
  // …
}
```

Process structure & three kinds of memories

- Typical structure of UNIX/Windows process

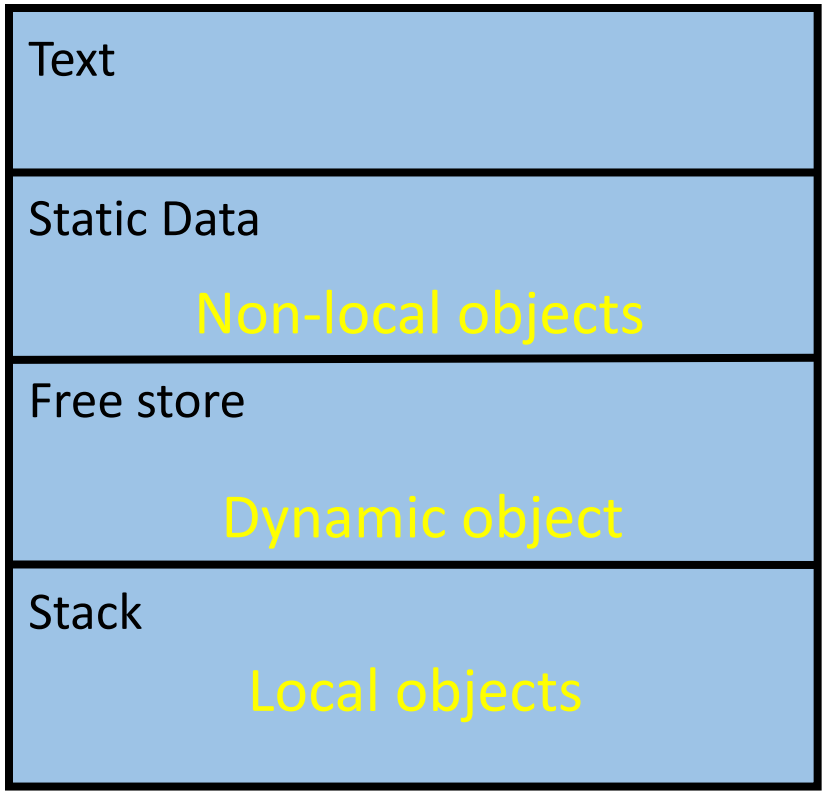



```
void f(auto int i, double j)
{
  string s = "ABC";
  vector<int> v(10);
  // …
}
```

```
class Point {
  int x, y;
public:
  Point() : x(0), y(0) {}
};
int main()
{
  Point p; // automatic object: ctor called
  Point* pp = &p; // automatic object
  Point* pp2 = new Point(); // pp2 is local,
                  // but the created point is dynamic
  Point* pa = new Point[10]; // dynamic array of 10 points
  int i = 10;
  delete pp2; // dtor of object pointed by pp2 called
  delete [] pa; // dtor of each elements of array called
  return 0;
} ← i, p, pp, pp2, pa are automatically destroyed here
```

Process structure & three kinds of memories

- Typical structure of UNIX/Windows process




```
void f(auto int i, double j)
{
  string s = "ABC";
  vector<int> v(10);
  // …
}
```

```
class Point {
  int x, y;
public:
  Point() : x(0), y(0) {}
};
int main()
{
  Point p; // automatic object: ctor called
  Point* pp = &p; // automatic object
  Point* pp2 = new Point(); // pp2 is local,
                  // but the created point is dynamic
  Point* pa = new Point[10]; // dynamic array of 10 points
  int i = 10;
  delete pp2; // dtor of object pointed by pp2 called
  delete [] pa; // dtor of each elements of array called
  return 0;
} ← i, p, pp, pp2, pa are automatically destroyed here
```

new ←→ delete

new [] ←→ delete []

Allocation and deallocation operators

Allocation and deallocation operators

- Operator new acquires its memory by calling an operator new(). Similarly, operator delete frees its memory by calling an operator delete().

Allocation and deallocation operators

- Operator new acquires its memory by calling an operator new(). Similarly, operator delete() frees its memory by calling an operator delete().

```
void* operator new(size_t); // use for individual object
void* operator new[](size_t); // use for array
void operator delete(void*, size_t); // use for individual object
void operator delete[](void*, size_t); // use for array
```

Allocation and deallocation operators

- Operator new acquires its memory by calling an operator new(). Similarly, operator delete() frees its memory by calling an operator delete().

```
void* operator new(size_t); // use for individual object
void* operator new[](size_t); // use for array
void operator delete(void*, size_t); // use for individual object
void operator delete[](void*, size_t); // use for array
```

- Placement new operators

- Don’t put objects on the free store if you don’t have to; prefer scoped variables.
- Avoid “naked new” and “naked delete”.
- Use RAII.
- Use standard-library facilities as a model for flexible, widely usable software.

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant