C70083 Principles and Practice of Programming

Part III: Object-based C++ Language Features

William Knottenbelt

3 November 2024

Recall the Tennis Player

What attributes might we want to create to reason about the tennis player?

C++ Tennis Player Program Code

```
int main(){
  const char* player_fname = "Lisa";
  const char* player_lname = "Bowls";
  int player_birth_year = 1985;
  int player_birth_month = 1;
  int player_birth_day = 7;

  print_name(player_fname, player_lname);
  //OUTPUT: Lisa Bowls
  print_dob(player_birth_year, player_birth_month, player_birth_day);
  //OUTPUT: 07/01/1985
  return 0;
}
```

What about the umpire?

Now what code do we need to write to include the umpire in our program?

C++ Program with Umpire Included

```
int main(){
  const char* player_fname = "Lisa";
  const char* player_lname = "Bowls";
  int player_birth_year = 1985;
  int player_birth_month = 1;
  int player_birth_day = 7;

  const char* umpire_fname = "Dave";
  const char* umpire_lname = "Pink";
  int umpire_birth_year = 1975;
  int umpire_birth_month = 5;
  int umpire_birth_day = 8;

  print_name(player_fname, player_lname);
  //OUTPUT: Lisa Bowls
  print_dob(player_birth_year, player_birth_month, player_birth_day);
  //OUTPUT: 07/01/1985

  print_name(umpire_fname, umpire_lname);
  //OUTPUT: Dave Pink
  print_dob(umpire_birth_year, umpire_birth_month, umpire_birth_day);
  //OUTPUT: 08/05/1975

  return 0;
}
```

Classes

This is getting messy. Let's implement classes to reduce duplication and complexity:

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day) :
    year(year), month(month), day(day) {}

  void print(){
    cout << ((day<10)? "0": "") << day << "/"
         << ((month<10)? "0": "") << month << "/"
         << year << endl;
  }
};
```

Updated Main Program

```
int main(){
  const char* player_fname = "Lisa";
  const char* player_lname = "Bowls";
  Date player_dob = Date(1985,1,7);

  const char* umpire_fname = "Dave";
  const char* umpire_lname = "Pink";
  Date umpire_dob = Date(1975,5,8);

  print_name(player_fname, player_lname);
  //OUTPUT: Lisa Bowls
  player_dob.print();
  //OUTPUT: 07/01/1985

  print_name(umpire_fname, umpire_lname);
  //OUTPUT: Dave Pink
  umpire_dob.print();
  //OUTPUT: 08/05/1975
  return 0;
}
```

Classes

We can further reduce duplication by introducing a Name object:

```
class Name{
private:
  const char* fname;
  const char* lname;
public:
  Name(const char* fname, const char* lname) :
    fname(fname), lname(lname){}

  void print(){
    cout << fname << " " << lname << endl;
  }
};
```

Updated Main Program

```
int main(){
  Name player_name = Name("Lisa","Bowls");
  Date player_dob = Date(1985,1,7);

  Name umpire_name = Name("Dave","Pink");
  Date umpire_dob = Date(1975,5,8);

  player_name.print();
  //OUTPUT: Lisa Bowls
  player_dob.print();
  //OUTPUT: 07/01/1985

  umpire_name.print();
  //OUTPUT: Dave Pink
  umpire_dob.print();
  //OUTPUT: 08/05/1975
  return 0;
}
```

Classes

And now a Person class:

```
class Person{
private:
  Name name;
  Date dob;
public:
  Person(Name name, Date dob) :
    name(name), dob(dob) {}

  void print_name(){
    name.print();
  }
  void print_dob(){
    dob.print();
  }
};
```

Updated Main Program

```
int main(){
  Person player = Person(Name("Lisa","Bowls"), Date(1985,1,7));
  Person umpire = Person(Name("Dave","Pink"), Date(1975,5,8));

  player.print_name();
  //OUTPUT: Lisa Bowls
  player.print_dob();
  //OUTPUT: 07/01/1985

  umpire.print_name();
  //OUTPUT: Dave Pink
  umpire.print_dob();
  //OUTPUT: 08/05/1975

  player.name.print();
  //error: 'Name Person::name' is private within this context

  return 0;
}
```

UML to C++

Date
- day: int - month: int - year: int
+ print(): void

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day) :
    year(year), month(month), day(day) {}

  void print(){
    cout << ((day<10)? "0": "") << day << "/"
         << ((month<10)? "0": "") << month << "/"
         << year << endl;
  }
};
```

Declaration and Definition

Declaration:

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day);
  void print();
};
```

Definition:

```
Date::Date(int year, int month, int day) :
  year(year), month(month), day(day) {}

void Date::print(){
  cout << ((day<10)? "0": "") << day << "/"
       << ((month<10)? "0": "") << month << "/"
       << year << endl;
}
```

Constructors and Destructors

Node
- word: char[]
+ ptr_to_next_node: Node*
+ get_word(): const char*

Constructor

Node(...) is a constructor, called whenever an object of class Node is instantiated.

```
class Node{
private:
  char word[MAX_WORD_LENGTH];
public:
  Node* ptr_to_next_node;
  Node(char input_word[]);
  const char* get_word() const;
};

Node::Node(char input_word[]){
  strcpy(word, input_word);
  ptr_to_next_node = NULL;
}

const char* Node::get_word() const {
  return word;
}
```

Constructors and Destructors

LinkedList
- start: Node*
- /length : int
+ add(): void
+ get_length(): int

```
class LinkedList{
private:
   Node* start;
public:
   LinkedList();
   ~LinkedList();
   void add(char word[]);
   int get_length();
};
```

Destructor

~LinkedList(); is the declaration of the destructor, called whenever a LinkedList is deleted.

Constructors and Destructors

The add function utilises the previously mentioned Node constructor to create a new node.
N.B. the Node is created on the heap.

```
void LinkedList::add(char input_word[]){
  if(start == NULL){
    start = new Node(input_word);
    return;
  }

  Node* current = start;

  while(current->ptr_to_next_node){
    current = current->ptr_to_next_node;
  }

  current->ptr_to_next_node = new Node(input_word);
}
```

Constructors and Destructors

What happens when we delete the list, with no destructor implemented?

```
int main(){
  LinkedList* list = new LinkedList();
  char word[MAX_WORD_LENGTH];

  cout << "Where are you?" << endl;
  cin >> word;
  list->add(word);

  cout << "How are you?" << endl;
  cin >> word;
  list->add(word);

  cout << "When are you?" << endl;
  cin >> word;
  list->add(word);

  delete list;

  return 0;
}
```

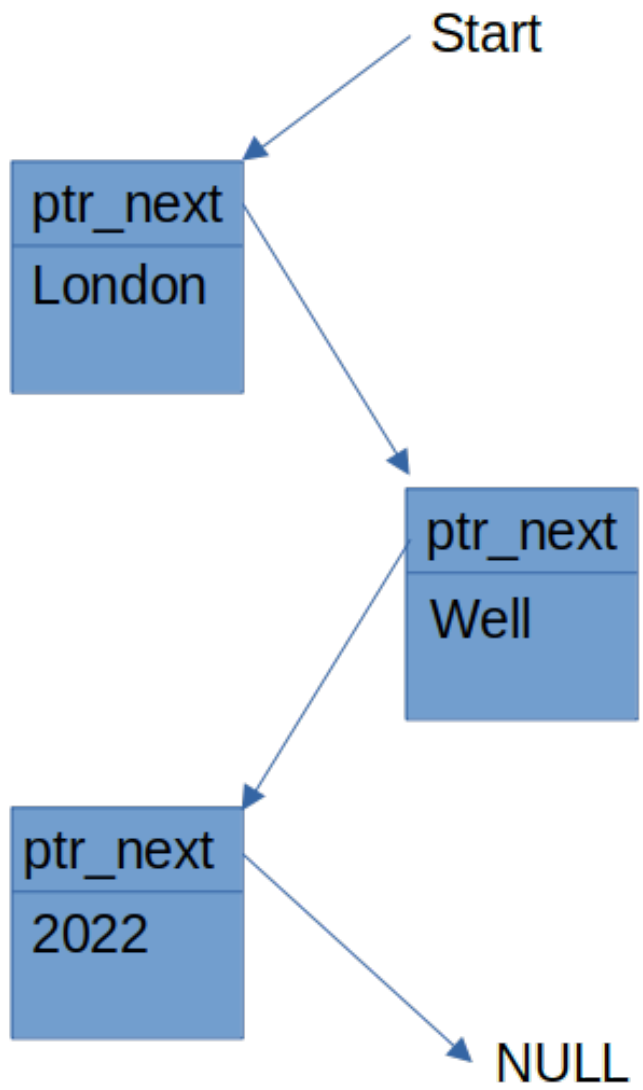
Memory Leak

Running the program with Valgrind:

```
==3768922== Memcheck, a memory error detector
==3768922== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3768922== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3768922== Command: ./run
==3768922==
Where are you?
London
How are you?
Well
When are you?
2022
==3768922==
==3768922== HEAP SUMMARY:
==3768922==     in use at exit: 264 bytes in 3 blocks
==3768922==   total heap usage: 7 allocs, 4 frees, 75,024 bytes allocated
==3768922==
==3768922== LEAK SUMMARY:
==3768922==    definitely lost: 88 bytes in 1 blocks
==3768922==    indirectly lost: 176 bytes in 2 blocks
==3768922==      possibly lost: 0 bytes in 0 blocks
==3768922==    still reachable: 0 bytes in 0 blocks
==3768922==         suppressed: 0 bytes in 0 blocks
==3768922== Rerun with --leak-check=full to see details of leaked memory
==3768922==
==3768922== For lists of detected and suppressed errors, rerun with: -s
==3768922== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

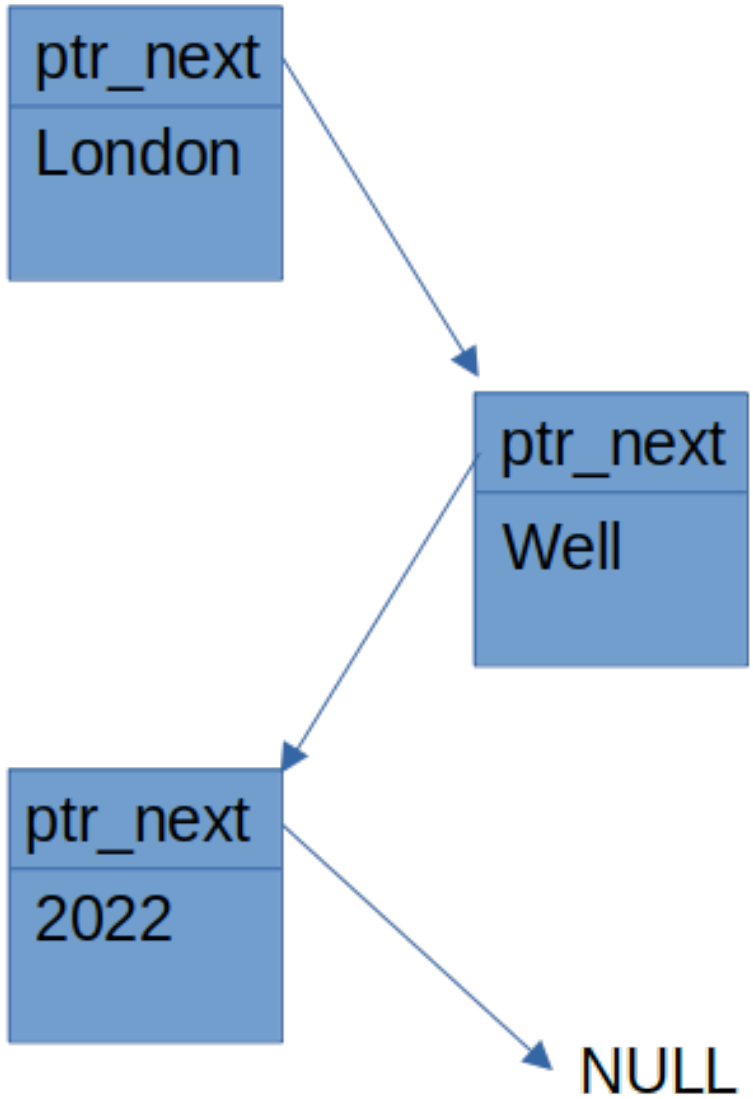
Memory Leak

What's happening? Here is a diagram showing the nodes of the LinkedList. Without a destructor, the Node* **start** pointer is deleted when the list is deleted, but the underlying nodes remain.



Memory Leak

```
int main(){
  LinkedList* list = new LinkedList();
  char word[MAX_WORD_LENGTH];

  cout << "Where are you?" << endl;
  cin >> word;
  list->add(word);

  cout << "How are you?" << endl;
  cin >> word;
  list->add(word);

  cout << "When are you?" << endl;
  cin >> word;
  list->add(word);

  delete list;

  return 0;
}
```



Constructors and Destructors

Now let's define the LinkedList Destructor. Note that the destructor prints the word of each node before deleting the node. This is for our sake, so we can see what's going on. It is not normal practice.

```
LinkedList::~LinkedList(){
  Node_ptr prev = start;
  while(prev){
    cout << "Deleting " << start->get_word() << endl;
    start = prev->ptr_to_next_node;
    delete prev;
    prev = start;
  }
}
```

No Memory Leak

Running the program again with Valgrind:

```
==3772924== Memcheck, a memory error detector
==3772924== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3772924== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==3772924== Command: ./run
==3772924==
Where are you?
London
How are you?
Well
When are you?
2022
Deleting London
Deleting Well
Deleting 2022
==3772924==
==3772924== HEAP SUMMARY:
==3772924==     in use at exit: 0 bytes in 0 blocks
==3772924==   total heap usage: 7 allocs, 7 frees, 75,024 bytes allocated
==3772924==
==3772924== All heap blocks were freed -- no leaks are possible
==3772924==
==3772924== For lists of detected and suppressed errors, rerun with: -s
==3772924== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Getters and Setters

Note how we got access to the **word** in each node: using **start->get_word()**. Why not just use **start->word**?

```
LinkedList::~LinkedList(){
  Node_ptr prev = start;
  while(prev){
    cout << "Deleting " << start->get_word() << endl;
    start = prev->ptr_to_next_node;
    delete prev;
    prev = start;
  }
}
```

Getters and Setters

Recall that **start** is a **Node***. In the **Node** class **word** is private. This is so that external objects won't be able to modify it.

```
class Node{
private:
  char word[MAX_WORD_LENGTH];
public:
  Node* ptr_to_next_node;
  Node(char input_word[]);
  const char* get_word() const;
};

Node::Node(char input_word[]){
  strcpy(word, input_word);
  ptr_to_next_node = NULL;
}

const char* Node::get_word() const {
  return word;
}
```

Instead, we allow read-only access to **word** via the **const** function **get_word()**. This is referred to as a **getter** function.

Getters and Setters

Setters

We may want to allow a private field to be modified via a public setter function, e.g. **set_word(char* new_word);**. If we do this, then why not just make **word** public?

1. Changing the state of a field may require another process to trigger. A setter could define exactly what should happen in this instance.
2. We may require that the new state of the field conforms to some properties. A setter could throw an error for example if the state is modified in an undesirable manner.

Composition

Recall our Person Class from earlier. This class uses **composition** by having a **Name** and **Date** object in its fields.

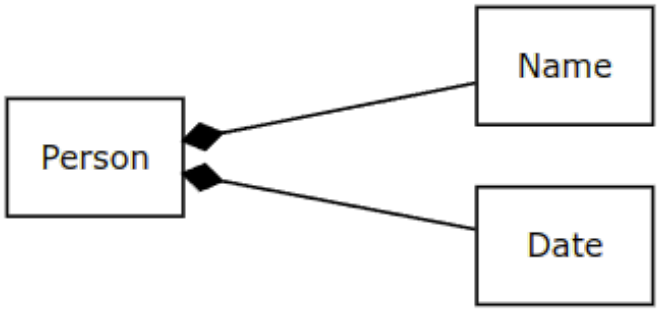


Figure: UML for Person Composition

```
class Person{
private:
  Name name;
  Date dob;
public:
  Person(Name name, Date dob) :
    name(name), dob(dob) {}

  void print_name(){
    name.print();
  }
  void print_dob(){
    dob.print();
  }
};
```

Composition

- Because Person is **composed** of a Name and a Date, whenever a Person is created in memory, the reserved block of memory will be large enough to contain a Name object and a Date object.
- In this context the Person class **owns** the Name object and the Date object.
- These two objects are created at the instantiation of Person and are destroyed when Person is deleted.

Shared Aggregation

Here the Person class is using shared **aggregation** by having a **Name** reference and **Date** reference in its fields.

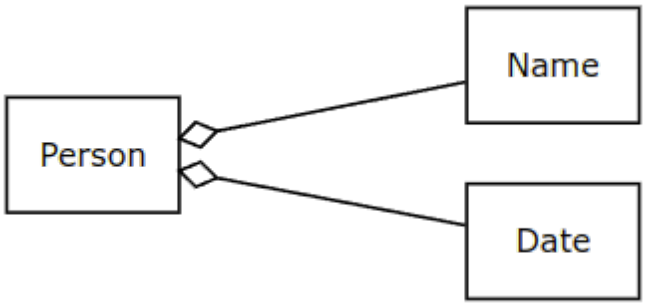


Figure: UML for Person Aggregation

```
class Person{
private:
  Name &name;
  Date &dob;
public:
  Person(Name &name, Date &dob) :
    name(name), dob(dob) {}

  void print_name(){
    name.print();
  }
  void print_dob(){
    dob.print();
  }
};
```

Shared Aggregation

```
int main(){
  Name player_name = Name("Lisa","Bowls");
  Date player_dob = Date(1985,1,7);

  Person player = Person(player_name, player_dob);

  player.print_name();
  //OUTPUT: Lisa Bowls
  player.print_dob();
  //OUTPUT: 07/01/1985

  return 0;
}
```

Figure: Main Program for Person with Shared Aggregation, using references

Shared Aggregation – Using Pointers

Here the Person class is using shared **aggregation** by having a **Name** pointer and **Date** pointer in its fields.

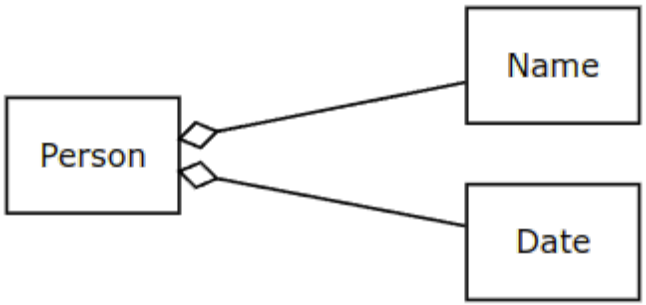


Figure: UML for Person Aggregation

```
class Person{
private:
  Name* name;
  Date* dob;
public:
  Person(Name* name, Date* dob) :
    name(name), dob(dob) {}

  void print_name(){
    name->print();
  }
  void print_dob(){
    dob->print();
  }
};
```

Shared Aggregation – Using Pointers

```
int main(){
  Name* player_name = new Name("Lisa","Bowls");
  Date* player_dob = new Date(1985,1,7);

  Person player = Person(player_name, player_dob);

  player.print_name();
  //OUTPUT: Lisa Bowls
  player.print_dob();
  //OUTPUT: 07/01/1985

  return 0;
}
```

Figure: Main Program for Person with Shared Aggregation, using pointers

Shared Aggregation

- Now Person contains only a reference, or pointer to the objects of class Name and Date.
- This means that when an object of class Person is instantiated, the reserved block of memory will require only an address of a Name and an address of a Date instead of an entire Name object and Date object.
- Under shared aggregation, the Person object does not own the Name and Date object.
- This means that the Name and Date object can be created before the Person object is created and can remain after the Person object is deleted.
- For example, if **player** is deleted, **player_name** and **player_dob** may not be deleted.

Shared Aggregation

```
int main(){
  Name* player_name = new Name("Lisa","Bowls");
  Date* player_dob = new Date(1985,1,7);

  Person player = Person(player_name, player_dob);
  Person umpire = Person(new Name("Dave","Pink"), player_dob);

  player.print_name();
  //OUTPUT: Lisa Bowls
  player.print_dob();
  //OUTPUT: 07/01/1985

  umpire.print_name();
  //OUTPUT: Dave Pink
  umpire.print_dob();
  //OUTPUT: 07/01/1985

  return 0;
}
```

Figure: Here **player_dob** is re-used by **umpire**

Shared Aggregation

- One Date may be **shared** by multiple Persons as shown on the previous slide.
- Additionally the Name for umpire is generated inline. This could lead to a memory leak if name is not deleted by the Person destructor.

Pointer vs Reference

1. In shared aggregation, when should we use a pointer and when should we use a reference?
2. In the **Person** constructor, if **name** is of type **Name***, then we could pass a **null** pointer or a pointer to an actual Name object.
3. However, if **name** was of type **Name&**, then we would have to pass a reference to an actual **Name** object.
4. In this way pointers can be used for **optional** fields, whereas references can be used for **required** fields.

Don't de-reference a null pointer

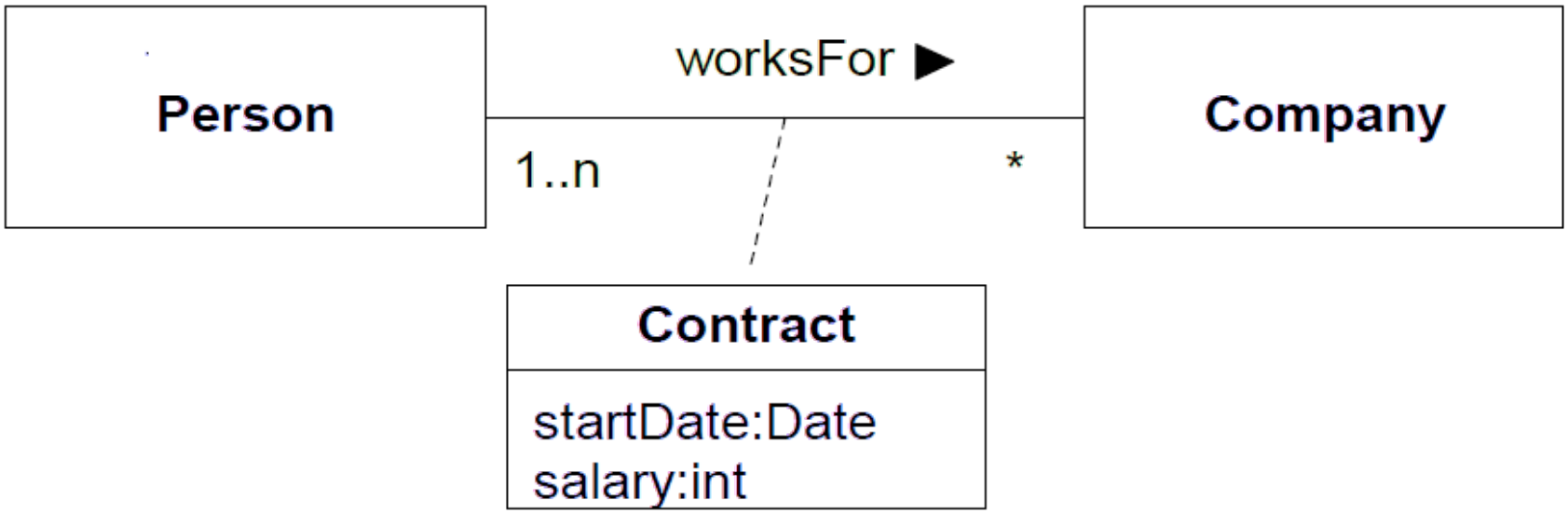
N.B. if pointers are used for fields of a class, then the member functions of the class should check whether the fields are null pointers before de-referencing them.

Association Class

- What if the tennis player needs to enter into a contract with Tennis Inc., a company?
- The contract may have attributes that we need to keep track of. We can create a Contract class to do so.
- However, who owns the contract class? Should it be owned by the Person class? Or should it be owned by the Company? Neither?

Association Class

Recall this example from the last lecture:



Association class – Contract Declaration

Here is an implementation of the Contract Class:

```
class Contract{
private:
  Person& employee;
  Company& employer;
  Date start_date;
  int salary;
public:
  Contract(Person& employee, Company& employer,
           Date start_date, int salary);
  Person& get_employee() const;
  Company& get_employer() const;
};
```

Note that employee and employer are references to a Person and Company respectively. This is because the Contract class doesn't own the employee or employer; the contract could be destroyed without the employee or employer being destroyed.

Association class – Contract Definitions

```
Contract::Contract(Person& employee, Company& employer,
                   Date start_date, int salary) :
  employee(employee), employer(employer),
  start_date(start_date), salary(salary){}

Person& Contract::get_employee() const {
  return employee;
}

Company& Contract::get_employer() const {
  return employer;
}
```

Association class

```
int main(){
  Person player = Person(Name("Lisa","Bowls"), Date(1985, 1, 7));
  Company* business = new Company("Tennis Inc.");
  Date today = Date(2022,1,1);

  Contract tennis_contract = Contract(player, *business, today, 50000);
  ...
```

N.B.

Recall that **Contract** takes a **Person&** and a **Company&** as the first two arguments to its constructor. In our main program, player is a **Person**, while business is a pointer to a **Company**. This is why ***business** is passed as the second argument in the constructor: we must pass the underlying object, rather than a pointer to the object.

Friends and Overloading

Recall our Date class defined earlier. It utilises the member function **print** to print itself to the console.

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day) :
    year(year), month(month), day(day) {}

  void print(){
    cout << ((day<10)? "0": "") << day << "/"
         << ((month<10)? "0": "") << month << "/"
         << year << endl;
  }
};
```

Friends and Overloading

What happens if we try to pass a Date to the '<<' operator?

```
int main(){
  Date player_dob = Date(1985,1,7);

  cout << player_dob << endl;
  //error: no match for 'operator<<'...

  return 0;
}
```

Friends and Overloading

Introducing Friends. The **friend** keyword can be used inside a class to denote that a function is allowed access to the private fields of the class even though it is not a member function.

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day);
  void print();
  friend std::ostream& operator<<(std::ostream& out, Date date);
};

std::ostream& operator<<(std::ostream& out, Date date){
  out << ((date.day<10)? "0": "") << date.day << "/"
      << ((date.month<10)? "0": "") << date.month << "/"
      << date.year;

  return out;
}
```

Insertion Operator

- The insertion operator '<<' denoted by operator<< is a binary function who's first argument is a reference to an **ostream** and second argument could be any class. In the previous slide the second argument is of class **Date**.
- The function returns an ostream. This is how the operator can be strung together in a chain:

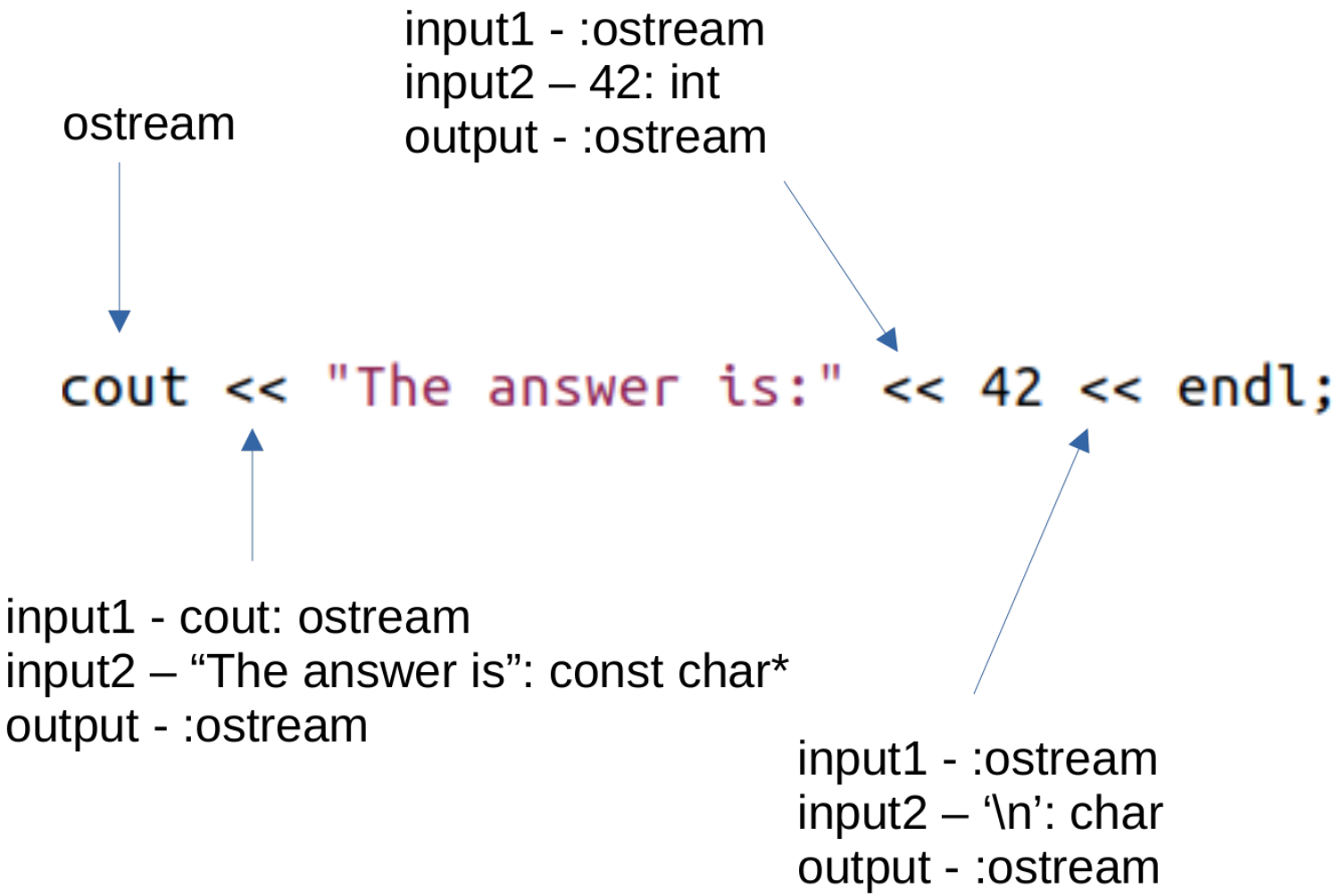
```
cout << "The answer is:" << 42 << endl;
```

NB

The first argument is what appears **before** the operator and the second argument is what appears **after** the operator.

Insertion Operator

Here is an illustration of the inputs and outputs at each part of the chain.



Insertion Operator

- As the operator always returns an ostream and takes an ostream as it's first input, a chain of any combination of classes can be strung together, as long as the function has been defined for each class.

Common Operators

Common operators						
assignment	increment decrement	arithmetic	logical	**comparison**	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`a ? b : c`

Friends and Overloading

Back to our main function, which now works:

```
int main(){
  Date player_dob = Date(1985,1,7);

  cout << player_dob << endl;
  //OUTPUT: 07/01/1985

  return 0;
}
```

Friends and Overloading

What if we pass a pointer?

```
int main(){
  Date* player_dob = new Date(1985,1,7);

  cout << player_dob << endl;
  //OUTPUT: 0x561c616d3eb0

  return 0;
}
```

We haven't defined how the operator should handle **Date*** objects, so it just prints the address that **player_dob** is pointing to.

Friends and Overloading

If we were keen for pointers to have their underlying objects printed, we could just overload the operator again:

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day);
  void print();
  friend std::ostream& operator<<(std::ostream& out, Date date);
  friend std::ostream& operator<<(std::ostream& out, Date* date);
};
```

Friends and Overloading

The implementation will be slightly different, as the function must de-reference the pointer:

```
std::ostream& operator<<(std::ostream& out, Date* date){
  out << ((date->day<10)? "0": "") << date->day << "/"
      << ((date->month<10)? "0": "") << date->month << "/"
      << date->year;

  return out;
}
```

Friends and Overloading

Now a function exists for the insertion operator for both **Date** and **Date*** objects:

```
int main(){
  Date* player_dob = new Date(1985,1,7);

  cout << player_dob << endl;
  //OUTPUT: 07/01/1985

  cout << *player_dob << endl;
  //OUTPUT: 07/01/1985

  return 0;
}
```

Friends and Overloading: another example

What happens if we try to increment a Date object?

```
int main(){
  Date player_dob = Date(1985,1,7);

  cout << player_dob << endl;
  //OUTPUT: 07/01/1985

  player_dob++;
  // error: no 'operator++(int)' declared...

  cout << player_dob << endl;

  return 0;
}
```

Friends and Overloading

Let's override the operator, this time implemented as a member function (as opposed to friend).

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day);
  void print();
  friend std::ostream& operator<<(std::ostream& out, Date date);
  friend std::ostream& operator<<(std::ostream& out, Date* date);
  void operator++(int);
};
```

Friends and Overloading

Here is the definition:

```
void Date::operator++(int){
  day++;
  //simplification:
  if(day == 32){
    month++;
    day = 1;
  }
  if(month == 13){
    year++;
    month = 1;
  }
}
```

Friends and Overloading

Now it works.

```
int main(){
  Date player_dob = Date(1985,1,7);

  cout << player_dob << endl;
  //OUTPUT: 07/01/1985

  player_dob++;

  cout << player_dob << endl;
  //OUTPUT: 08/01/1985

  return 0;
}
```

Friends and Overloading

- However, the increment operator should not have a void return type.
- Here is an example use of the **int** increment operator.
- Note that **holder** is assigned the original value of **number** (before **number** has been incremented).

```
int main(){
  int number = 5;
  int holder;

  cout << number << endl;
  //OUTPUT: 5

  holder = number++;

  cout << holder << endl;
  //OUTPUT: 5

  cout << number << endl;
  //OUTPUT: 6

  return 0;
}
```

Friends and Overloading

Let's re-define our Date increment operator, so that it returns a Date object.

```
class Date{
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day);
  void print();
  friend std::ostream& operator<<(std::ostream& out, Date date);
  friend std::ostream& operator<<(std::ostream& out, Date* date);
  Date operator++(int);
};
```

Friends and Overloading

Now we make a copy of the Date object before incrementing the Date and return the copy at the end.

```
Date Date::operator++(int){
  Date copy = *this;

  day++;
  //simplification:
  if(day == 32){
    month++;
    day = 1;
  }
  if(month == 13){
    year++;
    month = 1;
  }
  return copy;
}
```

The 'this' pointer

this is a keyword that acts as a pointer to the object it is in. In this context **this** is a pointer to the Date object, so ***this** is the object itself.

This Pointer

These two implementations are equivalent:

```
Date Date::operator++(int){
  Date copy = *this;

  this->day++;
  //simplification:
  if(this->day == 32){
    this->month++;
    this->day = 1;
  }
  if(this->month == 13){
    this->year++;
    this->month = 1;
  }
  return copy;
}
```

```
Date Date::operator++(int){
  Date copy = *this;

  day++;
  //simplification:
  if(day == 32){
    month++;
    day = 1;
  }
  if(month == 13){
    year++;
    month = 1;
  }
  return copy;
}
```

Guess The Output

```
int main(){
  Date today = Date(2022,1,1);

  cout << today << endl;


  Date capture = today++;

  cout << today << endl;


  cout << capture << endl;


  cout << capture++ << " " << capture << endl;


  return 0;
}
```

Guess The Output

```
int main(){
  Date today = Date(2022,1,1);

  cout << today << endl;
  //OUTPUT: 01/01/2022

  Date capture = today++;

  cout << today << endl;


  cout << capture << endl;


  cout << capture++ << " " << capture << endl;


  return 0;
}
```

Guess The Output

```
int main(){
  Date today = Date(2022,1,1);

  cout << today << endl;
  //OUTPUT: 01/01/2022

  Date capture = today++;

  cout << today << endl;
  //OUTPUT: 02/01/2022

  cout << capture << endl;


  cout << capture++ << " " << capture << endl;


  return 0;
}
```

Guess The Output

```
int main(){
  Date today = Date(2022,1,1);

  cout << today << endl;
  //OUTPUT: 01/01/2022

  Date capture = today++;

  cout << today << endl;
  //OUTPUT: 02/01/2022

  cout << capture << endl;
  //OUTPUT: 01/01/2022

  cout << capture++ << " " << capture << endl;


  return 0;
}
```

Guess The Output

```
int main(){
  Date today = Date(2022,1,1);

  cout << today << endl;
  //OUTPUT: 01/01/2022

  Date capture = today++;

  cout << today << endl;
  //OUTPUT: 02/01/2022

  cout << capture << endl;
  //OUTPUT: 01/01/2022

  cout << capture++ << " " << capture << endl;
  //OUTPUT: 01/01/2022 02/01/2022

  return 0;
}
```

When to use const, *, & for parameters.

Modified	Required		Optional
	Cheap	Expensive	Cheap/Expensive
N	value	const&	const*
Y	&	&	*

- If a function modifies a parameter, then it must be passed by reference or pointer.
- Else, if a parameter is cheap to copy, then it can simply be passed by value.
- Else, if a parameter is expensive to copy, but is not supposed to be modified, then it must be const.
- Generally, if a parameter is optional, it should be a pointer, otherwise it can be a reference.

Const

What does **const** mean in this context?

```
Person& Contract::get_employee() const {
  return employee;
}
```

- **get_employee** is a member function of **Contract**.
- The const keyword is a promise not to modify any fields of the **Contract** class.
- Doing so will result in a compiler error.
- This helps ensure that fields of the class are not modified by mistake.
- Comparison operators, e.g. $<$, $>$, $==$ should be marked **const** for example.

Good C++ Programming Practices

- Every class should have (at least one) constructor, and (if necessary) a destructor
- Constructors should ensure the integrity of the objects produced
- It is the programmer's responsibility to destroy dynamically created objects; it is the system's responsibility to destroy declared local and static objects.

Good C++ Programming Practices (cont.)

- Make private as many data members as possible.
- Some member functions should be private too.
- Guiding principle: disseminate information on a "need to know" basis.

Good C++ Programming Practices (cont.)

- Use const annotations whenever they are appropriate.
- Attach operators to classes if possible; write global functions only as a last resort.
- Some operators are expected to take constant reference parameters; consult the C++ standard to check.

Good C++ Programming Practices (cont.)

- Do not compromise on design elegance in order to just get the program to run.
- Remember your UML diagram will not express *all* aspects of the specification.
- Your C++ code will likely contain more functions and more attributes than your UML diagram.

The MSc C++ Programming Life Cycle

1. Read the problem description
2. Draw the UML class diagram
3. Design the C++ program (headers, main function)
4. For each class:
   - Specify behaviour of each member function
   - Write member function bodies
   - Write a test function
5. Integrate and test
6. If tests fail go back to first step