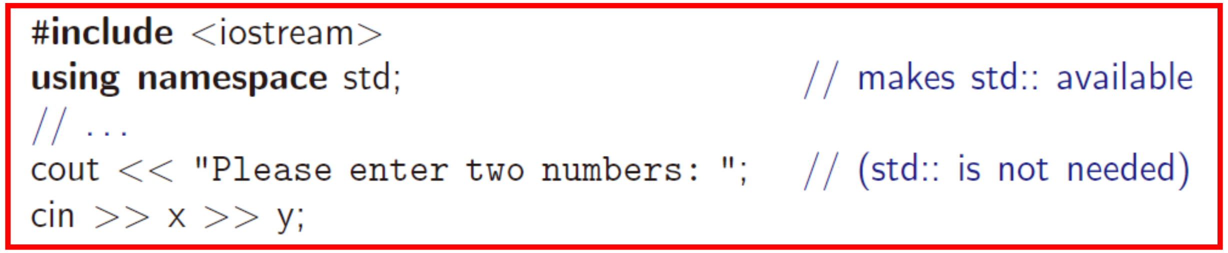
Data Structures (in C++)

- C++ Recap. -

C++ Recap.

A Simple C++ Program



```
#include <iostream>
using namespace std;                              // makes std:: available
// . . .
cout << "Please enter two numbers: ";             // (std:: is not needed)
cin >> x >> y;
```

Header file

Comment

Function

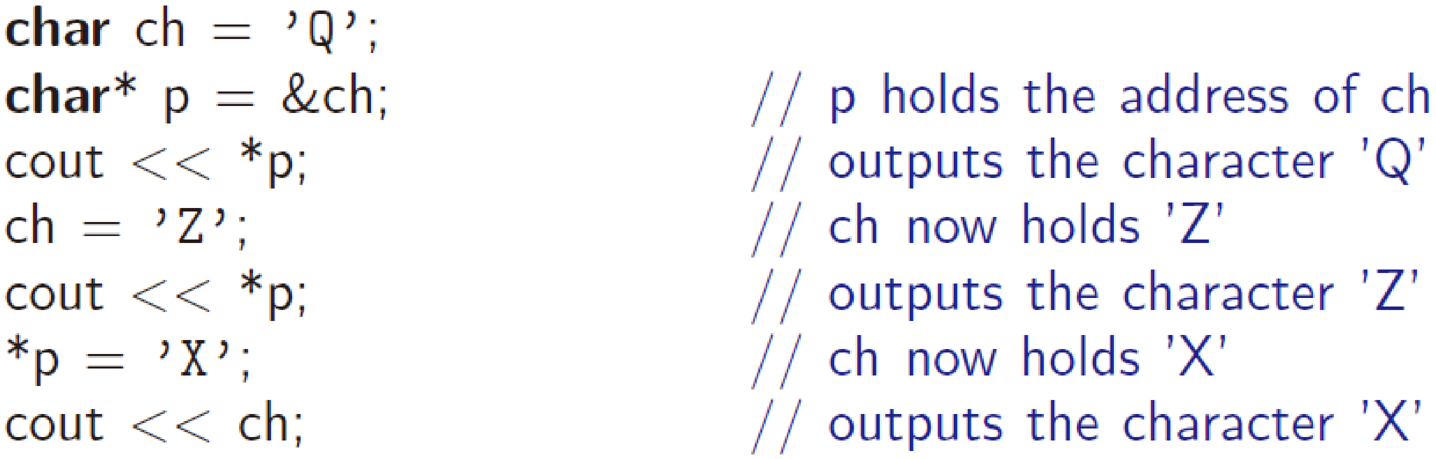
Variable

Namespace
Operators
Object/Class

```
#include <cstdlib>
#include <iostream>
/* This program inputs two numbers x and y and outputs their sum */
int main( ) {
  int x, y;
  std::cout << "Please enter two numbers: ";
  std::cin >> x >> y;                             // input x and y
  int sum = x + y;                                // compute their sum
  std::cout << "Their sum is " << sum << std::endl;
  return EXIT_SUCCESS;                            // terminate successfully
}
```

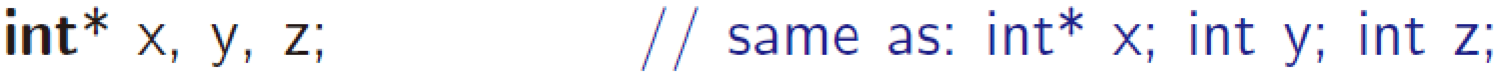
Pointers

- Each variable is stored in the machine’s memory at some location (*i.e.*, address)
- A **pointer** stores the address of a variable
  - address-of operator: ***&***
  - dereference (or indirection) operator: *******



```
char ch = ’Q’;
char* p = &ch;                  // p holds the address of ch
cout << *p;                     // outputs the character 'Q'
ch = ’Z’;                       // ch now holds 'Z'
cout << *p;                     // outputs the character 'Z'
*p = ’X’;                       // ch now holds 'X'
cout << ch;                     // outputs the character 'X'
```

- NOTE: The * operator binds with the variable name, not with the type name

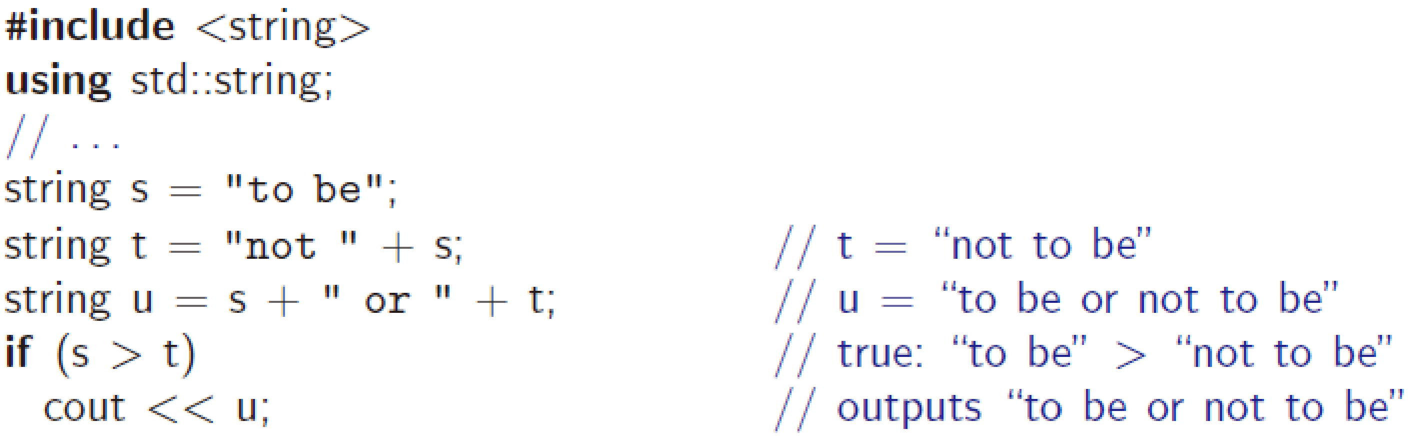


```
int* x, y, z;           // same as: int* x; int y; int z;
```

Strings

- **C-style Strings**
  - “Hello World”: A string literal
  - A fixed-length array of characters (+ null character at the end)
  - No string operations

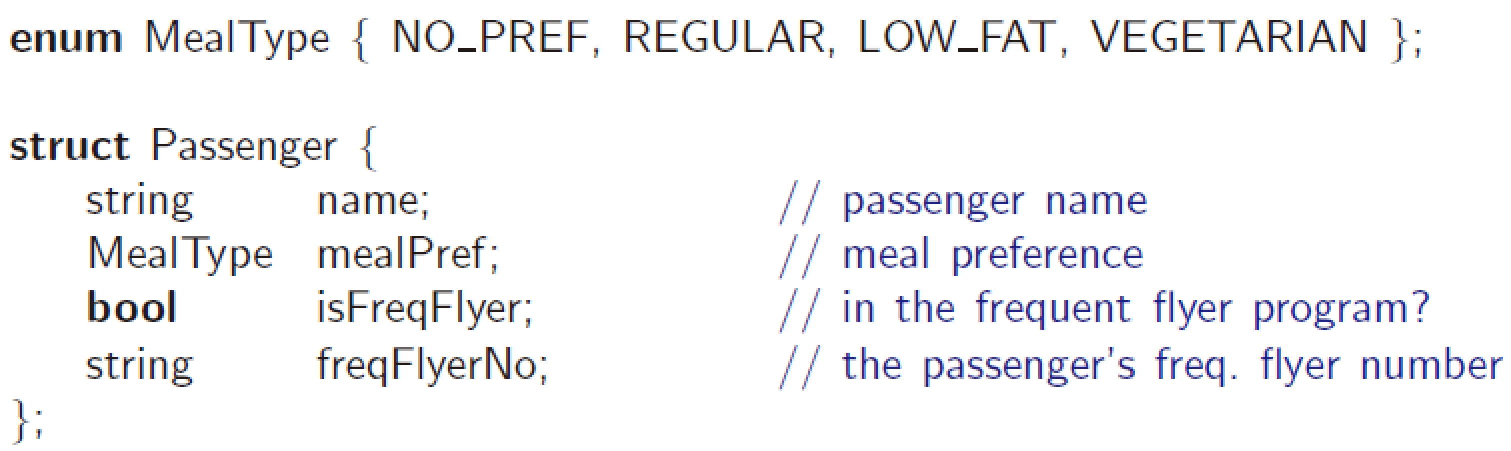
- **STL Strings**
  - #include <string>
  - Provides many convenient operations
    - Concatenation, Comparison, Searching, Conversion to upper-/lower- cases and so on



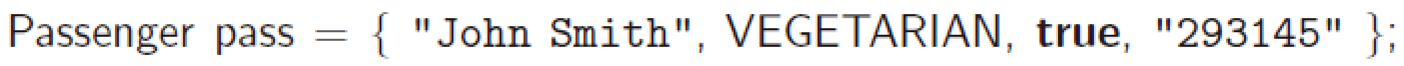
```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;                  // t = “not to be”
string u = s + " or " + t;              // u = “to be or not to be”
if (s > t)                              // true: “to be” > “not to be”
  cout << u;                            // outputs “to be or not to be”
```

C-Style Structures

- Useful for storing an aggregation of elements (*i.e.*, members or fields)
    - Member selection operator: .



```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string      name;                   // passenger name
    MealType    mealPref;               // meal preference
    bool        isFreqFlyer;            // in the frequent flyer program?
    string      freqFlyerNo;            // the passenger's freq. flyer number
};
```



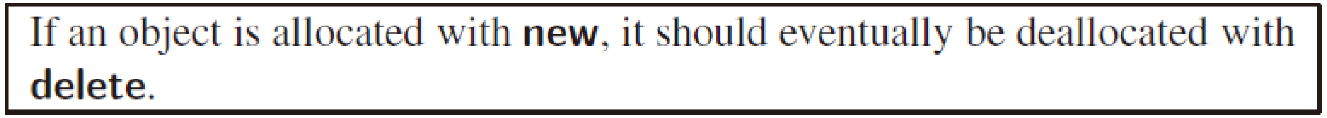
```
Passenger pass = { "John Smith", VEGETARIAN, true, "293145" };
```

```
pass.name = "Pocahontas";           // change name
pass.mealPref = REGULAR;            // change meal preference
```

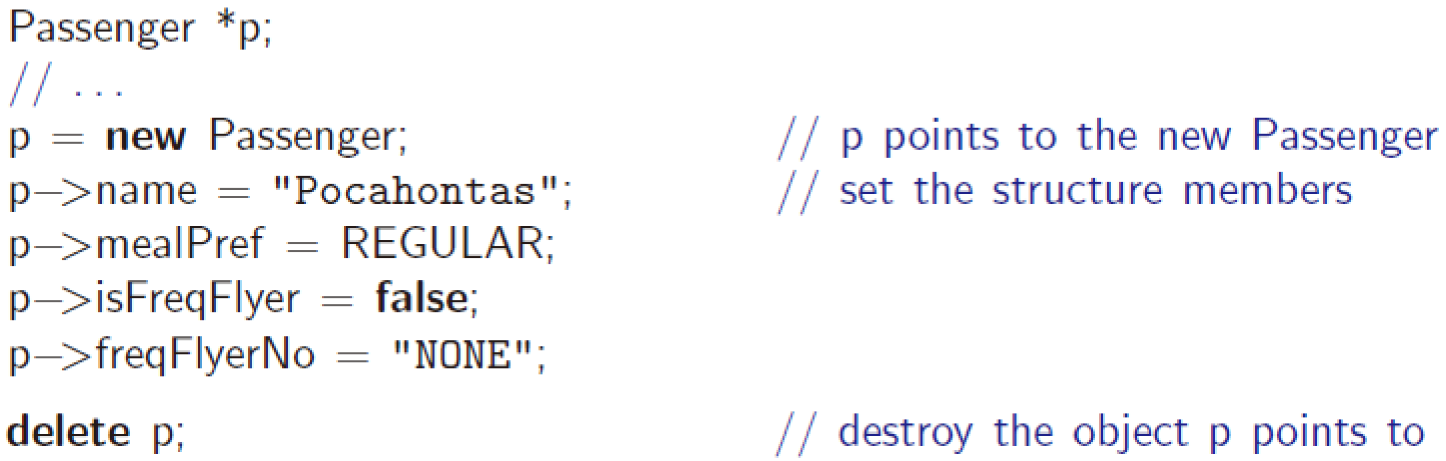
- This concept is extended to the **class** in C++

Dynamic Memory Allocation

- **Memory allocation at runtime**
  - Memory is allocated in heap memory (or free store)
  - Allocation: ***new***
    - The object’s **constructor** is called
  - Deallocation: ***delete***
    - The object’s **destructor** is called

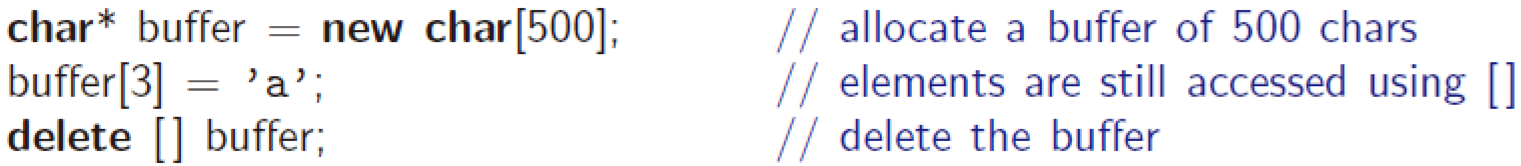


If an object is allocated with **new**, it should eventually be deallocated with **delete**.



```
Passenger *p;
// ...
p = new Passenger;                    // p points to the new Passenger
p->name = "Pocahontas";               // set the structure members
p->mealPref = REGULAR;
p->isFreqFlyer = false;
p->freqFlyerNo = "NONE";

delete p;                             // destroy the object p points to
```

Array allocation & deallocation



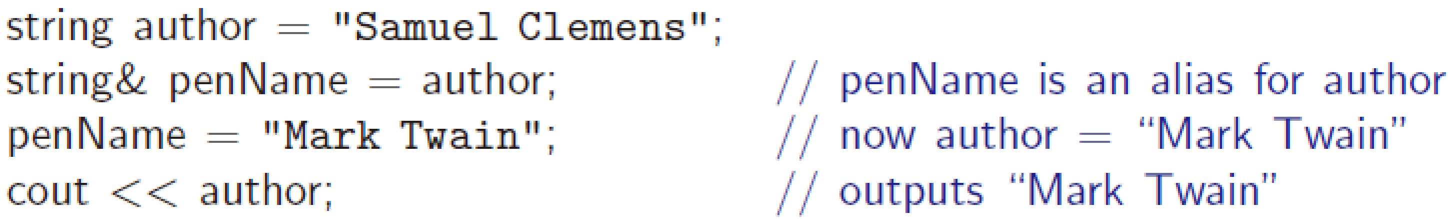
```
char* buffer = new char[500];         // allocate a buffer of 500 chars
buffer[3] = ’a’;                      // elements are still accessed using []
delete [] buffer;                     // delete the buffer
```

References

- An alternative name for an object

- A reference must refer to an actual variable
  - Note that a pointer can point nothing (*i.e.*, NULL pointer)

- Any access to the reference is an access to the underlying object
  - Useful for function arguments



```
string author = "Samuel Clemens";
string& penName = author;          // penName is an alias for author
penName = "Mark Twain";            // now author = "Mark Twain"
cout << author;                    // outputs "Mark Twain"
```

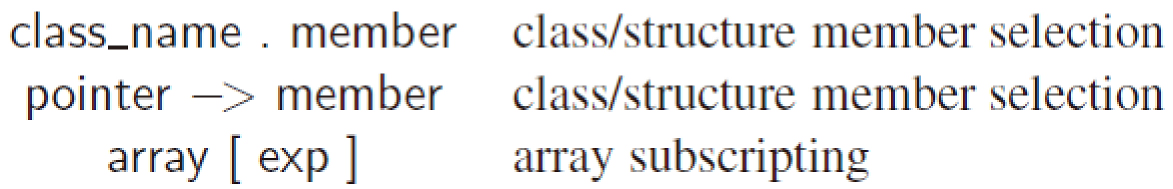
Expressions and Operators

Expression

- An expression is a sequence of operators and their operands, that specifies a computation
- Combines variables and literals with operators to create new values

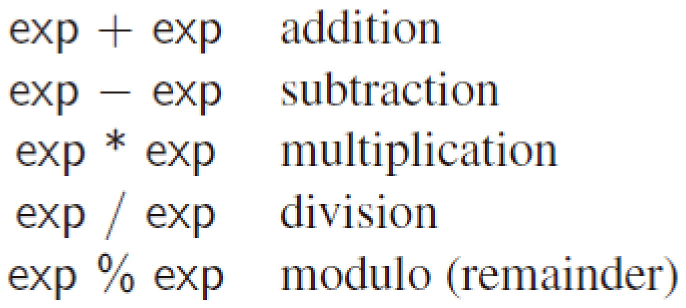
var: variable
exp: expression (i.e., value)

Member Selection and Indexing



class_name . member	class/structure member selection
pointer −> member	class/structure member selection
array [exp]	array subscripting

Arithmetic Operators

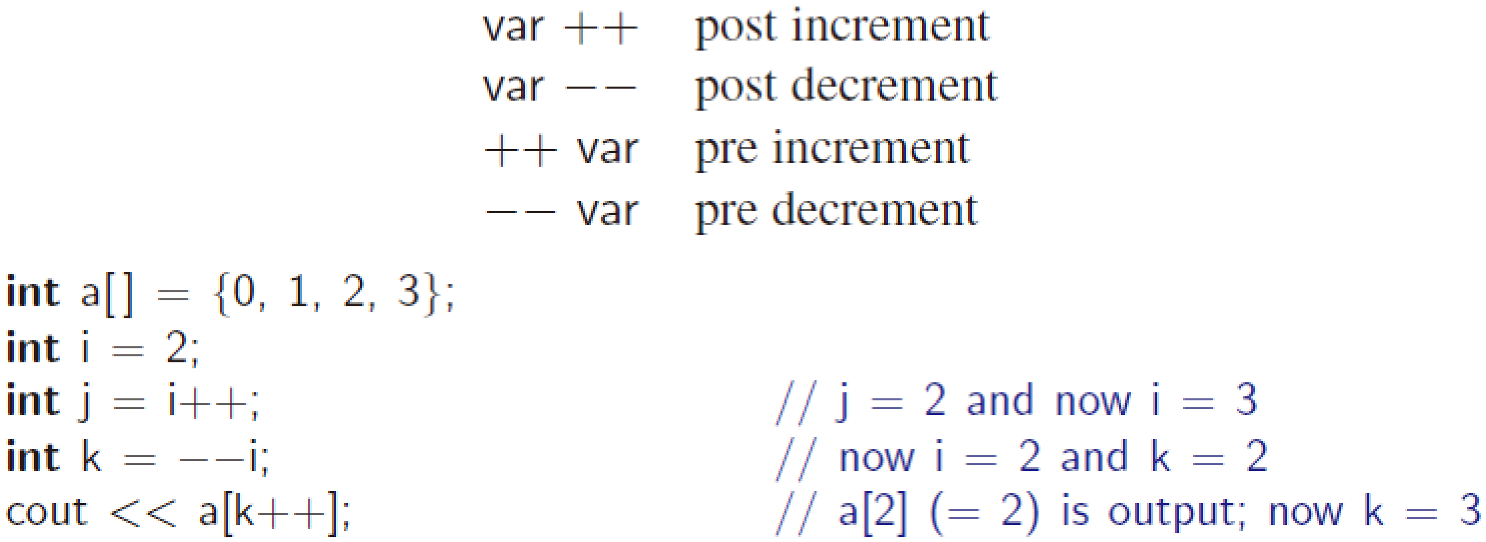


exp + exp	addition
exp − exp	subtraction
exp * exp	multiplication
exp / exp	division
exp % exp	modulo (remainder)

Expressions and Operators

Increment and Decrement Operators

- Post-increment/decrement
  - Returns a variable's value then increase/decrease its value
- Pre-increment/decrement
  - Increase/decrease a variable's value then return it

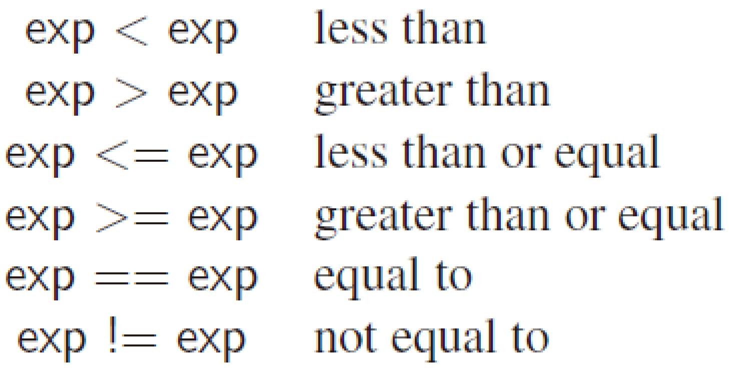


var ++	post increment
var --	post decrement
++ var	pre increment
-- var	pre decrement

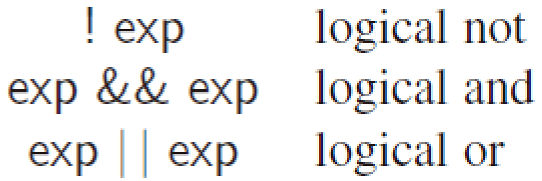
```
int a[] = {0, 1, 2, 3};
int i = 2;
int j = i++;                    // j = 2 and now i = 3
int k = --i;                    // now i = 2 and k = 2
cout << a[k++];                 // a[2] (= 2) is output; now k = 3
```

Expressions and Operators

▪ Relational and Logical Operators



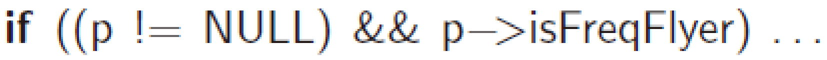
exp < exp	less than
exp > exp	greater than
exp <= exp	less than or equal
exp >= exp	greater than or equal
exp == exp	equal to
exp != exp	not equal to



! exp	logical not
exp && exp	logical and
exp || exp	logical or

▪ Short-Circuit Evaluation

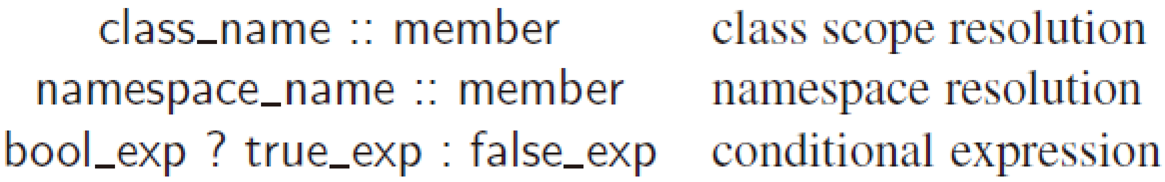
- && and || operators evaluate sequentially from left to right
- If the left operand is enough to determine the expression value, the right one is skipped



```
if ((p != NULL) && p->isFreqFlyer) ...
```

Expressions and Operators

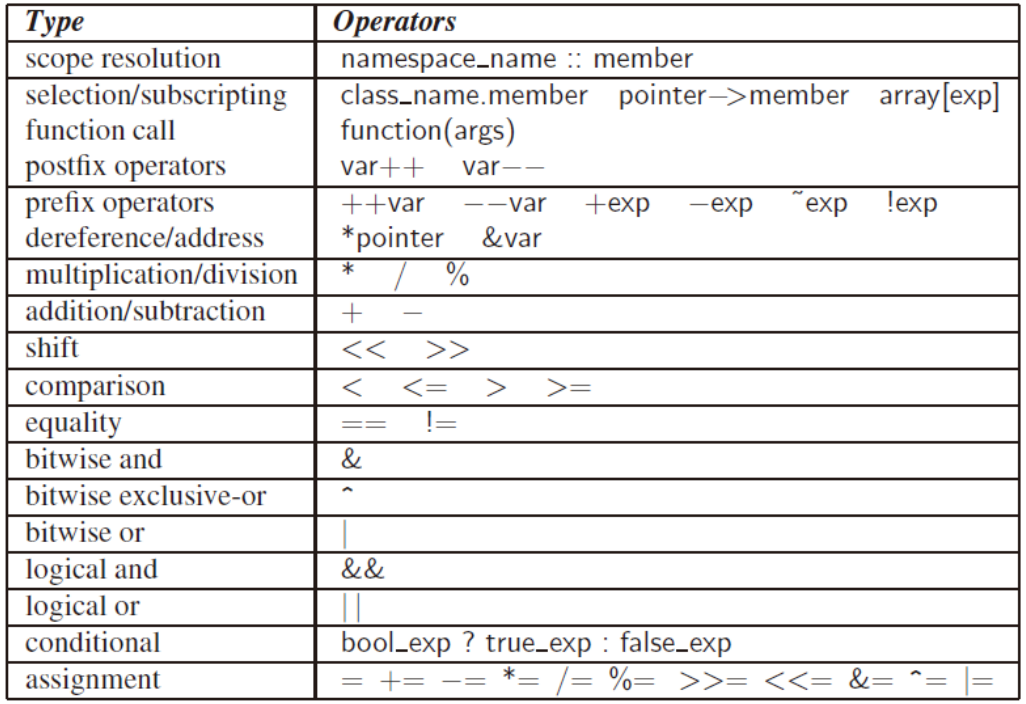
▪ Other Operators



class_name :: member	class scope resolution
namespace_name :: member	namespace resolution
bool_exp ? true_exp : false_exp	conditional expression

ternary operator

▪ Operator Precedence



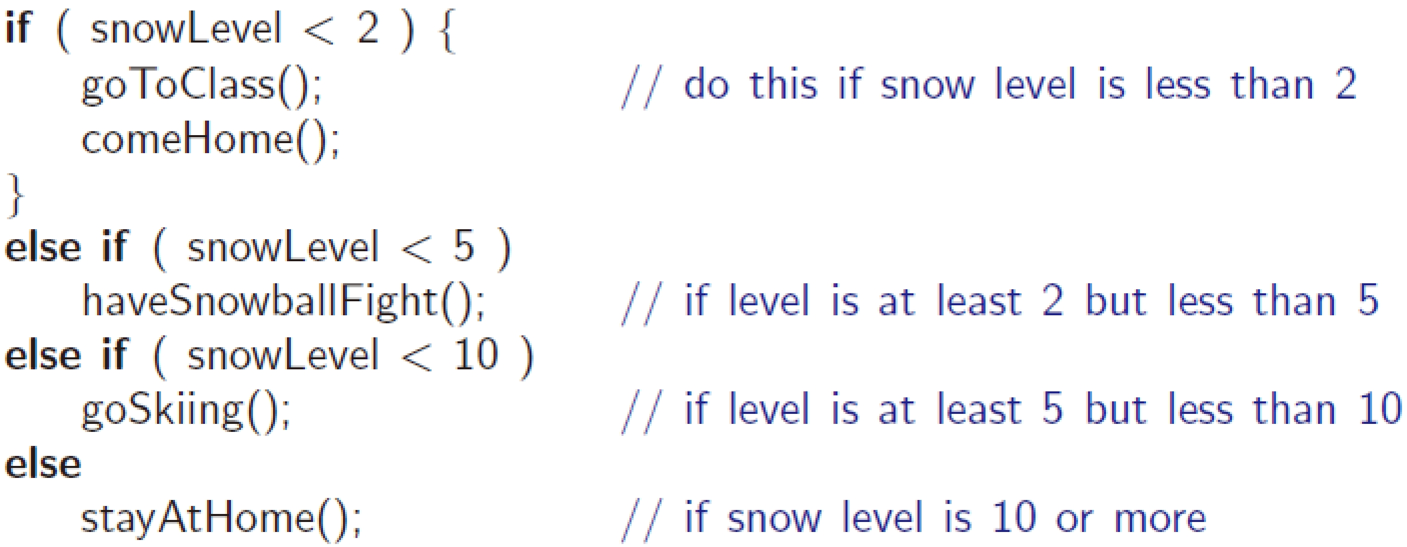
Type	*Operators*
scope resolution	namespace_name :: member
selection/subscripting function call postfix operators	class_name.member pointer−>member array[exp] function(args) var++ var−−
prefix operators dereference/address	++var −−var +exp −exp ˜exp !exp *pointer &var
multiplication/division	* / %
addition/subtraction	+ −
shift	<< >>
comparison	< <= > >=
equality	== !=
bitwise and	&
bitwise exclusive-or	ˆ
bitwise or	|
logical and	&&
logical or	||
conditional	bool_exp ? true_exp : false_exp
assignment	= += −= *= /= %= >>= <<= &= ˆ= |=

Highest

Lowest

Control Flow

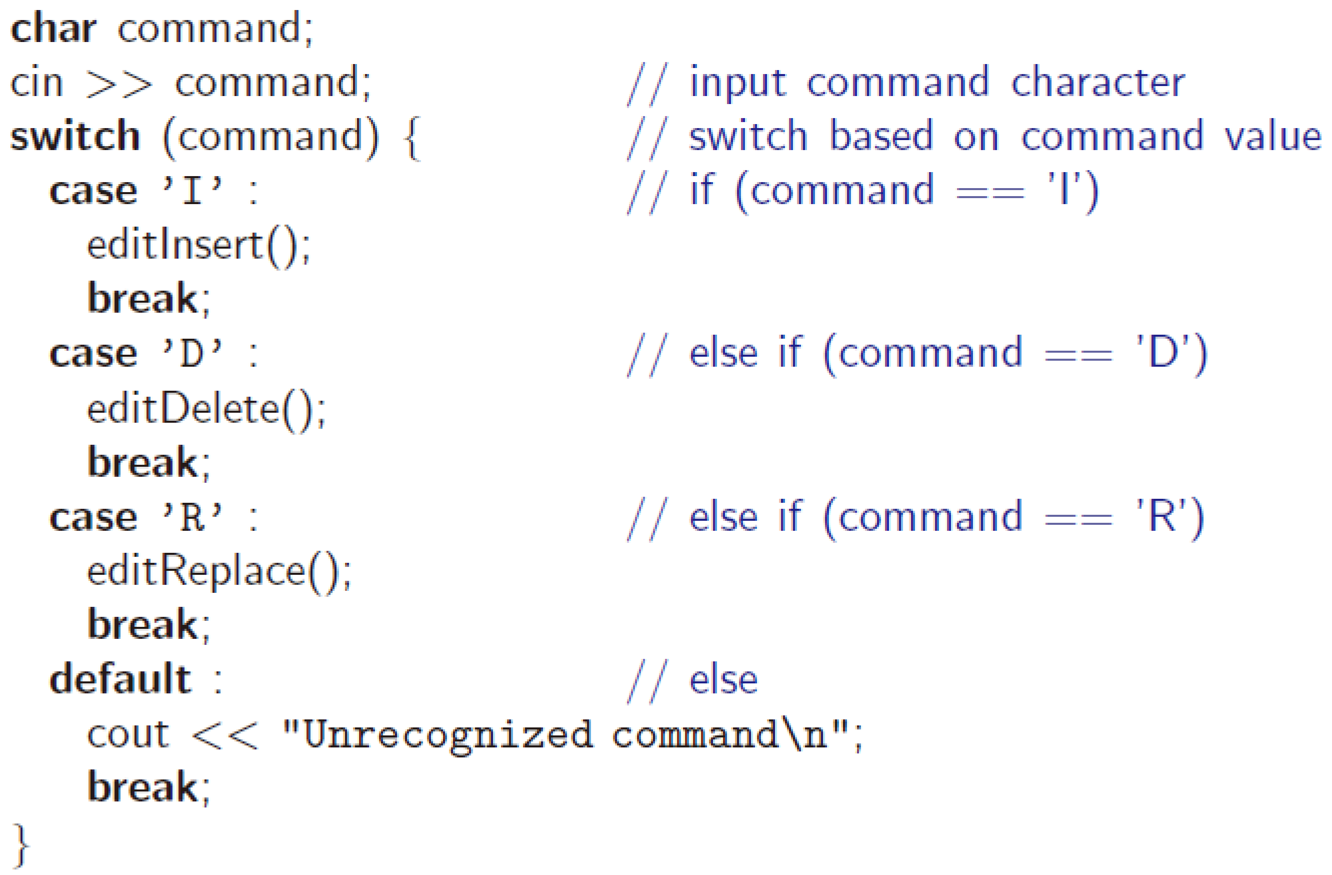
- ***if*** **Statement**
    - ***else if*** and ***else*** parts are optional



```
if ( snowLevel < 2 ) {
    goToClass();                // do this if snow level is less than 2
    comeHome();
}
else if ( snowLevel < 5 )
    haveSnowballFight();        // if level is at least 2 but less than 5
else if ( snowLevel < 10 )
    goSkiing();                 // if level is at least 5 but less than 10
else
    stayAtHome();               // if snow level is 10 or more
```

Control Flow

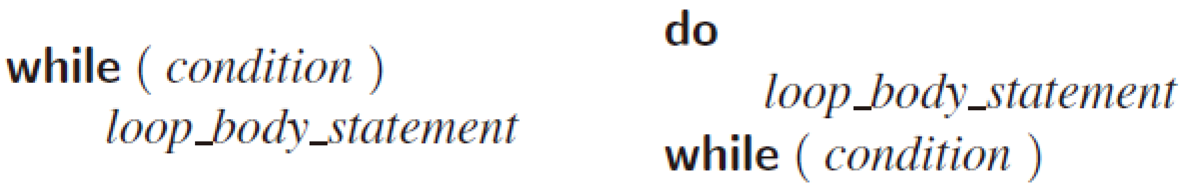
- ***switch* Statement**
    - Distinguish between many different integral type options



```
char command;
cin >> command;                 // input command character
switch (command) {              // switch based on command value
  case 'I' :                    // if (command == 'I')
    editInsert();
    break;
  case 'D' :                    // else if (command == 'D')
    editDelete();
    break;
  case 'R' :                    // else if (command == 'R')
    editReplace();
    break;
  default :                     // else
    cout << "Unrecognized command\n";
    break;
}
```

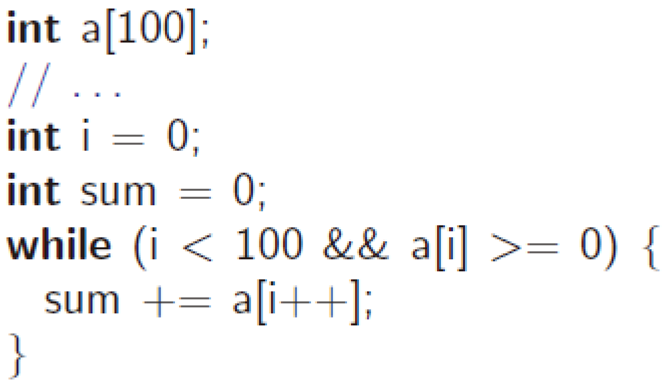
Control Flow

- ***while*** **and** ***do-while*** **loops**
  - Iterates over a set of statements as long as some specified condition holds



```
while ( condition )
    loop_body_statement
```

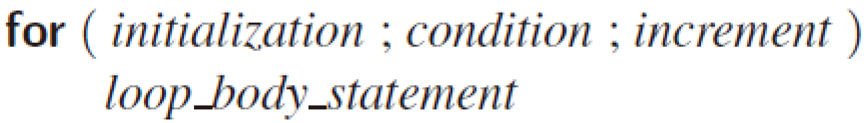
```
do
    loop_body_statement
while ( condition )
```



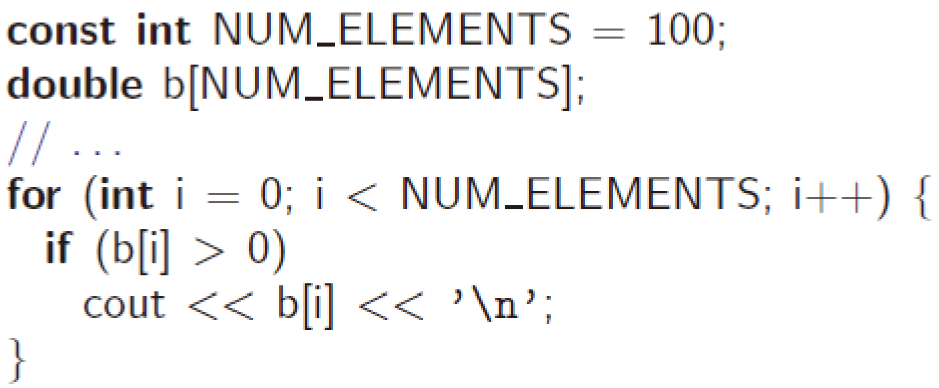
```
int a[100];
// ...
int i = 0;
int sum = 0;
while (i < 100 && a[i] >= 0) {
  sum += a[i++];
}
```

Control Flow

- ***for*** **loops**
  - Encapsulates three elements for a loop: an initialization, a condition, and an increment



```
for ( initialization ; condition ; increment )
     loop_body_statement
```

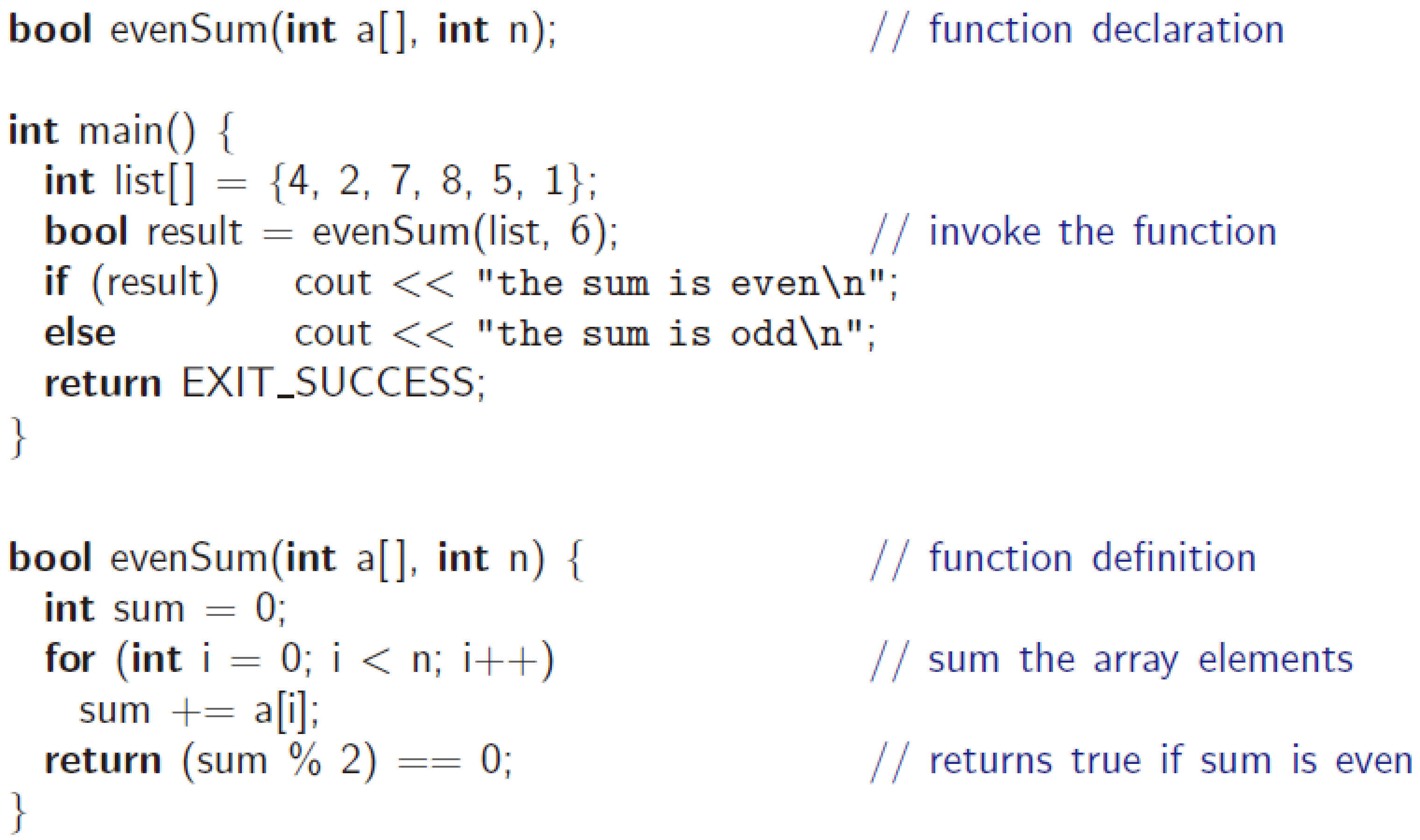


```
const int NUM_ELEMENTS = 100;
double b[NUM_ELEMENTS];
// ...
for (int i = 0; i < NUM_ELEMENTS; i++) {
  if (b[i] > 0)
    cout << b[i] << '\n';
}
```

Functions

- A chunk of code that can be called to perform some well-defined task
- To define a function:
  - Return type
  - Function name
  - Argument list
  - Function body

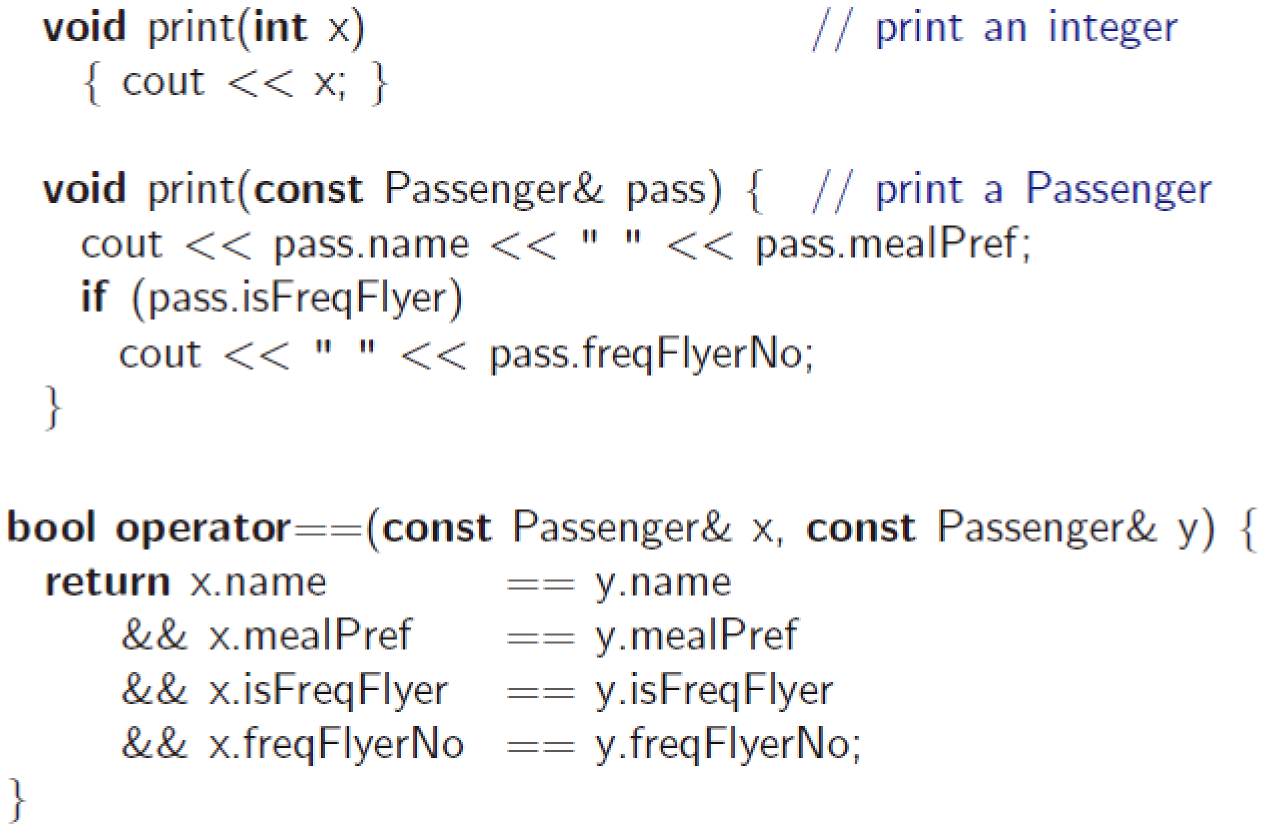
Signature
or
Prototype



```
bool evenSum(int a[], int n);                    // function declaration

int main() {
  int list[] = {4, 2, 7, 8, 5, 1};
  bool result = evenSum(list, 6);                // invoke the function
  if (result)      cout << "the sum is even\n";
  else             cout << "the sum is odd\n";
  return EXIT_SUCCESS;
}

bool evenSum(int a[], int n) {                   // function definition
  int sum = 0;
  for (int i = 0; i < n; i++)                    // sum the array elements
    sum += a[i];
  return (sum % 2) == 0;                         // returns true if sum is even
}
```

Overloading

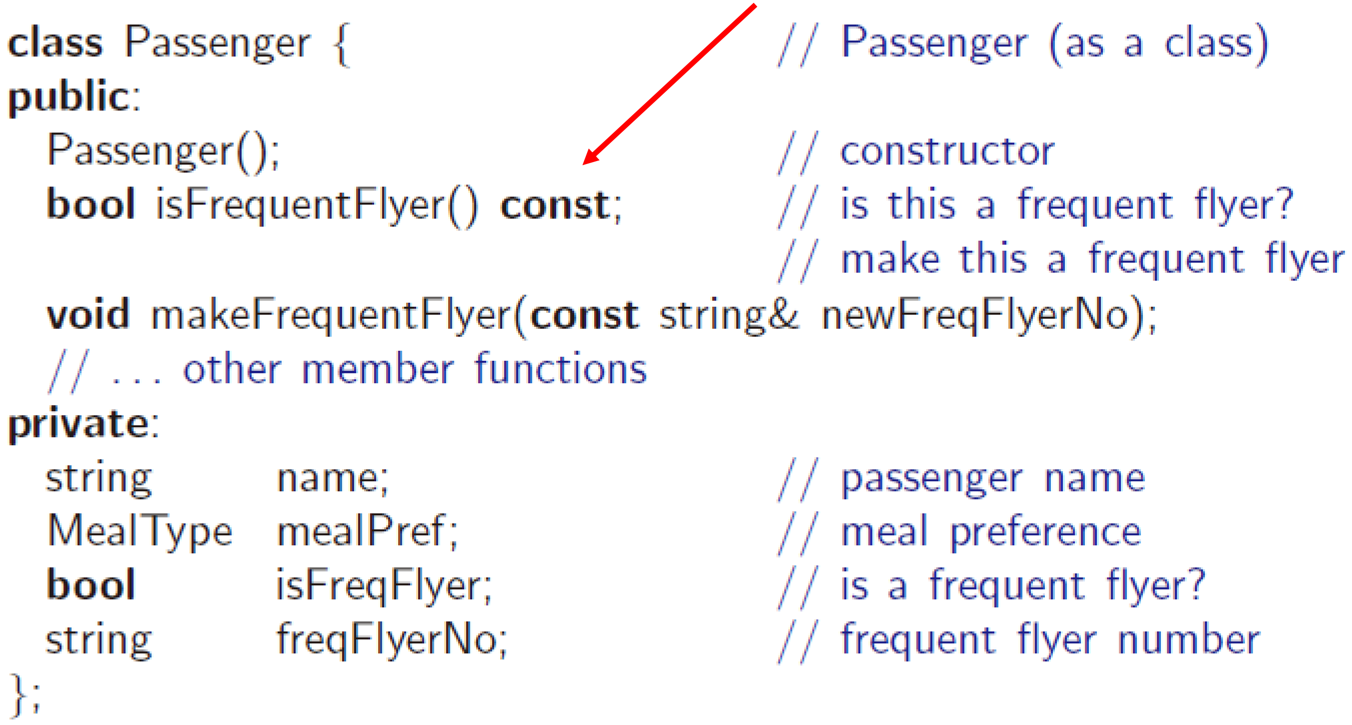
- **Function/Operator Overloading**
  - Two or more functions/operators are defined with the same name but with different argument lists
  - The complier determines which function should be invoked



```
void print(int x)                          // print an integer
  { cout << x; }

void print(const Passenger& pass) {   // print a Passenger
  cout << pass.name << " " << pass.mealPref;
  if (pass.isFreqFlyer)
    cout << " " << pass.freqFlyerNo;
}

bool operator==(const Passenger& x, const Passenger& y) {
  return x.name         == y.name
      && x.mealPref     == y.mealPref
      && x.isFreqFlyer  == y.isFreqFlyer
      && x.freqFlyerNo  == y.freqFlyerNo;
}
```

Classes

- **Class**
  - A user-defined type which consists of members:
    - member variables
    - member functions
  - Access specifiers
    - Private (by default)
    - Public
    - Protected



```
class Passenger {                           // Passenger (as a class)
public:
  Passenger();                              // constructor
  bool isFrequentFlyer() const;             // is this a frequent flyer?
                                            // make this a frequent flyer
  void makeFrequentFlyer(const string& newFreqFlyerNo);
  // . . . other member functions
private:
  string     name;                          // passenger name
  MealType   mealPref;                      // meal preference
  bool       isFreqFlyer;                   // is a frequent flyer?
  string     freqFlyerNo;                   // frequent flyer number
};
```

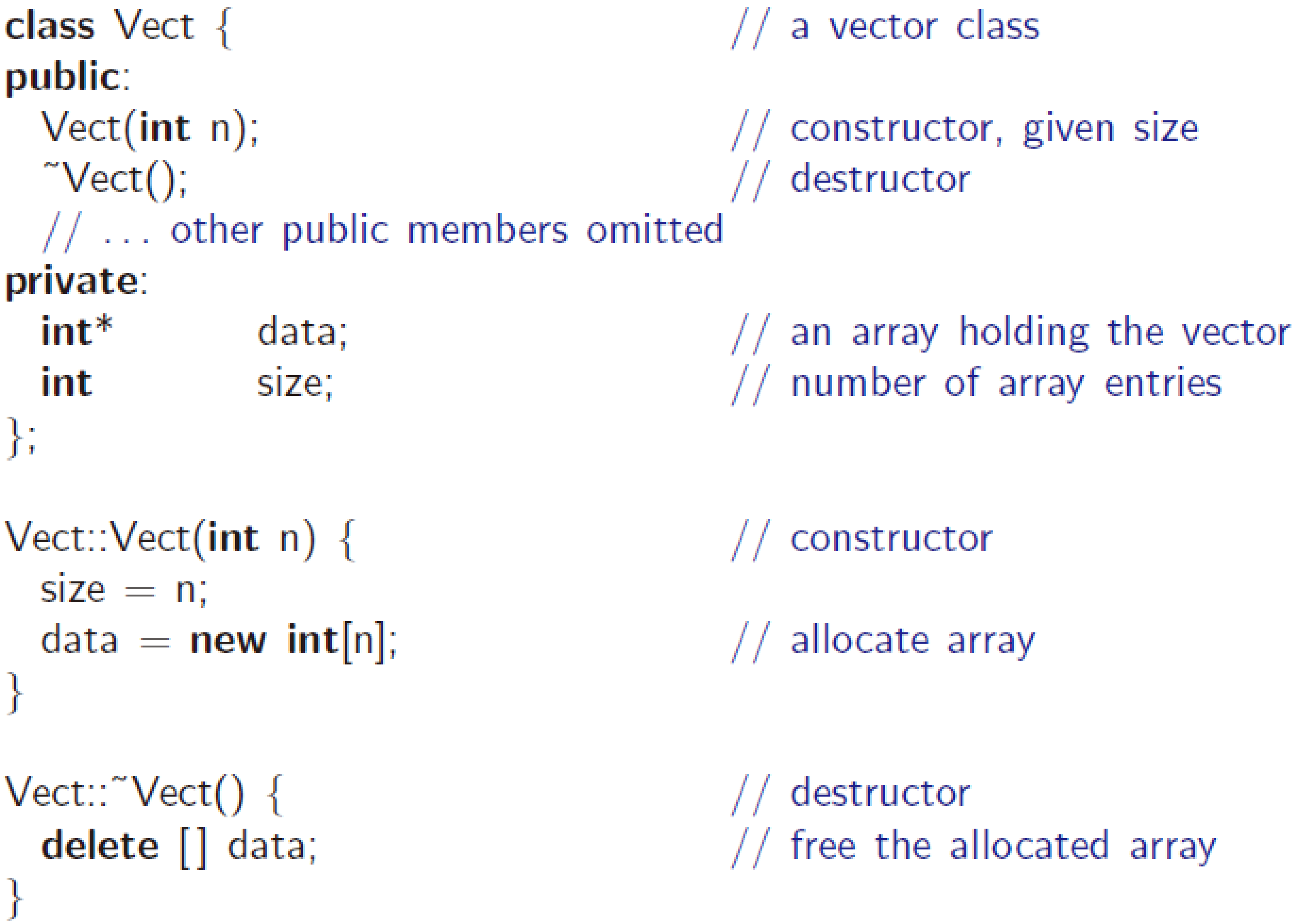
Indicates an accessor

Can access from the outside

Cannot access from the outside

Classes

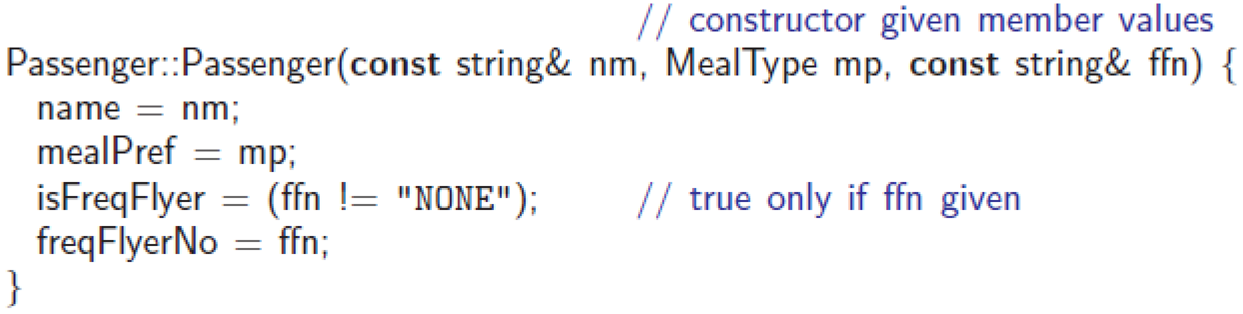
- **Constructors**
  - A special member function for the initialization: ***class_name(arguments_list)***
  - Invoked when a new class instance is created
- **Destructors**
  - A special member function for the destruction: ***~class_name()***
  - Invoked when an existing class instance goes out of existence



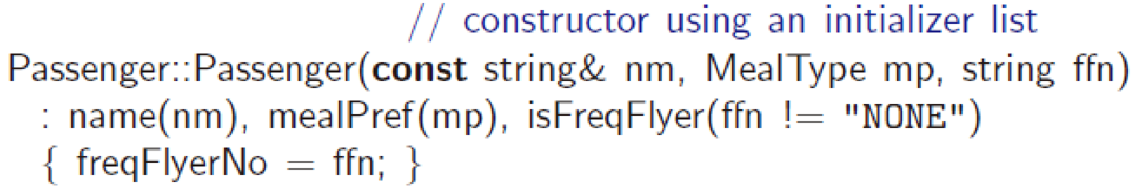
```
class Vect {                                   // a vector class
public:
  Vect(int n);                                 // constructor, given size
  ~Vect();                                     // destructor
  // . . . other public members omitted
private:
  int*      data;                              // an array holding the vector
  int       size;                              // number of array entries
};

Vect::Vect(int n) {                            // constructor
  size = n;
  data = new int[n];                           // allocate array
}

Vect::~Vect() {                                // destructor
  delete [] data;                              // free the allocated array
}
```

Classes

- **Initializer List**
  - Placed between the constructor's argument list and its body
  - ***member_name(initial_value)***

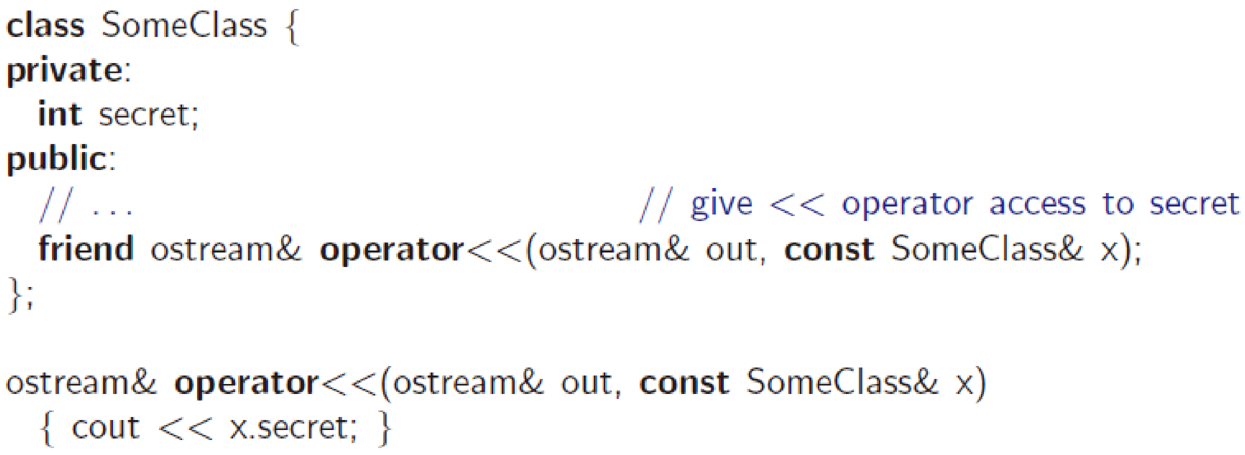


```
                                          // constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
  name = nm;
  mealPref = mp;
  isFreqFlyer = (ffn != "NONE");          // true only if ffn given
  freqFlyerNo = ffn;
}
```



```
                                          // constructor using an initializer list
Passenger::Passenger(const string& nm, MealType mp, string ffn)
  : name(nm), mealPref(mp), isFreqFlyer(ffn != "NONE")
  { freqFlyerNo = ffn; }
```

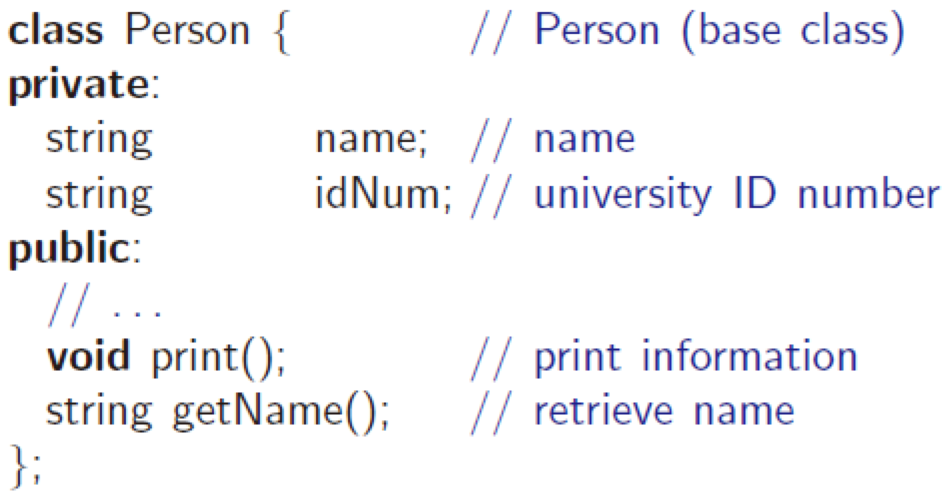
- ***Class Friends***
  - A friend of a class can access the private data of the class



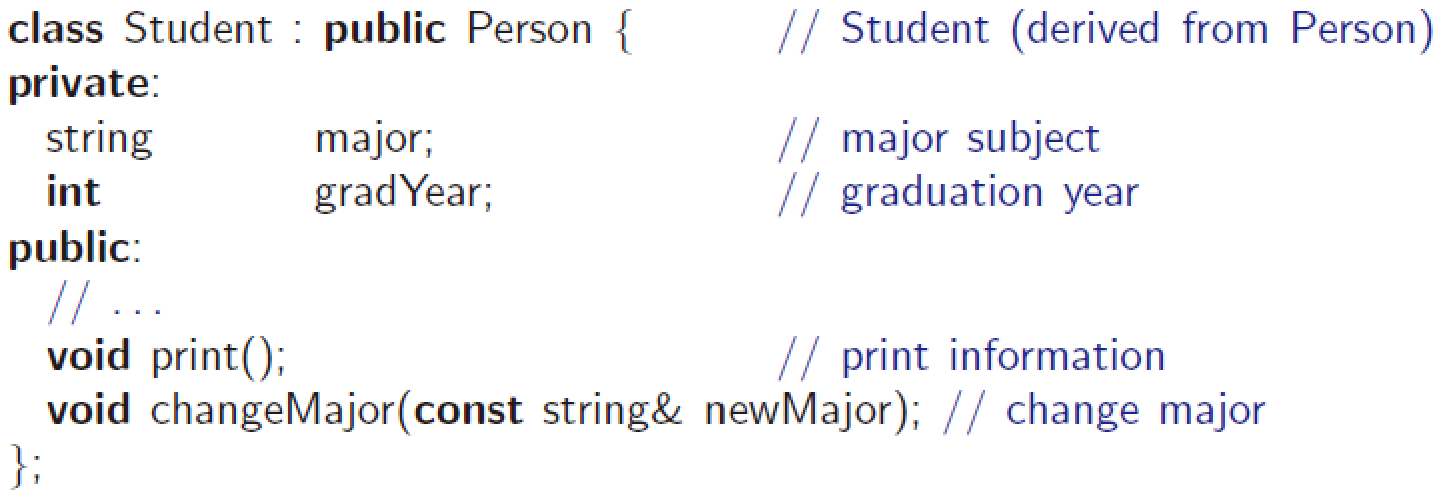
```
class SomeClass {
private:
  int secret;
public:
  // . . .                              // give << operator access to secret
  friend ostream& operator<<(ostream& out, const SomeClass& x);
};

ostream& operator<<(ostream& out, const SomeClass& x)
  { cout << x.secret; }
```

Inheritance and Polymorphism

- **Inheritance**
  - Allows the design of generic classes that can be specialized to more particular classes
  - A generic class is known as a ***base class***, ***parent class***, or ***superclass***
  - Any class that specializes or extends a base class is called a ***derived class***, ***child class***, or ***subclass***



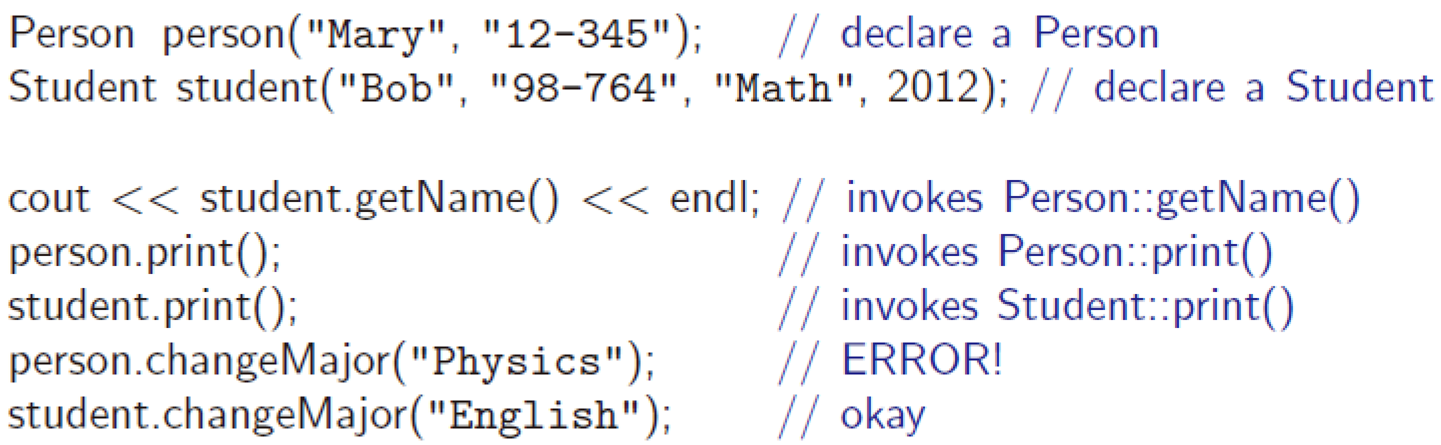
```
class Person {              // Person (base class)
private:
  string        name;       // name
  string        idNum;      // university ID number
public:
  // . . .
  void print();             // print information
  string getName();         // retrieve name
};
```



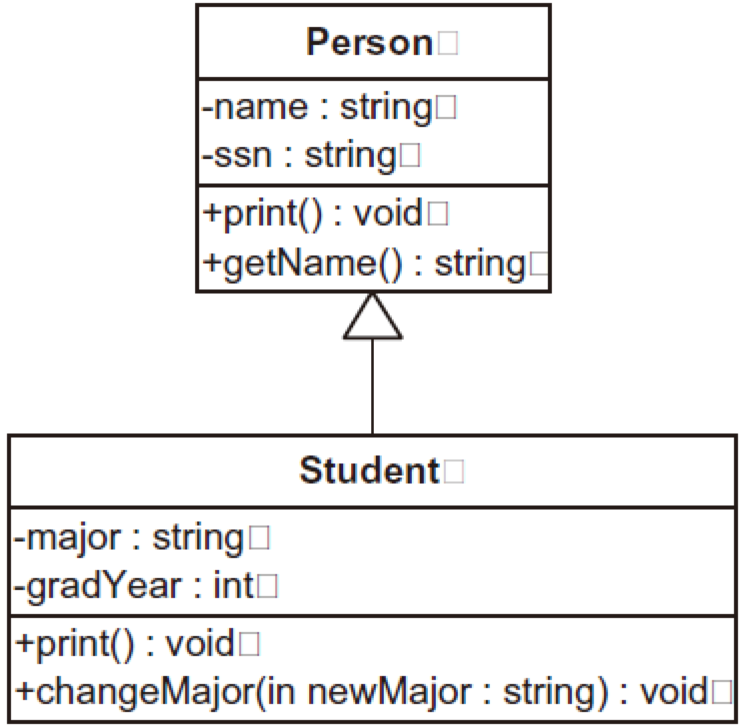
```
class Student : public Person {          // Student (derived from Person)
private:
  string        major;                   // major subject
  int           gradYear;                // graduation year
public:
  // . . .
  void print();                          // print information
  void changeMajor(const string& newMajor); // change major
};
```

Inheritance and Polymorphism

- **Inheritance**

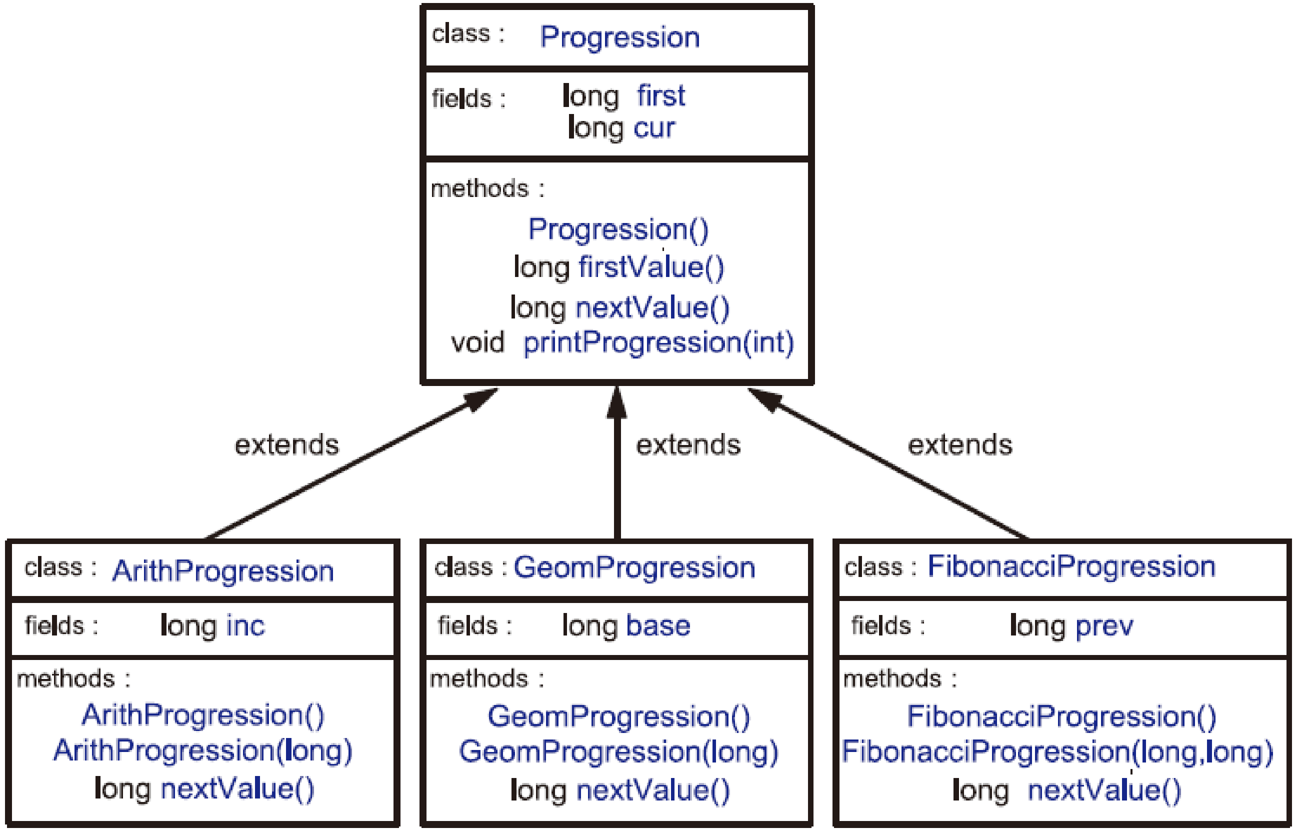


```
Person  person("Mary", "12-345");     // declare a Person
Student student("Bob", "98-764", "Math", 2012); // declare a Student

cout << student.getName() << endl; // invokes Person::getName()
person.print();                    // invokes Person::print()
student.print();                   // invokes Student::print()
person.changeMajor("Physics");     // ERROR!
student.changeMajor("English");    // okay
```



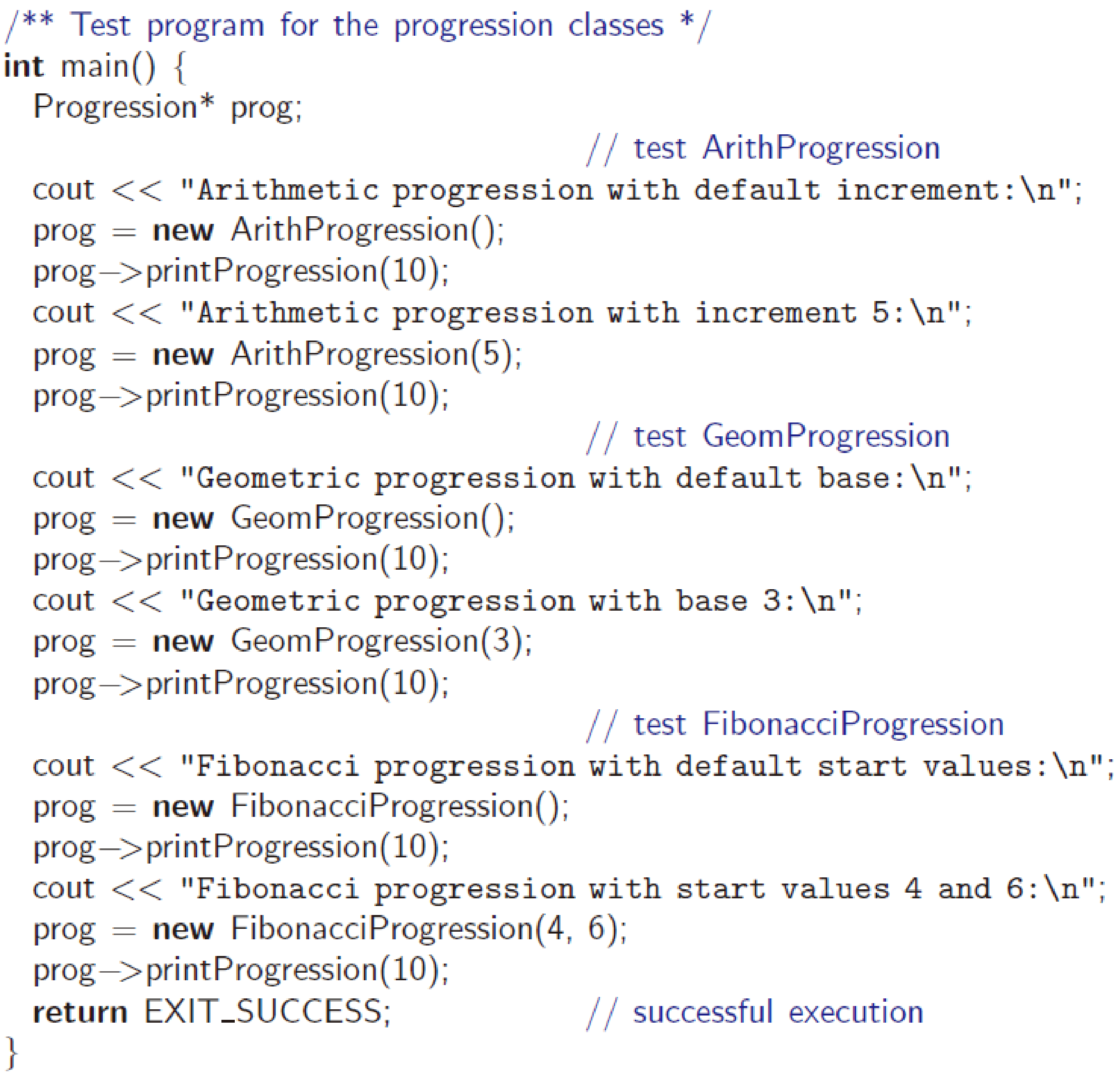
Inheritance and Polymorphism

- **Polymorphism**
  - The ability of a variable to take different types
  - A variable *p* declared to be a pointer to some class *S* implies that *p* can point to any object belonging to any derived class *T* of *S*



Inheritance and Polymorphism

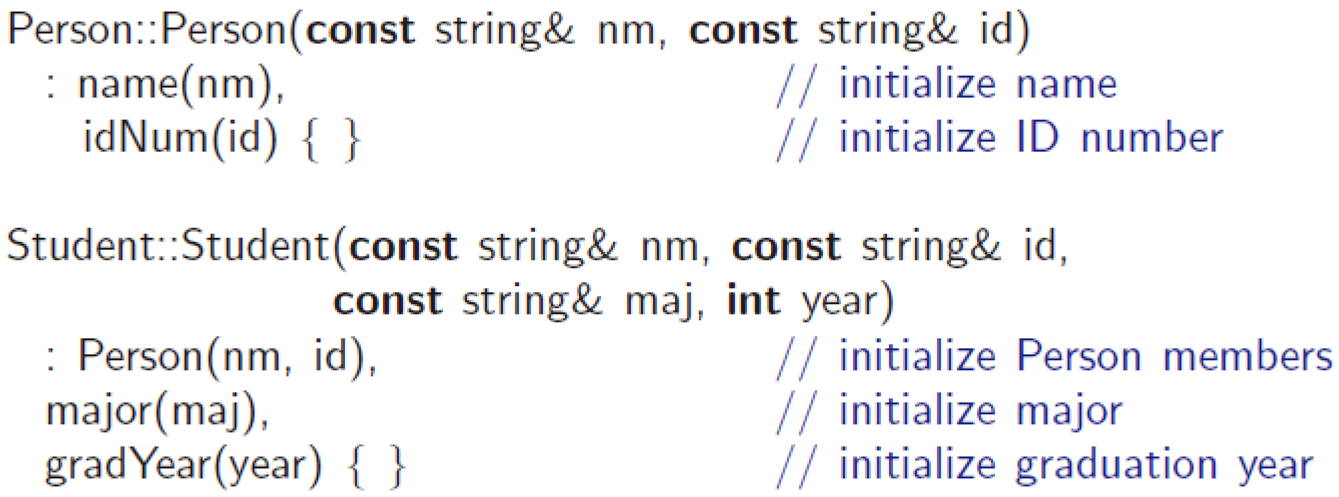
- **Polymorphism**



```
/** Test program for the progression classes */
int main() {
  Progression* prog;
                                        // test ArithProgression
  cout << "Arithmetic progression with default increment:\n";
  prog = new ArithProgression();
  prog->printProgression(10);
  cout << "Arithmetic progression with increment 5:\n";
  prog = new ArithProgression(5);
  prog->printProgression(10);
                                        // test GeomProgression
  cout << "Geometric progression with default base:\n";
  prog = new GeomProgression();
  prog->printProgression(10);
  cout << "Geometric progression with base 3:\n";
  prog = new GeomProgression(3);
  prog->printProgression(10);
                                        // test FibonacciProgression
  cout << "Fibonacci progression with default start values:\n";
  prog = new FibonacciProgression();
  prog->printProgression(10);
  cout << "Fibonacci progression with start values 4 and 6:\n";
  prog = new FibonacciProgression(4, 6);
  prog->printProgression(10);
  return EXIT_SUCCESS;                  // successful execution
}
```

Inheritance and Polymorphism

- **Constructors**
  - Parent class constructor first, then child class constructor



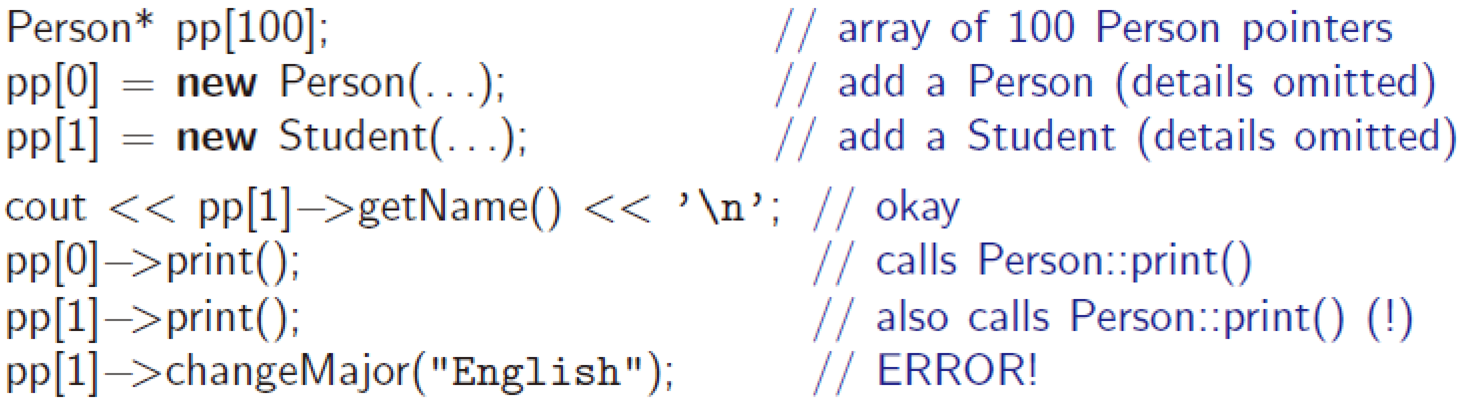
```
Person::Person(const string& nm, const string& id)
  : name(nm),                          // initialize name
    idNum(id) { }                      // initialize ID number

Student::Student(const string& nm, const string& id,
                 const string& maj, int year)
  : Person(nm, id),                    // initialize Person members
  major(maj),                          // initialize major
  gradYear(year) { }                   // initialize graduation year
```

- **Destructors**
  - Child class destructor first, then parent class destructor

```
delete s;                        // calls ~Student() then ~Person()
```

Dynamic Binding and Virtual Functions

- **Static Binding**
  - An object’s declared type determine its behavior (not by its actual type)

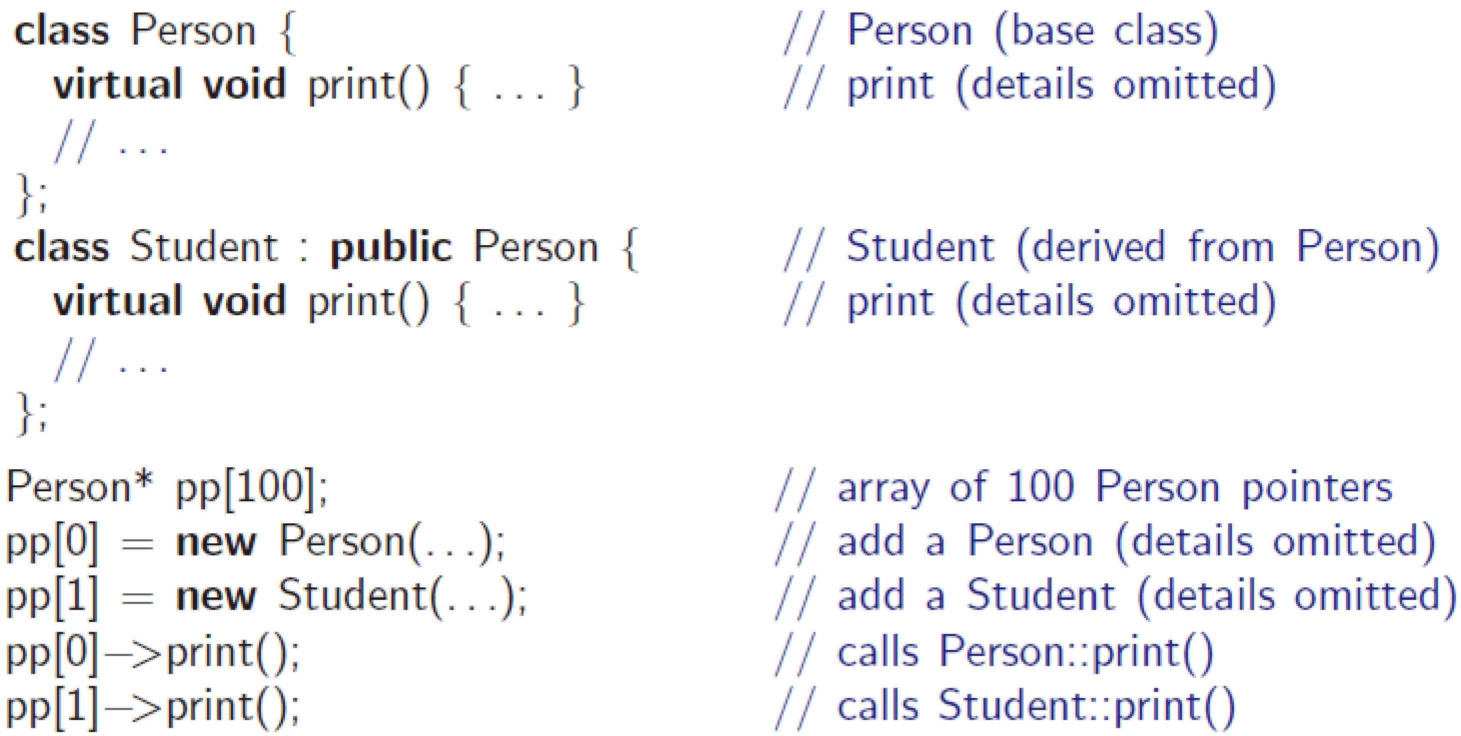


```
Person* pp[100];                        // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);               // add a Student (details omitted)
cout << pp[1]->getName() << '\n';  // okay
pp[0]->print();                         // calls Person::print()
pp[1]->print();                         // also calls Person::print() (!)
pp[1]->changeMajor("English");          // ERROR!
```

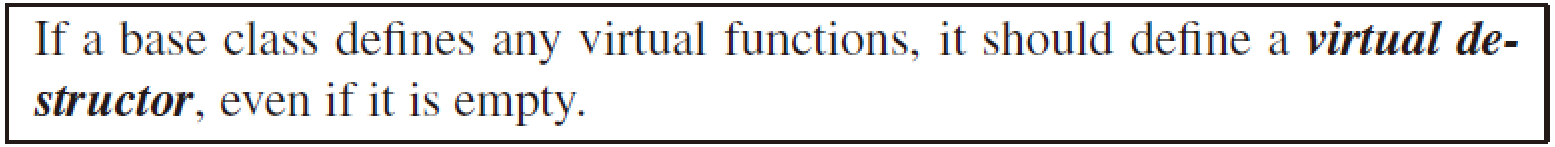
Dynamic Binding and Virtual Functions

Dynamic Binding

- An object's contents determine its behavior (by its actual type)
- ***virtual*** keyword is needed



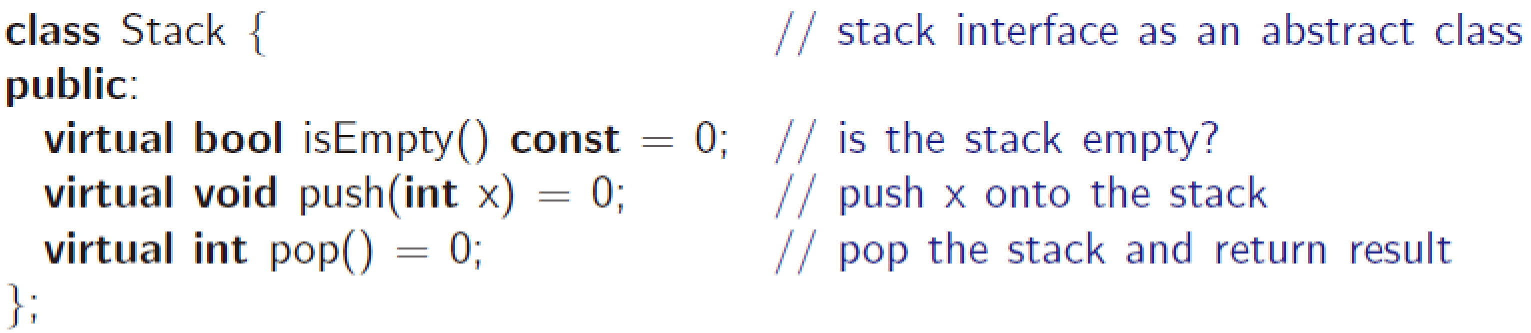
```
class Person {                          // Person (base class)
  virtual void print() { ... }          // print (details omitted)
  // ...
};
class Student : public Person {         // Student (derived from Person)
  virtual void print() { ... }          // print (details omitted)
  // ...
};

Person* pp[100];                        // array of 100 Person pointers
pp[0] = new Person(...);                // add a Person (details omitted)
pp[1] = new Student(...);               // add a Student (details omitted)
pp[0]−>print();                         // calls Person::print()
pp[1]−>print();                         // calls Student::print()
```



If a base class defines any virtual functions, it should define a ***virtual destructor***, even if it is empty.

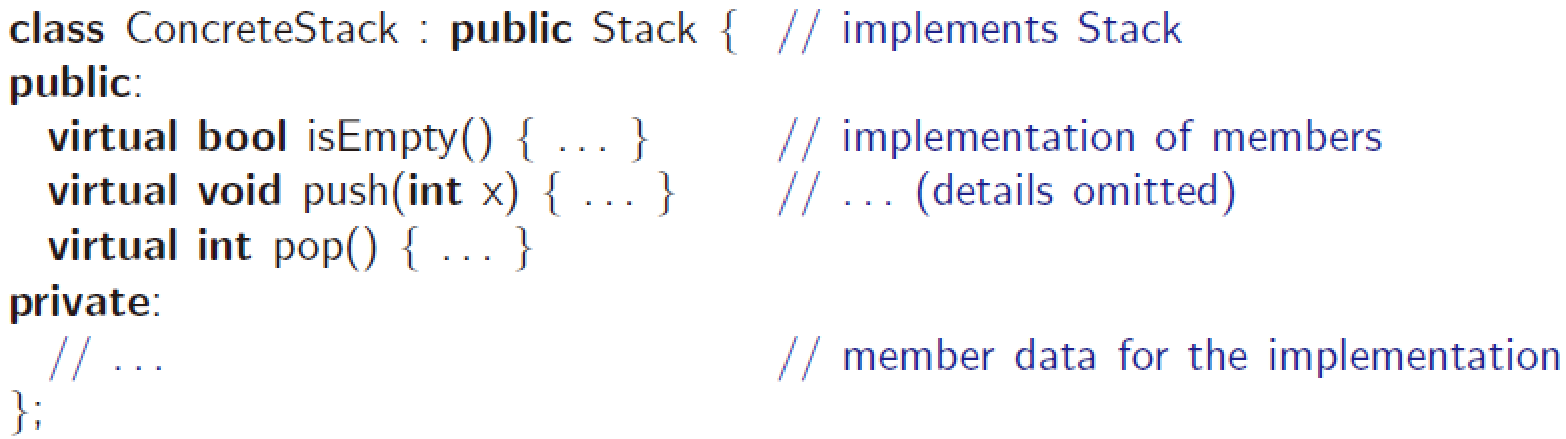
Dynamic Binding and Virtual Functions

- **Abstract Class**
  - A class that is used only as a base class
  - A class instance cannot be created



```
class Stack {                               // stack interface as an abstract class
public:
  virtual bool isEmpty() const = 0;         // is the stack empty?
  virtual void push(int x) = 0;             // push x onto the stack
  virtual int pop() = 0;                    // pop the stack and return result
};
```

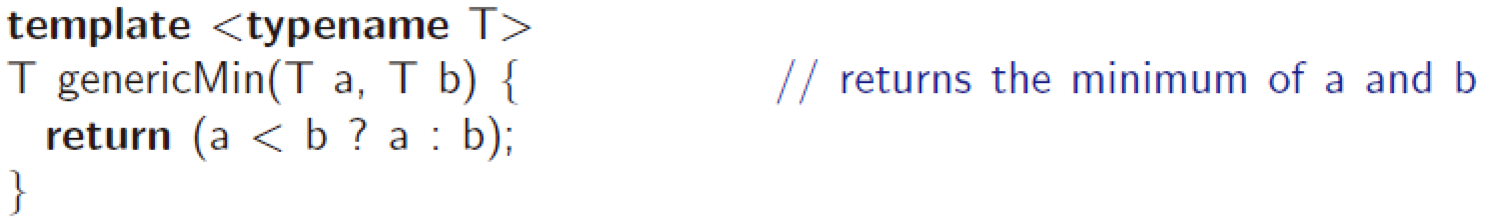
- **Pure Virtual Function**
  - No implementation is provided in the parent class
  - Child classes must implement it



```
class ConcreteStack : public Stack {        // implements Stack
public:
  virtual bool isEmpty() { ... }            // implementation of members
  virtual void push(int x) { ... }          // ... (details omitted)
  virtual int pop() { ... }
private:
  // ...                                    // member data for the implementation
};
```

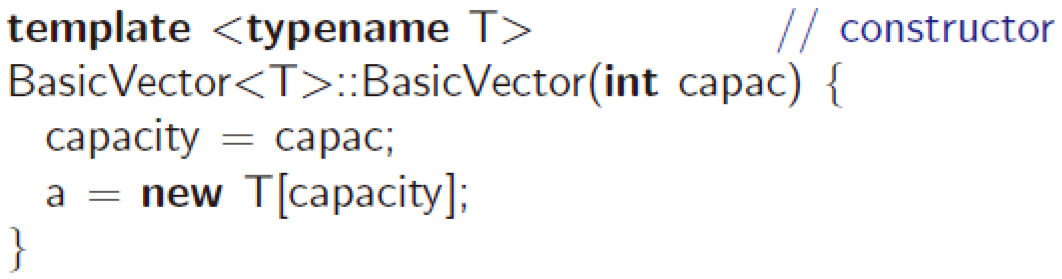
Templates

- **Function Templates**
  - Special functions that can operate with generic types
  - Achieved using template parameters
  - ***template*** and ***typename*** keywords

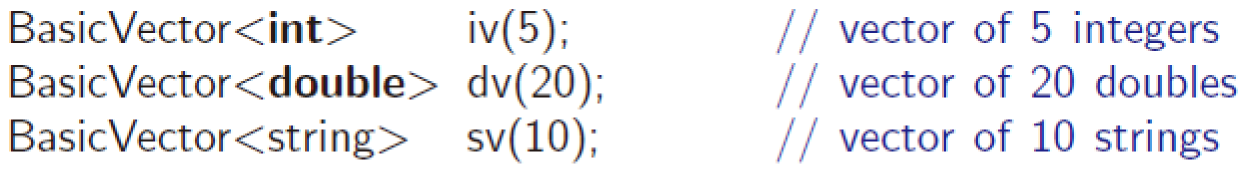


```
template <typename T>
T genericMin(T a, T b) {                    // returns the minimum of a and b
  return (a < b ? a : b);
}
```

- **Class Templates**
  - Can define a class independent of the data type
  - STL uses class templates extensively



```
template <typename T>                 // constructor
BasicVector<T>::BasicVector(int capac) {
  capacity = capac;
  a = new T[capacity];
}
```

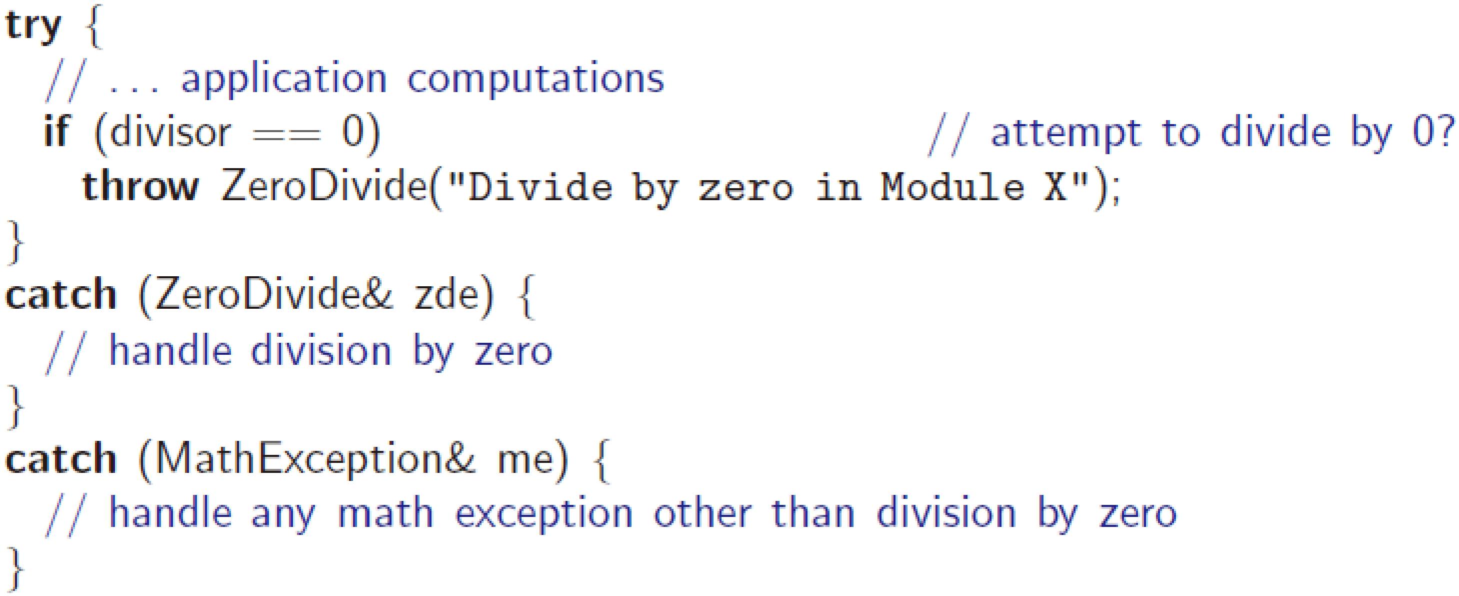


```
BasicVector<int>      iv(5);          // vector of 5 integers
BasicVector<double>   dv(20);         // vector of 20 doubles
BasicVector<string>   sv(10);         // vector of 10 strings
```

Exceptions

- **Exceptions**
  - Unexpected events that occur during the execution
  - Thrown by some unexpected condition
  - Caught by exception handlers or the program is terminated unexpectedly

- **Try-Catch Block**

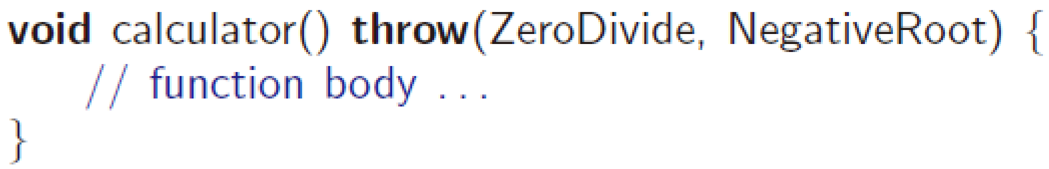


```
try {
  // . . . application computations
  if (divisor == 0)                         // attempt to divide by 0?
    throw ZeroDivide("Divide by zero in Module X");
}
catch (ZeroDivide& zde) {
  // handle division by zero
}
catch (MathException& me) {
  // handle any math exception other than division by zero
}
```

Exceptions

Exception Specification

- A function can specify the exception it might throw



```
void calculator() throw(ZeroDivide, NegativeRoot) {
    // function body ...
}
```

```
void func1();                    // can throw any exception
void func2() throw();            // can throw no exceptions
```

Summary

- Dynamic Memory Allocation
- Control Flow
- Classes
- Inheritance and Polymorphism
- Templates
- Exceptions