CS100
Introduction to Programming

Lecture 29, Part II:

Summary on C++

Basics

Object-Oriented Programming

- in OOP, code and data are combined into a single entity called a ***class***
  - each ***instance*** of a given class is an ***object*** of that class type

- principles of Object-Oriented Programming
  - encapsulation
  - inheritance
  - polymorphism

OOP: Encapsulation

- ***encapsulation*** is a form of information hiding and abstraction
- data and functions that act on that data are grouped together (inside a class)

- *ideal:* separate the interface/implementation so that you can use the former without any knowledge of the latter

OOP: Inheritance

- ***inheritance*** allows us to create and define new classes from an existing class (i.e. sub-classes)

- this allows us to re-use code
  - faster implementation time
  - fewer errors
  - easier to maintain/update

OOP: Polymorphism

- ***polymorphism*** is when a single name can have multiple meanings
  - normally used in conjunction with inheritance
  - ability to decide at runtime what will be done

- Different forms of polymorphism in C++
  - overloading functions
  - virtual functions
  - templates

Basic C++ syntax

```
class Date {
public:
    int m_month;
    int m_day;
    int m_year;
};
```

Using Public, Private, Protected

- **`public`**
  - anything that has access to a **`Date`** object also has access to all public member variables and functions

- not normally used for variables;
  used for most functions

- need to have at least one item be public

Using Public, Private, Protected

- **`private`**
  - private members variables and functions can only be accessed by ***member functions*** (cannot be accessed in main(), etc.)

- if not specified, members default to private
  - should specify anyway – good coding practices!

Using Public, Private, Protected

- **`protected`**
  - protected member variables and functions can only be accessed by ***member functions***, and by <u>member functions of any derived classes</u>

Member Function Types

- Many classifications.
- Example:
  - accessor functions
  - mutator functions
  - auxiliary functions

Member Functions: Accessor

- *convention*: start with **`Get`**
- allow retrieval of private data members

- examples:

```
int GetMonth();
int GetDay();
int GetYear();
```

Member Functions: Mutator

- *convention*: start with **`Set`**
- allow changing the value of a private data member

- examples:

```
void SetMonth(int m);
void SetDay(int d);
void SetYear(int y);
```

Member Functions: Auxiliary

- provide support for the operations
  - public if generally called outside function
  - private/protected if only called by member functions

- examples:

```
void OutputMonth();    → public
void IncrementDate();  → private
```

Access Specifiers for Date Class

```
class Date {
public:
  void OutputMonth();
  int  GetMonth();
  int  GetDay();
  int  GetYear();
  void SetMonth(int m);
  void SetDay  (int d);
  void SetYear (int y);
private:
  int m_month;
  int m_day;
  int m_year;
};
```

Constructors

- special ***member functions*** used to create (or “construct”) new objects

- automatically called when an object is created
  - implicit: `Date today;`
  - explicit: `Date today(10, 15, 2014);`

- initializes the values of all data members

Overloading

- we can define multiple versions of the constructor – we can ***overload*** it

- different constructors for:
  - when all values are known
  - when no values are known
  - when some subset of values are known

Example: Date Constructors

```
Date::Date (int m, int d, int y);
Date::Date (int m, int d);
Date::Date ();
```

Avoiding Multiple Constructors

- defining multiple constructors for different sets of known values is a lot of unnecessary code duplication

- we can avoid this by setting ***default parameters*** in our constructors

Default Parameters

- in the ***function prototype*** <u>only</u>, provide default values you want the constructor to use

```
Date (int m = 10, int d = 15,
      int y = 2014);
```

Default Constructors

- a ***default constructor*** is provided by compiler

Default Constructors

- **but**, if you create **any** other constructor, the compiler doesn’t provide a default constructor

- so if you create a constructor, make a default constructor too, even if its body is just empty

```
Date::Date ()
{
   /* empty */
}
```

More on constructors

- When making a class instance, the default constructor of its fields are invoked

```
class Integer {
public:
  int m_val;
  Integer() {
    m_val = 0; printf("Integer default constructor\n");
  }
};
class IntegerWrapper {
public:
  Integer m_val;
  IntegerWrapper() {
    printf("IntegerWrapper default constructor\n");
  }
};

int main() {
  IntegerWrapper q;
}
```

Output:
Integer default constructor
IntegerWrapper default constructor

Object Relationships

- Two types of object relationships
- The “is-a” relationship
  - inheritance
- The “has-a” relationship
  - composition
  - aggregation

both are forms of association

Inheritance Relationship Code

```
class Car: public Vehicle {




} ;
```

Car inherits from the Vehicle class

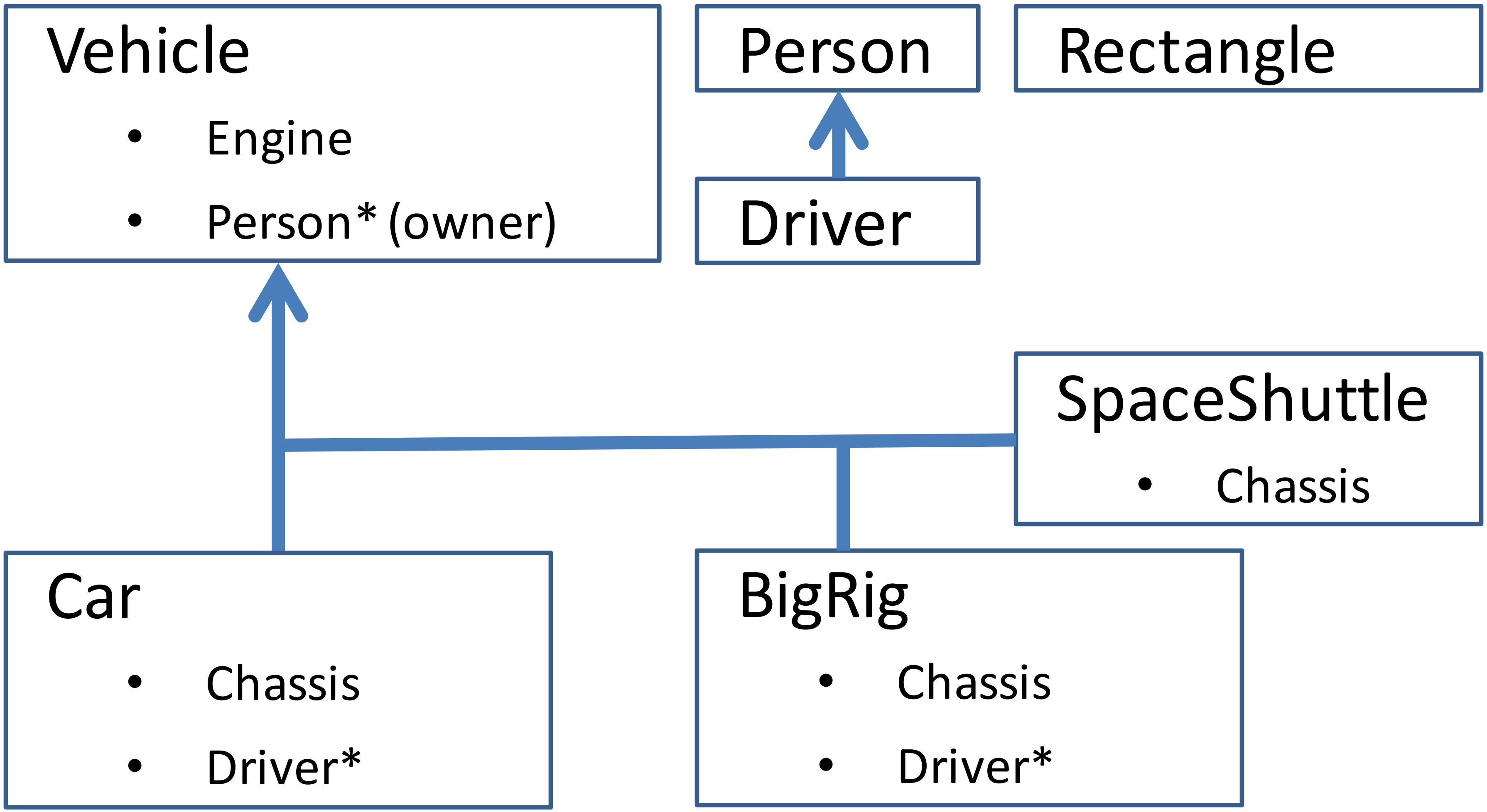
Composition Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (composition)
    Chassis m_chassis;
} ;
```

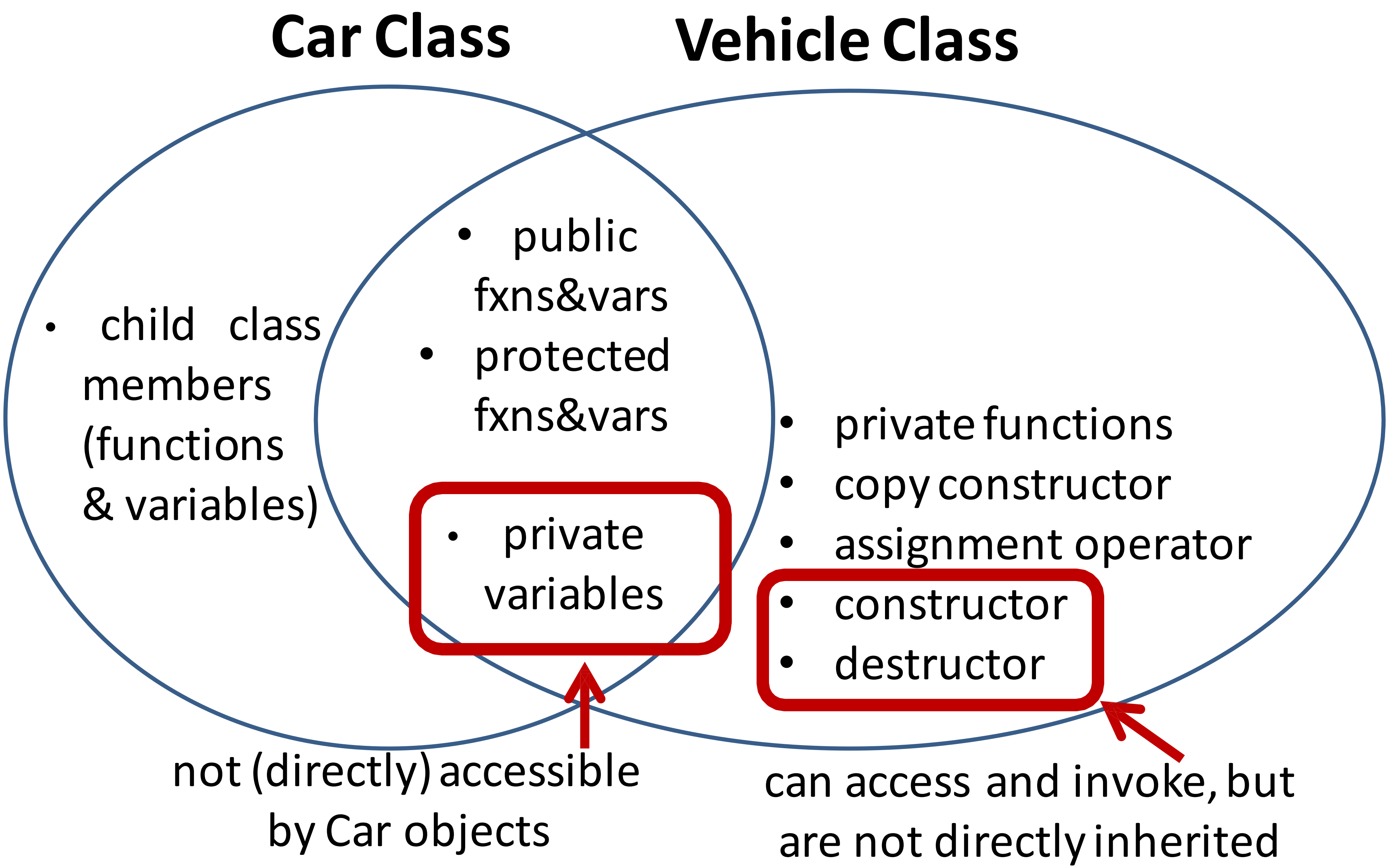
Aggregation Relationship Code

```
class Car: public Vehicle {
  public:
    //functions
  private:
    // member variables, etc.

    // has-a (aggregation)
    Driver *m_driver;
} ;
```

Visualizing Object Relationships



What is Inherited



Handling Access

- child class has access to parent class's:
  - public member variables/functions
  - protected member variables/functions
  - but *not* private member variables/functions
- how should we set the access modifier for parent member variables we want the child class to be able to access?

Handling Access

- we should <u>not</u> make these variables protected!

- leave them private!
- instead, child class uses protected functions when interacting with parent variables
  - mutators
  - accessors

Overloading vs Overriding

- ***overloading***
  - use the same function name, but with different parameters for each overloaded implementation

- ***overriding***
  - use the same function name and parameters, but with a different implementation
  - child class method “hides” parent class method
  - **only possible by using inheritance**

Attempted Overloading Example

- Car class attempts to **overload** a function Move(double distance) with new parameters

```
void Car::Move(double distance,
               double avgSpeed)
{
  // new overloaded Car-only code
}
```

- but this does something we weren't expecting!

Precedence

- **overriding takes precedence over overloading**
  - instead of *overloading* the Move() function, the compiler assumes we are trying to *override* it

- declaring `Car::Move(2 parameters)`
- overrides `Vehicle::Move(1 parameter)`

- we no longer have access to the original `Move()` function from the Vehicle class

Overloading in Child Class

- to overload, we must have both original and overloaded functions in child class

```
void Car::Move(double distance);
void Car::Move(double distance,
               double avgSpeed);
```

- the “original” one parameter function can then explicitly call parent function

What is Polymorphism?

- ability to manipulate objects in a **type-independent** way
- already done to an extent via ***overriding***
  - child class overrides a parent class function
- can take it further using subtyping, AKA ***inclusion polymorphism***

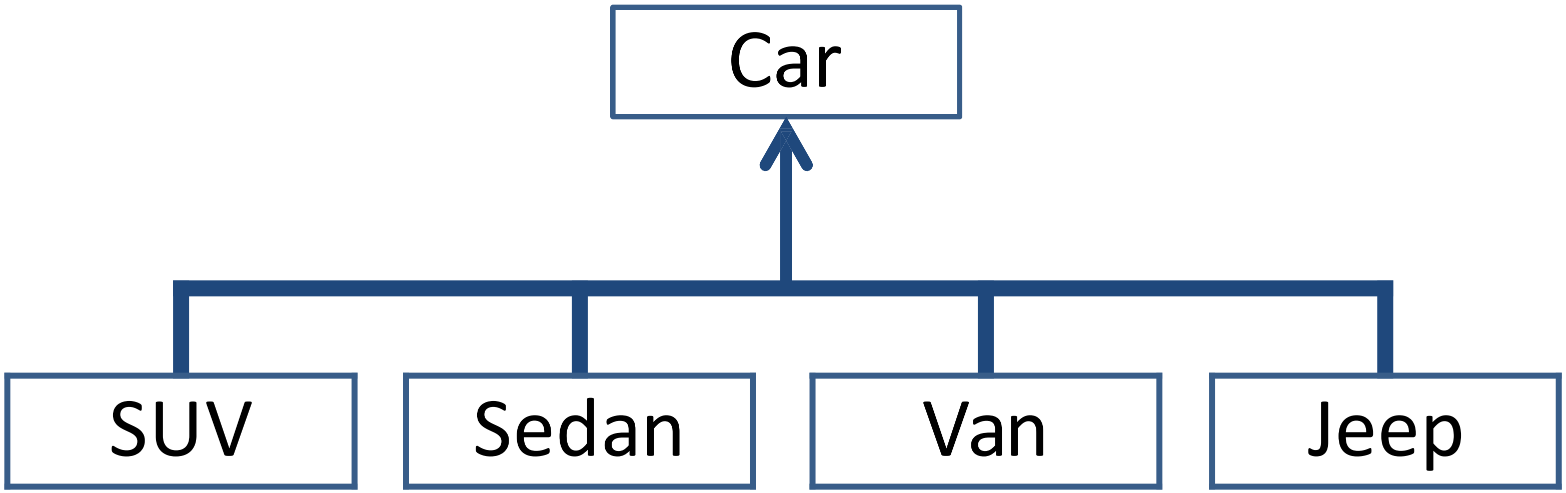
Using Polymorphism

- a pointer of a parent class type can point to an object of a child class type

```
Vehicle *vehiclePtr = &myCar;
```

- why is this valid?
  - because **`myCar`** is-a **`Vehicle`**

Car Example




```
class SUV:   public Car {/*etc*/};
class Sedan: public Car {/*etc*/};
class Van:   public Car {/*etc*/};
class Jeep:  public Car {/*etc*/};
```

Polymorphism: Car Rental

```
vector <Car*> rentalList;
```

vector of **`Car*`** objects

SUV	**SUV**	**Jeep**	**Van**	**Jeep**	**Sedan**	**Sedan**	**SUV**

- can populate the vector with any of **`Car`**'s child classes

Virtual Functions

- can grant access to child methods by using ***virtual functions***
- virtual functions are how C++ implements ***late binding***
  - used when the child class implementation is unknown or variable at parent class creation time

Late Binding

- simply put, binding is determined at run time
  - as opposed to at compile time

- in the context of polymorphism, you're saying

  I don't know for sure how this function is going to be implemented, so wait until it's used and then get the implementation from the object instance.

Using Virtual Functions

- declare the function in the parent class with the keyword **`virtual`** in front

```
virtual void Drive();
```

- only use **`virtual`** with the prototype

```
// don't do this
virtual void Vehicle::Drive();
```

Function Types – Virtual

```
virtual void Drive();
```

- parent class **must** have an implementation
  - even if it's trivial or empty
- child classes may override if they choose to
  - if not overridden, parent class definition used

Function Types – Pure Virtual

```
virtual void Drive() = 0;
```

- denote pure virtual by the " `= 0`" at the end
- the parent class has **no implementation** of this function
  - child classes **must** have an implementation
  - parent class is now an ***abstract class***

Virtual Destructors

```
Vehicle *vehicPtr = new Car;
delete vehicPtr;
```

- for any class with virtual functions, you must declare a virtual destructor as well

- non-virtual destructors will only invoke the base class's destructor

C++ Templates

- Support generic programming
  - develop reusable software components (e.g. function, class)
- Template uses generic data type T
  - Replaced by concrete type at compile type
  - Enables “on-the-go” construction of a member of a family of functions and classes that perform the same operation on different data types
    - functions → function templates
    - classes → class templates

Function Templates

- For functions of considerable importance which have to be used frequently with different data types
- Simple solution:
  - Many functions each operating on one data type only
- Better solution:
  - Defining one function template (i.e. generic function)
- Syntax:

```
template <class T, … >
returntype function_name (arguments)
{
    /* Body of function */
}
```

Class Templates

- Class template
  - generalized to hold/operate on different data types

- Syntax:

```
template <class T1, class T2, …..>
class class_name
{
…
     T1 m_data1;     // data items of template type
     // functions of template argument
     void func1 (T1 a, T2 & b);
     T1 func2 (T2 * x, T2 * y);
};
```

Inheritance of Class Template

Through one of the following techniques:

- Derive a class template from a base class, which is a template class (more template parameters may be added)

```
template <class T1, …>
class derivedclass : public baseclass<T1,…> {
  // member data and functions
};
```

- Derive a class from a base class, which is a template class and restrict the template feature, so that the derived class and its derivatives do not have the template feature

```
class derivedclass : public baseclass<T1,…> {
  // member data and functions
};
```

Where to put templates?

- Templates are no concrete implementations!
- They are just a template!
- Concrete implementations are derived on demand at compile (in the background)

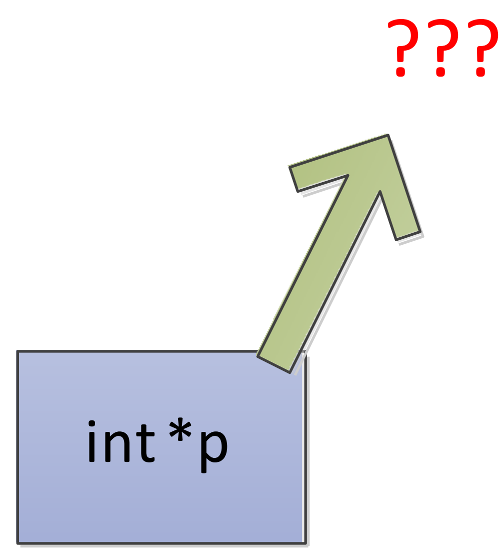
→ Put templates into a header-files!

Scoping and Memory

- When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value
  - Here, p has become a **dangling pointer** (points to memory whose contents are undefined)

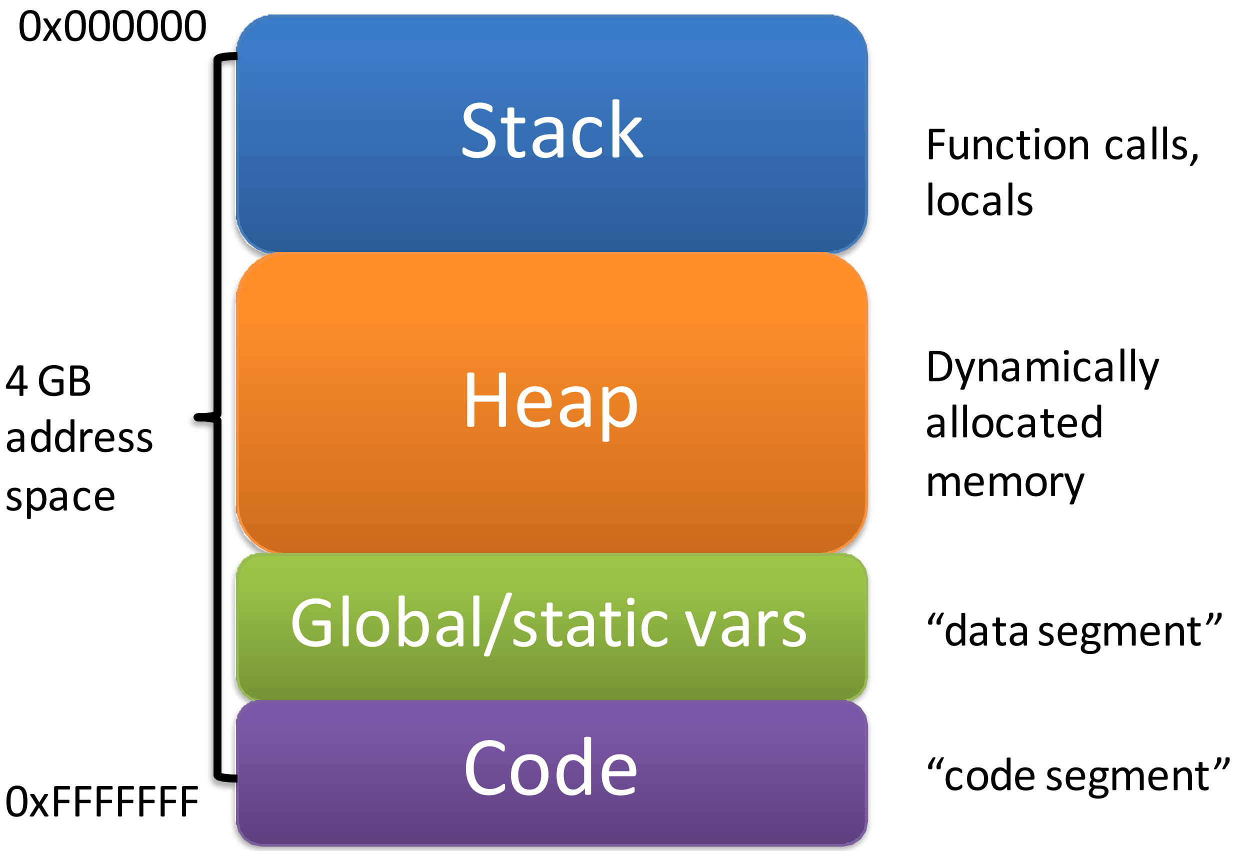
```
int main() {
  int *p;
  if (true) {
    int x = 5;
    p = &x;
  }
  printf("%d\n", *p); // ???
}
```

here



Memory Types

- each process gets its own memory chunk, or ***address space***



Stack Allocation

- memory allocated by the program as it runs
  - local variables
  - function calls
- fixed at compile time

Heap Allocation

- dynamic memory allocation
  - memory allocated at run-time

The new operator

- A way to allocate memory, where the memory will remain allocated until you manually de-allocate it
- Returns a pointer to the newly allocated memory:
  - If using **int x;** the allocation occurs on **the stack**
  - If using **new int;** the allocation occurs **the heap**

The delete operator

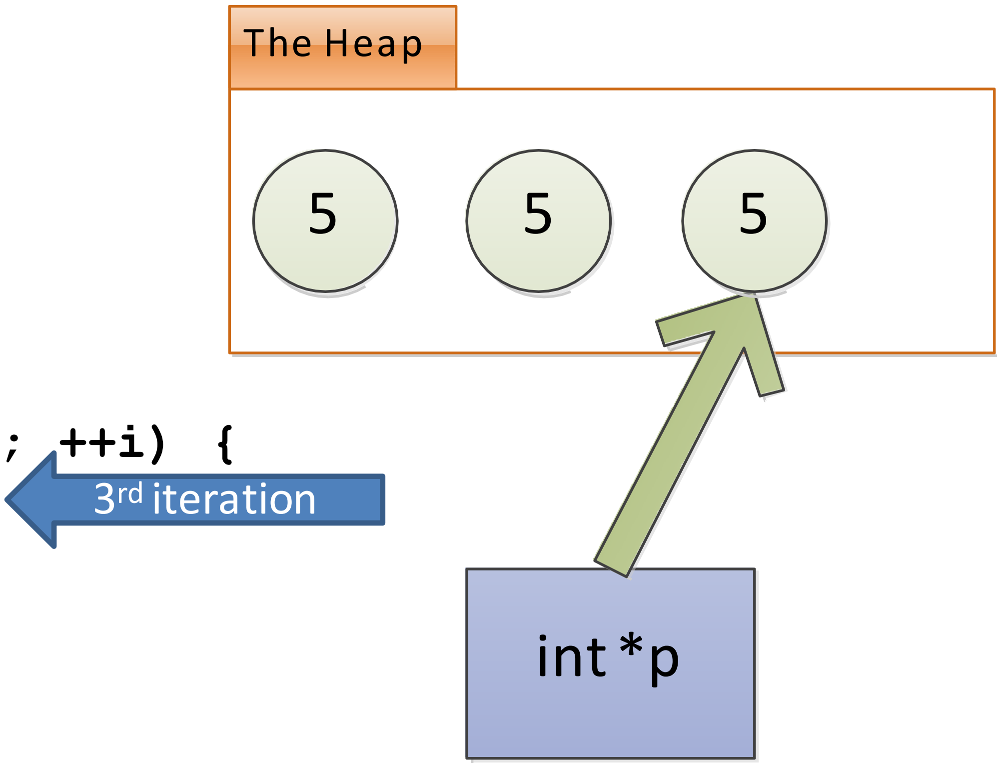
- De-allocates memory that was previously allocated using **new**
- Takes a pointer to the memory location

```
int *x = new int;
// use memory allocated by new
delete x;
```

Delete Memory When Done Using It

- When your program allocates memory but is unable to de-allocate it, this is a **memory leak**

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *p;
  for (int i = 0; i < 3; ++i) {
    p = getPtrToFive();
    printf("%d\n", *p);
  }
}
```



Don’t Use Memory After Deletion

(Segmentation fault)

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  delete x;
  printf("%d\n", *x); // ???
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
}
```

Don’t delete memory twice

incorrect

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
  delete x;
}
```

correct

```
int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
}

int main() {
  int *x = getPtrToFive();
  printf("%d\n", *x); // 5
  delete x;
}
```

Only delete if memory was allocated by new

incorrect

```
int main() {
  int x = 5;
  int *xPtr = &x;
  printf("%d\n",    *xPtr);
  delete xPtr;
}
```

correct

```
int main() {
  int x = 5;
  int *xPtr = &x;
  printf("%d\n", *xPtr);
}
```

Allocating Arrays

- If we use **new[]** to allocate arrays, they can have variable size
- De-allocate arrays with **delete[]**

```
int numItems;
printf("how many items?\n");
scanf("%d", &numItems);
int *arr = new int[numItems];
delete[] arr;
```

Overriding the default copy constructor!

```
class IntegerArray {
public:
  int *m_data; int m_size;
  IntegerArray(int size) {
    m_data = new int[size];
    m_size = size;
  }
  IntegerArray(IntegerArray &o) {
    m_data = new int[o.m_size];
    m_size = o.m_size;
    for (int i = 0; i < m_size; ++i)
      m_data[i] = o.m_data[i];
  }
  ~IntegerArray() {
    delete[] m_data;
  }
};
int main() {
  IntegerArray a(2);
  a.m_data[0] = 4; a.m_data[1] = 2;
  if (true) {
    IntegerArray b = a;
  }
  printf("%d\n", a.m_data[0]); // 4
}
```

Passing by reference

- Pass variable in C++ without copying
  - Pass-by-reference:

```
void functionX( double & val ) {
    val = …; //no de-referencing
}
```

Passing by reference

- References are:
  - Valid types, just like pointers
  - Internally: just a pointer
  - Easier to manipulate
    - No de-referencing needed
  - Safer
    - can only be initialized from a valid instance of an object

Class member references

- Use an initializer list!

```
class Car {
public:
    Car( Driver & driver );
private:
    Driver & m_driver;
};

Car::Car( Driver & driver ) : m_driver(driver)
{}
```

Initializer lists must be used for any members that do not have a default constructor!

What are operators?

+			-	*	/	%
^		&	|	~	!	&&
||		++	--	<<	>>	,
<		<=	==	!=	>	>=
=		+=	-=	*=	/=	%=
&=	|=	^=	<<=		>>=	
[]		()	->	new		delete

What are operators?

- Example of an operator (global operator)

```
class Box {
public:
      Box (int v) : value(v) {}
      int value;
};

// define meaning of comparison for boxes
bool operator< (Box & left, Box & right) {
      return left.value < right.value;
}
```

- Binary comprison operator!

The Increment and Decrement Operators

```
class Box {
public:
        Box (int v) : value(v) { }

        // prefix versions (++someBox)
        int operator++ () { value++; return value; }
        int operator-- () { value--; return value; }

        int operator++ (int) // postfix versions (someBox++)
        {
                int result = value; // step 1, save old value
                value++;            // step 2, update value
                return result;      // step 3, return original
        }
        int operator -- (int) {
                int result = value;
                value--;
                return result;
        }
private:
        int value;
};
```

Part 2: STL

Strings

- In C we used char* to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string
- Need to understand basic string handling
  - Construction
  - Concatentation
  - Comparison …

Input/Output in C++

```
#include <stdio.h>
#include <iostream>
using namespace std;
printf("test: %d\n", x);
cout << "test: " << x << endl;

scanf("%d", &x);
cin >> x;
```

The << Operator

- insertion operator → used along with **`cout`**

- separate each "type" of thing we print out

```
int x = 3;
cout << "X is: " << x
<< "; squared "
<< x*x << endl;
```

The >> Operator

- extraction operator → used with **`cin`**
  - returns a boolean for (un)successful read

- like scanf and fscanf, skips leading whitespace, and stops reading at next whitespace

- don't need to use ampersand on variables

```
cin >> firstName >> lastName >> age;
```

Reading In Files in C++

~~`FILE *ifp;`~~

`ifstream inStream;`

~~`ifp = fopen(“testFile.txt”, “r”);`~~

`inStream.open(“testFile.txt”);`

~~`if ( ifp == NULL ) { /* exit */ }`~~

`if (!inStream) { /* exit */ }`

- Check to make sure file was opened

Writing To Files in C++

- `ofstream outStream;`
  - Declare an output file variable

- `outStream.open("testFile.txt");`
  - Open a file for writing

- `if (!outStream) { /* exit */ }`
  - Check to make sure file was opened

Using File Streams in C++

- once file is correctly opened, use your **`ifstream`** and **`ostream`** variables the same as you would use **`cin`** and **`cout`**

```
 inStm >> firstName >> lastName;

outStm << firstName << " "
 << lastName << endl;
```

Finding EOF with ifstream

- use a “priming read”

```
inStream >> x;

while( !inStream.eof() )
{
  // do stuff with x

  // read in next x
  inStream >> x;
}
```

Using File Streams in C++

- Example of reading line-by-line, and element-by-element

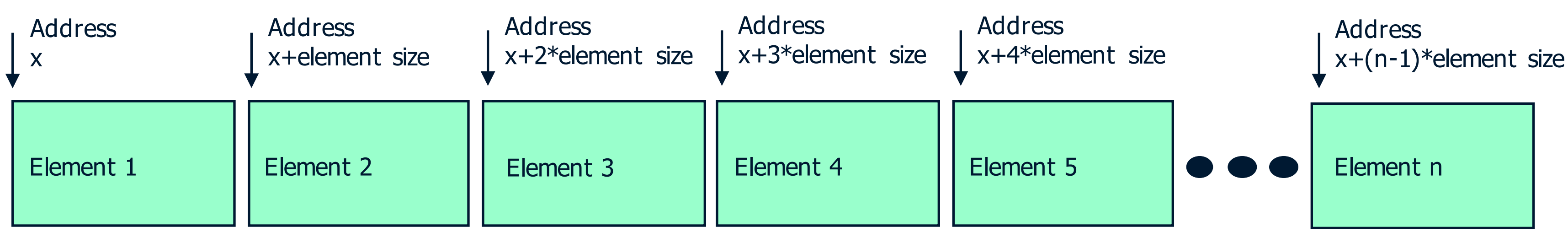
```
while( inStm.good() ) {
    std::string oneLine;
    std::getline( inStm, oneLine );

    std::stringstream lineStm(oneLine);
    while( lineStm.good() ) {
          std::string copy;
          lineStm >> copy;
          std::cout << copy << " ";
    }
    std::cout << "\n";
}
```

Standard Template Library

- Uses template mechanism for generic …
  - … containers (classes)
    - Data structures that hold **<u>anything</u>**
    - **<u>Ex.:</u>** `list, vector, map, set`
  - … algorithms (functions)
    - handle common tasks (searching, sorting, comparing, etc.)
    - **<u>Ex.:</u>** `find, merge, reverse, sort, count, random shuffle, remove, nth-element, rotate,` …

std::vector<T>

- Provides an alternative to the built in array
- A vector is self grown (dynamic in size)
- Use it instead of the built in array!
- Contiguous placement in memory
  - Constant-time look-up given known element size!
  - Expensive when adding/removing elements!



Example

```
int main() {
        int input;
        vector<int> ivec;

        // input
        while (cin >> input )
                ivec.push_back(input);

        // sorting
        sort(ivec.begin(), ivec.end());

        // output
        vector<int>::iterator it;
        for ( it = ivec.begin();
               it != ivec.end(); ++it ) {
                cout << *it << " ";
        }
        cout << endl;

        return 0;
}
```

Iterators

- Provide a **general way for accessing** each element in sequential (vector, list) or associative (map, set) containers

Pointer Semantics

- Let `iter` be an iterator then :
  - `++iter` (or `iter++`)
    Advances the iterator to the next element
  - `*iter` returns element addressed by the iterator

Begin and End

- Each container provide a **`begin()`** and **`end()`** member functions
  - **`begin()`** returns an iterator that addresses the **first** element of the container
  - **`end()`** returns an iterator that addresses **1 past the last** element

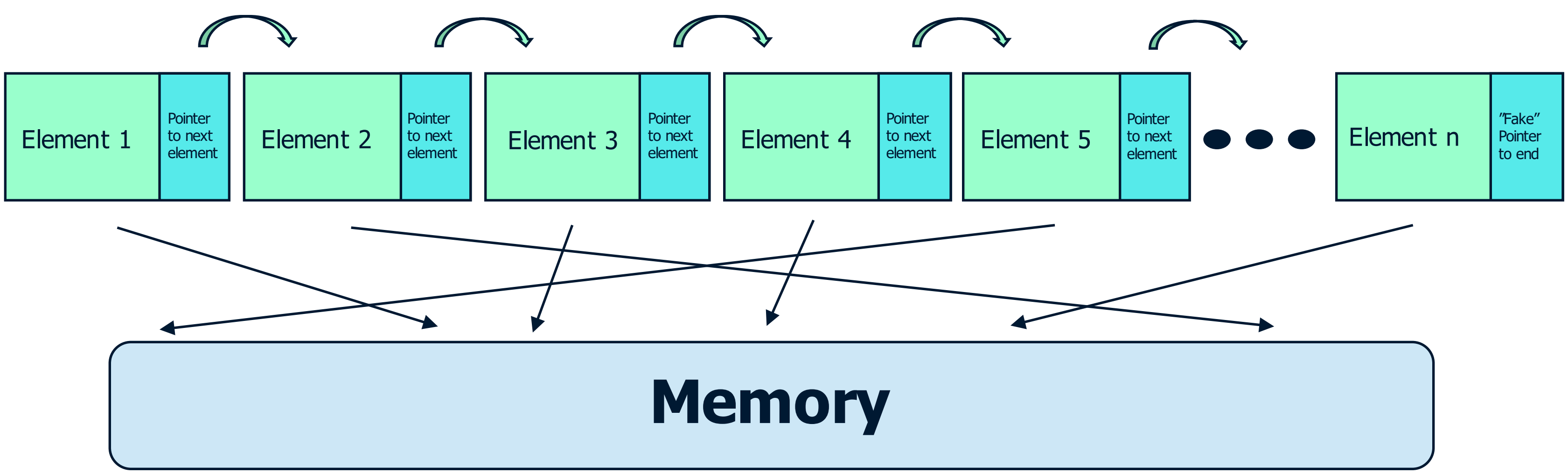
Iterating Over Containers

- Iterating over the elements of any container type

```
for ( iter = container.begin();
      iter != container.end();
      ++iter )
{
      // do something with the element
}
```

std::list<T>

- Linked list
- Arbitrary location of elements in memory
  - Expensive to access n^{th} element (have to iterate)
  - Easy to add/remove elements!



Example

```
int main() {
        int input;
        list<int> ilist;

        // input
        while (cin >> input )
                ilist.push_back(input);

        // sorting
        ilist.sort();

        // output
        list<int>::iterator it;
        for ( it = ilist.begin();
               it != ilist.end(); ++it ) {
                cout << *it << " ";
        }
        cout << endl;

        return 0;
}
```

STL - Include files

```
#include <iostream>   // I/O
#include <list>       // container
#include <algorithm> // sorting

using namespace std;
```

vector vs list

Where are the bottlenecks?

- Sorting: needs easy access to nth element → use **`vector`**!
- Storing: number of elements unknown → use **`list`**!

```cpp
int main() {
    int input;
    vector<int> ivec;

    // input
    while (cin >> input )
        ivec.push_back(input);

    // sorting
    sort(ivec.begin(), ivec.end());

    // output
    vector<int>::iterator it;
    for ( it = ivec.begin();
          it != ivec.end(); ++it ) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

```cpp
int main() {
    int input;
    list<int> ilist;

    // input
    while (cin >> input )
        ilist.push_back(input);

    // sorting
    ilist.sort();

    // output
    list<int>::iterator it;
    for ( it = ilist.begin();
          it != ilist.end(); ++it ) {
        cout << *it << " ";
    }
    cout << endl;

    return 0;
}
```

vectors and unknown size

- What is the problem?
  - Needs to be continuous memory block!
  - If pushing back 1 element, and space is insufficient
    - → Entire vector is copied to different place
- How to prevent?
  - Reserve sufficient size upfront!

```
// input
ivec.reserve(100);
while (cin >> input )
  ivec.push_back(input);
```

STL - Map

- **Solution:** Map – Associative Array
- We provide a key/value pair. The key serves as an index into the map, the value serves as the data to be stored
- Insertion/find operation:
    - O(log n)

Using Map

- Have a map, where the key will be the employee name and the value the employee object.

```
name   → employee
string → Employee

map<string, Employee *> employees;
```

Map Iterators

```
map<key, value>::iterator iter;
```

- What type of element iter does addresses?
  - The key ?
  - The value ?
- It addresses a key/value **pair**

STL - Pair

- Stores a pair of objects, first of type T1, and second of type T2

```
template<class T1, class T2>
struct pair<T1, T2>
{
      T1 first;
      T2 second;
};
```

Example: ordering employees

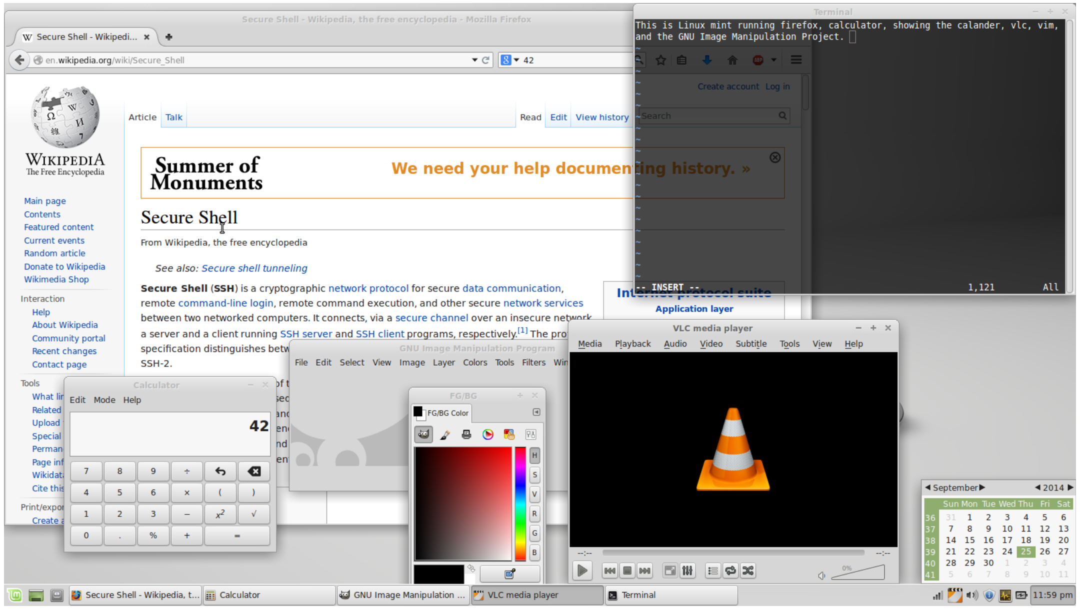
```
bool lessThen( pair<…> &p1, pair<…> &p2 ) { … }

int main() {
    map<string, Employee *> employees;
    /* Populate the map. */

    vector< pair<string, Employee *> > employeeVec;
    copy( employees.begin(), employees.end(),
          back_inserter( employeeVec ) );

    sort( employeeVec.begin(), employeeVec.end(),
          lessThen );

    vector< pair<string, Employee *> >::iterator it;
    for ( it = …; it != employeeVec.end(); ++it ) {
        cout << (it->second)->getName() << " "
             << (it->second)->getSalary() << endl;
    }
    return 0;
}
```

Ordering function

```
bool
lessThen(pair<string, Employee *> &l,
         pair<string, Employee *> &r )
{
      return (l.second)->Salary() <
             (r.second)->Salary()
}
```

Multi-tasking, Concurrency, and Parallel Computing

- What is multi-tasking?
  - Multi-tasking is one of the main functionalities supported by an operating system. It allows the **<u>quasi-parallel</u>** execution of multiple **<u>processes</u>**



Source: Wikipedia

Multi-tasking, Concurrency, and Parallel Computing

- What is concurrency?
  - More general
  - Execution of several <u>**computations**</u> at overlapping times
  - Concurrency can happen at the level of:
    - Network (cloud computing)
    - Computer / OS (multi-tasking, multiple <u>**processes**</u>)
    - Program (multiple <u>**threads**</u>)

Multi-tasking, Concurrency, and Parallel Computing

- What is parallel computing?
  - Strictly parallel execution of (possibly same) computations
  - Requires parallel computing hardware
    - Multi-core processor
    - Graphics Processing Unit (GPU)
    - Field Programmable Gate Array (FPGA)
    - Derived ASICs
    - Specialized software-programmable SoCs (Ambarella etc.)

A process

- A process is …
  - … started at the operating system level
  - … assigned a space of individual memory that is typically not shared with other processes
  - … communicating with other processes via other interfaces (network, disk space, etc.)

A thread

- A thread is …
  - … started at the process/program level
  - … granted access to the memory space that has been allocated to the process
  - … possibly sharing that memory space with other threads from the same process
  - … able to communicate directly with other threads through the assigned memory
  - … a means to realize parallel computing

Thread joining

- Main thread waits for other threads to finish!

```cpp
#include <thread>
#include <iostream>

void threadFunction() {
        std::cout << "Hello from thread 1\n";
}

int main() {
        std::thread th(threadFunction);
        std::cout << "Hello from thread 0\n";
        th.join();
}
```

Thread joining

What are critical sections?

- Data is usually shared between threads
- Problem:
  - Multiple threads access the same object at the same time
  - If operation is atomic (i.e. not divisible)
    - No other thread could read/operate on a partial result
    - It is safe!
  - If operation is not atomic (i.e. divisible into several steps)
    - Other threads could read/operate on partial result if switching happens in between
    - It is not safe!

What are critical sections?

- Critical section
  - A piece of code that accesses/modifies a shared resource, and the access/modification is non-atomic
  - → Access must not be concurrent!
  - → Simultaneous access by multiple threads must be prevented
  - → Access needs to be **mutually exclusive**!

Mutex

- How does it work?
  - Create a mutex by creating an instance of **`std::mutex`**
  - Lock it with a call to the member function **`lock()`**
  - Unlock it with a call to the member function **`unlock()`**

```
class Counter {
public:
  Counter() { m_value = 0; };

  int getValue() { return m_value; };
  void increment() {
    m_mutex.lock();
    ++m_value;
    m_mutex.unlock();
  };

private:
  int m_value;
  std::mutex m_mutex;
};
```

Output:
25000 every time!

Mutex

- How does it work?
  - A mutex does not directly lock a part of the code
  - A mutex is a resource (i.e. a lock), and we use it passively to protect a critical section
  - **`lock()`** is blocking until it “has the lock”
  - **`unlock()`** ”releases the lock”
  - Only one at a time can have the lock
  - → Bound all critical sections (w.r.t. to the same data) by the same mutex

std::atomic

- C++11 introduces atomic types as a generic template class that can be wrapped around any type

```
std::atomic<Type> object;
```

- Can be used with any type
- Makes the operations on that type atomic
- Locking technique depends on type, and can be very fast for small objects (faster than mutex!)

Example atomic

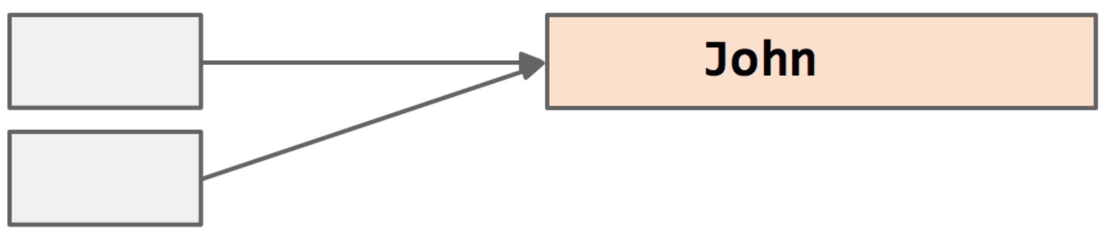
- Back to our counter example

```
class Counter {
public:
  Counter() { m_value = 0; };

  int getValue() { return m_value; };
  void increment() {
    m_mutex.lock();
    ++m_value;
    m_mutex.unlock();
  };

private:
  int m_value;
  std::mutex m_mutex;
};
```

std::shared_ptr

Shared pointers:

Provide shared ownership
Many pointers may own the same object
The last one to survive is responsible of its disposal through the deleter



std::shared_ptr

Shared pointers have a garbage collection mechanism based on a reference counter contained in a control block

Each new shared owner copies the pointer to the control block and increases the count by 1