Object Oriented Programming in C++

RMIT UNIVERSITY

Acknowledgements

- Lecture notes are based on material created by Andrew Ryan and Geoff Leech with edits by John Thangarajah, Xiangmin (Emily) Zhou and Paul Miller.

Course Details

- Lecturer: *Xiangmin (Emily) Zhou*
  - Office: 14.10.03
  - Email: xiangmin.zhou@rmit.edu.au
  - Consultation times: see via Blackboard
- Head Tutor/ Tutor/Lab Assistant: *Paul Miller*
  - Office: 14.09.13
  - Email: paul.miller@rmit.edu.au
  - Consultation times: see via Blackboard
- Tutor/Lab Assistant: Luke Gallagher:
  - Email:luke.Gallagher@rmit.edu.au
- Tutor: Irvan Arief Ang, Email: irvan.ariefang@rmit.edu.au
- Lab Assistant: Xiaolu Lu, Email: xiaolu.lu@rmit.edu.au

Course Details

- Blackboard
  - Here you will find:
    - Lecture Notes
    - Tutorials and Laboratories
    - Assignments (also for submission!)
    - Sample Source Code
    - Discussion forum: send your questions/comments/concerns about the course there!

Prerequisites

- This course requires you to have completed, and have an understanding of
  - Programming Principles 1B (Java) or Java for C Programmers
  - Programming Principles 2A (C language)
- A lot of knowledge is assumed, so be sure you know this.

Prerequisites

- STOP!
- It is expected that you will know
  - Basic object oriented concepts
  - Fundamentals of the C language
  - Fundamentals of the Java language
- If you are unsure of these, perhaps reconsider doing this course in another semester.
- Feel free to e-mail me to discuss your options.

Objectives of this course

- Develop object-oriented applications using the full C++ language, with
  - *Efficiency*
  - *Reliability* and *Maintainability*
- Apply the fundamental principles of object-oriented programming
  - *Encapsulation* for reliability and maintainability
  - *Inheritance* for extensibility and eliminating code duplication
  - *Polymorphism* for adaptability
- Learn how, when, and why to choose among different C++ constructs and coding techniques

Objectives of this course

- Understand the C++ runtime environment
  - And C++'s relation to other programming languages
- Use the C++ Standard Library components
  - Template containers and exceptions
  - Strings and utility classes
  - Standard IO

Assessment

- Assessment for this course is by assignments and by an examination
  - Assignment 1 (W6) is worth 20 marks
  - Assignment 2 (W12) is worth 30 marks
  - Examination is worth 50 marks
  - No hurdle (50 out of 100)

Tutorials

- Tutorials are available from the course webpage.
- Tutorials are 1 hour in length.
- Please allocate yourself to a tutorial and attend that tutorial only.
- Visit http://inside.cs.rmit.edu.au/tutorial_allocation/ to do so

Plagiarism

- Submitting an assignment containing other people's work.
- Helping other students to plagiarise.

All submitted work must be your own.

The only exception is: other people's work can be included if the assignment has explicit instructions to do so. All copied work (from the internet, other students, or staff) must be fully referenced.

A student submitting copied work will receive **no marks** for that assignment. Partial marks will **not** be given, **even if only part** of the assignment was copied. If this means that a hurdle is not reached, then the student fails the subject.

A student who plagiarises a second time will be sent to the disciplinary committee. Penalties include failure, fines and expulsion from the university.

For more information, see "Plagiarism" on http://www.rmit.edu.au/browse;ID=p1l82w9nky8a

Guide to lecture series

- Lecture 1
  - Administration, basics of the language, input and output to screen, data types, declarations and definitions, arrays, pointers and references, flow of control.
  - C++ compilation and program structure.
- Lecture 2
  - Functions, call-by-value, address and reference, Classes and objects, Object oriented concepts, visibility, Member functions, constructors, destructors
- Lecture 3
  - Introduction to C++14
- Lecture 4
  - Standard Template Library, data structures and algorithms

Guide to lecture series

- Lecture 5
  - Optimisation, profiling, efficiency, in-lining, memory pools
- Lecture 6
  - Inheritance, polymorphism, is-a versus has-a, object hierarchies, abstract base classes, virtual functions and pure virtual functions
- Lecture 7
  - Strings and streams, file I/O, I/O manipulators, string streams

Guide to lecture series

- Lecture 8
  - Multiple inheritance, virtual base classes, protocol classes
  - Operator overloading, uses, abuses, canonical forms of operators, declaring as member functions vs declaring as free functions. Friend functions.
- Lecture 9
  - Template classes, template functions, generic programming principles, member template functions.

Guide to lecture series

- Lecture 10
  - Exceptions, exception safety, exception handling, try/catch blocks, throw lists, implications of thrown exceptions.
- Lecture 11
  - More on C++14 and Good Software Design / software development practices
- Lecture 12
  - Where to from here? Future directions, what more can you learn?

Any questions?

- Feel free to ask questions during lectures, during breaks, on Blackboard.
- Post your questions on blackboard rather than emailing, so that all of you can share your QAs.

Bit of history - C, C++, and Other OOPLs

- C was designed in the early 1970s
  - To implement the UNIX operating system
- The design goals of C included
  - Efficiency - easy to compile to efficient machine code
  - Hardware access and flexibility - vital for implementing operating systems
  - Portability - of the language
  - Conciseness
- The design goals of C++ *<u>did not</u>* include
  - Reliability - compile and run time checks

C, C++, and Other OOPLs (cont.)

- Object Oriented Programming (OOP) predates C - Simula 67
  - But was a boutique academic area
  - Focus in the 70’s was on “structured programming”
- In the 70’s object-oriented and object-based languages gradually became more mainstream
  - Smalltalk and Ada
- Classes were added to a C dialect in the early 80’s
  - By Bjarne Stoustrup, of Bell Laboratories
- By the early 90’s, C++ was the emerging leading language for OOP
- C++’s goals included
  - Downward compatibility with C - reuse/mixing of C/C++
  - Efficiency - no loss of performance over C
  - Flexibility - to meet different programming styles

C, C++, and Other OOPLs (cont.)

- Java was a new OOP, in the mid 90's, addressing C++'s drawbacks
  - Focus was on reliability and simplicity
  - Portability
    - Below the source level through an intermediate language - Java Byte codes and standard-sized data types
  - Complexity and additional functionality moved into libraries
    - For graphics, tasking, memory management, as appeared later with Java/J2EE
- Java's main drawback over C/C++ was performance
  - But the J2EE library was a major productivity gain

C, C++, and Other OOPLs (cont.)

- Java had its limitations
  - Performance - x5 times slower than C/C++ even with Just In Time (JIT) compilers
  - Portability an issue
    - Assumes a Java Virtual Machine (JVM) for runtime support
    - JVMs are subtly different for each computer platform
  - No standardization
    - C++ is an international (ISO) standard
    - SUN still holds the Java standard/certification

C, C++, and Other OOPLs (cont.)

- Early in 2000 Microsoft released .NET
- .NET solved many of the problems associated with Windows software development
  - Many incompatible libraries (e.g., SDK, MFC, ATL, COM, DCOM, ActiveX) and languages (C/C++, Visual Basic)
    - Solution: a portable intermediate language - MSIL standardized through ECMA
    - All .NET languages compile into MSIL and can be mixed and matched - a Visual Basic class can inherit from a C# class
  - “DLL hell” - dynamic linking is unreliable and insecure
    - Solution: assemblies in MSIL, with builtin meta data, versioning, and security
- .NET uses a “Just In Time” (JIT) compilers
  - To overcome performance problems of intermediate languages

C, C++, and Other OOPLs (cont.)

- There is no one best OOPL, each has their strengths and application areas
  - C++ - efficient, but few standard libraries
  - .NET - limited, at present, to Windows
  - Java - portable, but performance limited

Modern software development focuses on libraries and assembling programs from reusable components rather than "coding from scratch"

Let’s start!--Hello World

```
// hello world program: hello_world.cpp
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "Hello, World!" << endl;
    return 0;
}
```

- Let’s take a look at each section in turn.

Hello World

```
// hello world program
#include <iostream>
```

- The first part is a comment. In C++, we can use `/*...*/` and `//` comments in the same way as we would from Java.
- `//` means anything that follows on the same line is a comment.
- You can use `/*...*/` comments to make a comment span multiple lines

Hello World

- The second part of the code example is a `#include` directive. These take the contents of another file, and automatically include them at this point.
- These files are usually called *header files*.
- The `iostream` header file is provided with the Standard C++ Library, and gives us declarations that we can use to output data to screen, and get data from the user.
- Other header files are available for your use.

Hello World

```
int main(int argc, char* argv[])
```

- This is the main function. All code begins executing in the main function.
  - Note that, unlike Java, the main function returns an `int`.
  - Moreover, the main() method is NOT inside the class; rather, it is a *global function*.
  - It also has different arguments (Java has a `String[]` as its argument list). In this code example, we don't use the arguments, although they function exactly like they would in C.

Hello World

- The actual body of code is:

```
{
    std::cout << "Hello, World!" << endl;
    return 0;
}
```

- `std::cout` (see-out) is our console-output – the console is the screen as we see it. We use `cout` to display messages to the user. `cout` exists in the Standard C++ Library (`std`).

Hello World

- The '<<' is called the stream insertion operator.
- This operator takes the `"Hello, World!"` string that we provided, and inserts it into the `cout` stream. This is then displayed on the screen.
  - Operators can be *overloaded* (to be described in the future…!)
- The final part of the line, `endl`, is a directive to `cout` to end the line, and start a new line.
- `endl` also exists in the Standard C++ Library.

Hello World

- All objects in the Standard C++ Library exist in namespace `std`,
- We need to either provide a global using directive, or individually specify each object we intend to use, like `using std::cout;`
- If we don't specify a using directive, we can manually specify the namespace whenever we use an object.

```
std::cout << "Hello, World!" << std::endl;
```

Hello World

- The final part of the program is the return code:

```
    return 0;
}
```

- In C++, a successful program returns 0, and a program that failed for some reason a non-zero value.

Compiling our program

- To compile our programs, we use g++.

```
g++ -o helloworld hello.cpp
```

- Assumes file containing our program is hello.cpp.
- Note that `g++` syntax is the same as `gcc` syntax.
- `g++` is installed on most Unix/Linux systems. Alternatively, use cygwin under Windows to get a Unix-like shell.

Running the program

- To run the program, type the following:

```
./helloworld
```

- We can change the executable name by altering the value after the `–o` option.
- If we compile with `–g`, we can also use `gdb` to run:

```
gdb ./helloworld
```

Data types--Basic data types

- C++ provides the same data types as C, plus a few more.

`char`	Character
`Short`	Short integer
`int`	integer
`long`	long integer
`float`	Single-precision floating point
`double`	Double-precision floating point
`bool`	Boolean (true/false)
`#include <string>` `std::string`	String class.

Data type uses

- `char`s, `int`s, `long`s, `float`s, `double`s function the same way as they do in Java, except:
  - `char`s are only 1 byte long.
  - There are no standard sizes of most data types (other than `char`). The C rules apply here:
    - `short <= int <= long`
- The string class is part of the Standard C++ library, and isn't a POD-type (plain ol' data-type).

Example 2: Adding two `int`s

```
//Addingtwoints.cpp

#include <iostream>

int main() {
  std::cout << "Enter two numbers: ";
  int i, j;
  std::cin >> i >> j;
  std::cout << "\nThe sum is: " << i+j << "\n”;
  return 0;
}
```

- Note how operators are different between `cout` and `cin`.

Declaration or definition?

```
int a;
```

- This is a declaration. “I **declare** that `a` is an `int`”. We do not provide `a` with a value.

```
a = 2;
```

- This is a definition. “I **define** `a` to be 2”.

```
int a = 2;
```

- We can combine the two. This is both a declaration and a definition.

Arrays and pointers

- C++ supports C operations with pointers, e.g.

```
int a;

int* pA = &a;
```

- Likewise, we can declare and define arrays of data types:

```
int array[100];
```

Const datatypes

- C++ has eliminated `#define`'s from C, using `const` instead:

```
const int MAX_SIZE=100;

int array[MAX_SIZE];
```

- Const data is exactly like `final` data types in Java.

```
final int MAX_SIZE=100;  // same as C++ const
```

Example 3: What's the meaning of life?

```
#include <iostream>  //namespace.cpp

using namespace std;

int main() {
   const int answer = 42;
   cout << answer << " is the answer" << endl;
}
```

- `const` is used as Java `final`.
- You can associate "left-to-right" using `<<`.

Example 3 (cont): Namespaces

- The next part of the first example we'll look at is:

```
using namespace std;
```

- Namespaces are similar in idea to Java's packages. In fact, saying `using namespace std` is the same as saying `import java.lang.*` in Java.
- These are called **using directives**.

Namespaces (cont.)

```
#include <iostream>    //namespace_input.cpp

int main() {
  using std::cin;
  std::cout << "Enter two numbers: ";
  int i, j;
  cin >> i >> j;
  using namespace std;
  cout << "\nTheir sum is: " << i+j << "\n”;
  return 0;
}
```

- Note that `using` makes `cin` visible in this scope.
- Same for `using namespace`, makes `cout` visible in this scope.

Flow control

- C++ has the same flow control structures as C and Java:
  - `if` statements,
  - `do..while` loops,
  - `while` loops,
  - `for` loops, and
  - the conditional operator.

If statements

```
if (a < b) {
   //
}
else if (a < c) {
 //
}
else {
 //
}
```

..should be a familiar flow control structure!

While loop

```
while (a < b) {
 //

}
```

- Continue looping while condition is true.
- No guarantee the `while` loop will be executed. (The condition is tested **before** the loop executes).

Do…while loop

```
do {
    //
} while (a < b);
```

- Same as a `while` loop, but the condition is tested **after** the loop is executed.
- Guaranteed to execute at least once.

For loop

- The workhorse loop!

```
for (int i=0;i<MAX_SIZE;++i) {
    //
}
```

- Initialise counter to some value (`int i=0`), test a condition **before** the loop, and perform an action (`++i`) after the loop.

Conditional operator

- Great for short (and probably unreadable) code.

```
(a < b) ? 1 : 0;
```

- Is the same as:

```
if (a < b) {
    return 1;
}
else {
    return 0;
}
```

References

- References in C++ are different to references in Java.
- In C++ a reference is an alias for another object.

```
int a = 4;

int& refA = a; // refA is a ref to a

cout << refA; // outputs "4"

refA = 5; // refA is alias for a, so a is changed

cout << a; // outputs "5"
```

The C/C++ Preprocessor

- C and C++ use the same preprocessor
  - A preprocessor is run before compilation takes place
- Common preprocessor directives are: `#include`, `#define`, and `#ifdef`

```
#ifndef H_UTILITIES
#define H_UTILITIES
// A file of useful macros
#define PI 3.14159265
#define MAX(X, Y) (X > Y ? X : Y)

...

#endif
```

This idiom prevents this header file (in `utilities.h`) being included twice

An "inline function"

```
#include <iostream>  //utilities.cpp
#include "utilities.h"

int main() {
  std::cout << "Enter two numbers: ";
  int i, j;
  std::cin >> i >> j;
  std::cout << "\nTheir max is: " << MAX(i, j);
  return 0;
}
```

Read in the contents of this local file

The C/C++ Preprocessor (cont.)

- `#include <xxx>` indicates a library header file
  - The `.h` file name extension is not used now in standard C++
- `#include "yyy.h"` indicates a local (project specific) header file
- `#ifdef` is used for “conditional compilation”
  - Compile flags can control what is compiled into an application
  - However, complex conditional compilation can be very hard to maintain
- `#define` is now largely obsolete as inline functions can be nightmares to maintain (consider `MAX(++i, j)`)
  - For simple constants, such as `PI` above, use a C++ const definition
  - For inline functions, such as `MAX` above, use the `inline` function modifier

Storage Allocation

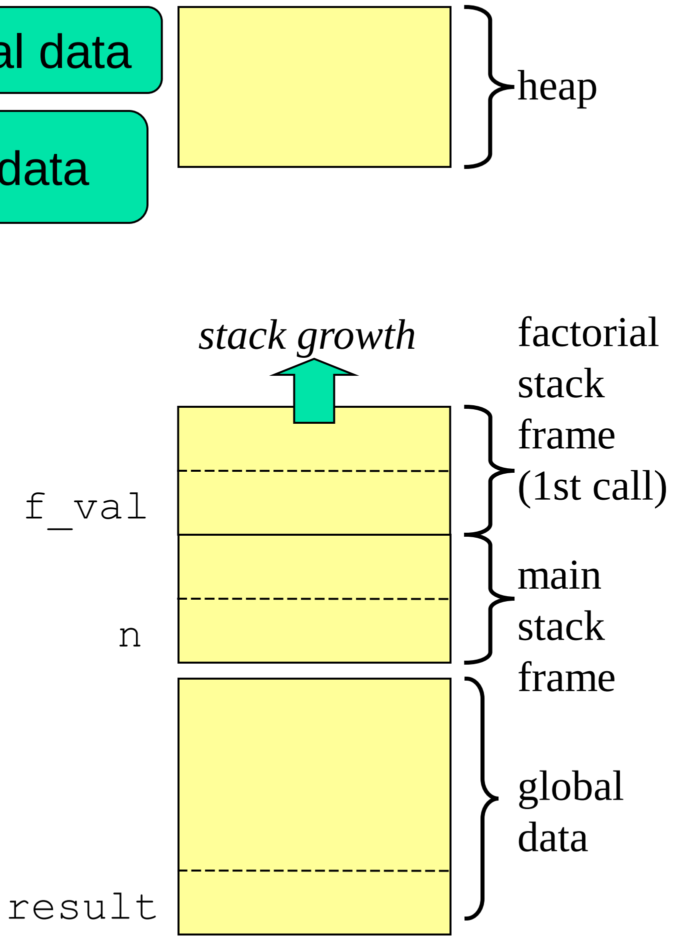
- For efficiency, C/C++ were designed around the memory model and instruction sets of computers
  - *Memory* is sequentially addressed
  - Machine instructions reference memory addresses either with
    - A *fixed* (constant) memory address
    - An *offset* address - the sum of a constant and the contents of a register
- C/C++ allocates storage in 3 areas:
  - *Global* - fixed addresses
  - *Stack* - simple offset addresses from a stack pointer register
  - *Heap* - complex offset addresses

Storage Allocation (cont.)

```
#include <iostream> //storage.cpp

int result;
int factorial(int n) {
  int f_val = n;
  if (f_val == 1) return f_val;
  else return f_val*factorial(n-1);
}

int main() {
   std::cout << "Enter a number: ";
   int n;
   std::cin >> n;
   std::cout << "\nThe factorial is: "
          << factorial(n);
   return 0;
}
```

global data

local data



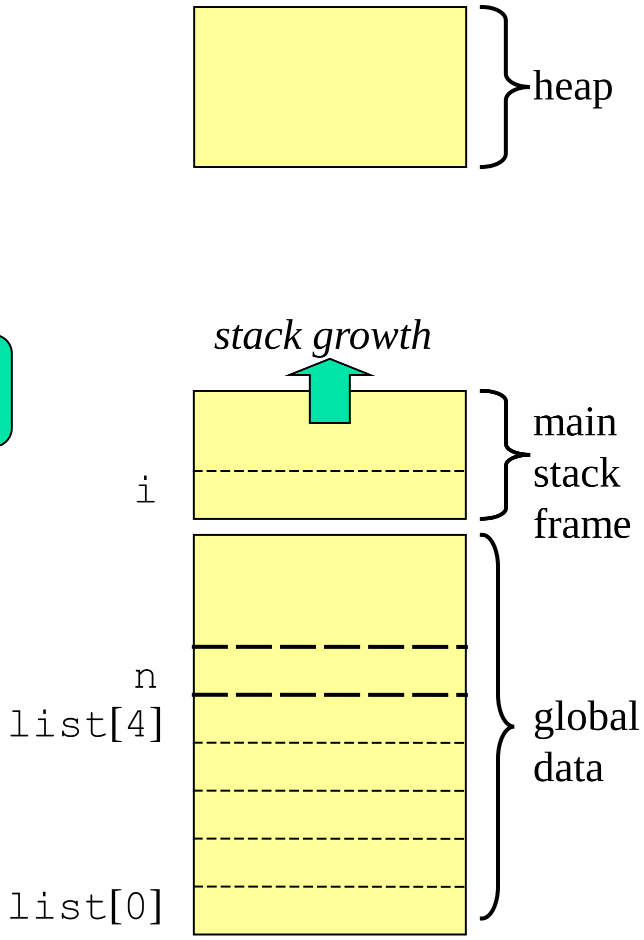
- What happens on input 3?
- What happens on input -3?

Arrays

```
#include <iostream> //arrays.cpp

const int MAX = 5;
int list[MAX];
int sum = 0;

int main() {
  std::cout << "Enter " << MAX
  <<
    " numbers: ";
  for (int i=0; i<MAX; ++i) {
    std::cin >> list[i];
    sum += list[i];
  }
  std::cout << "\nTheir sum is:
  " << sum;
  return 0;
}
```

array of 5 ints

i local to for loop



- C/C++ arrays have a fixed size
  - Declared as type `var_name[const_size]`
  - Indexed from `0` through `const_size-1`

Arrays (cont.)

- `list` points to the first element of the array
  - `list` is a **`const`** pointer to an **`int`**
  - C/C++ define array accessing as pointer arithmetic
  - `list[i]` means the same as `*(list + i)` - "add `i` to the address `list`, then dereference it (get the contents)"
- Because array indexing is just a shorthand for pointer arithmetic, C/C++ programs often use pointers instead of array indexing
  - The two loops below generate the same machine instructions!

```
...
for (int i=0; i<MAX; ++i) {
  std::cin >> list[i];
  sum += list[i];
}
...
```

`ip` is an int pointer: `++` increments `ip`

`*ip` gets the contents of what `ip` points to

```
...                //array_cont.cpp
for (int* ip=list; ip<list+MAX; ++ip) {
  std::cin >> *ip;
  sum += *ip;
}
...
```

`list+MAX == &list[MAX]`

Arrays (cont.)

- C/C++ allow a program to take the address of, or dereference, any object.
  - `&name` means "get the address of `name`"
  - `*ptr` means "get the contents of what `ptr` points to (dereference `ptr`)"

```
...
int i;
int main() {
  int* ip=&i;
  *ip = 42;
  std::cout << i;
  ...;
}
```

global variables are initialized to zero by default; local variables are initially undefined

`ip` now points to `i`

or `std::cout << *ip`

Arrays (cont.)

- There is no bounds checking on C/C++ arrays!
  - Accessing `list[-1]` or `list[MAX]` leads to “undefined” run time behaviour
  - Since storage is generally allocated sequentially, what might assigning to `list[MAX]` do?
- C/C++ arrays are unsafe and should generally be avoided. Use standard library containers instead
- C++ provides standard library containers that
  - Grow automatically as elements are added
  - Throw exceptions on access outside the container bounds
  - Check at compile time that the elements are the right type
    - Unlike Java, C++ does not provide “heterogeneous object containers”

Containers

```
#include <iostream> //containers.cpp
#include <vector>
std::vector<int> int_input;

int main() {
  std::cout << "Enter numbers, terminate with a non-number\n";
  int num;
  int sum = 0;
  while (std::cin >> num) {
    int_input.push_back(num)
  }
  for (int i=0; i<int_input.size(); ++i) {
    sum += int_input[i];
  }
  std::cout << "\nTheir sum is: " << sum;
  return 0;
}
```

`int_input` is a *specialization* of the generic library **`template`** class `list`

`>>` is false on invalid input

`push_back` and `size` are standard container methods

Containers (cont.)

- The standard template class containers include:
- Sequences: `list`, `vector`, `deque`
  - Access to the `front` (except `vector`) and `back` via `push` and `pop`
  - Random access array indexing: **`operator`**[ ] (except for `list`)
- Associative containers: `map`, `multimap`, `set`, `multiset`
- The preferred way to traverse a container in C++ is using an *iterator*
- Similar to Java's iterator and enumeration interfaces

Containers (cont.)

`typedef` in C/C++ creates an alias for a type name

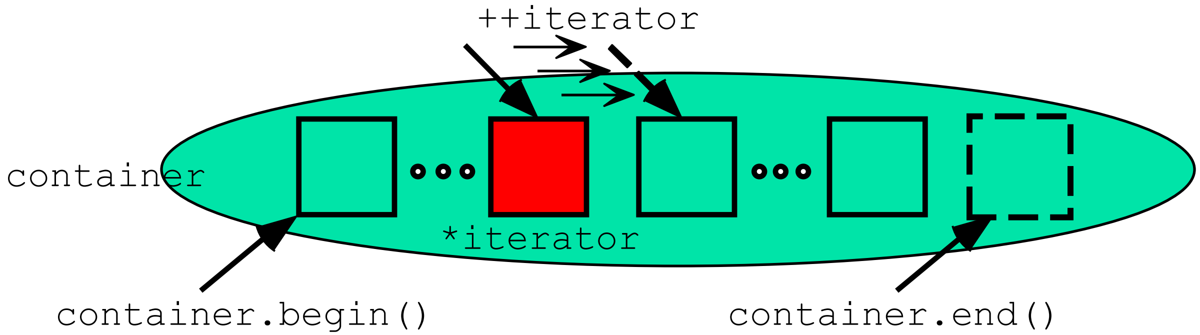
```
...                           //continers_cont.cpp
typedef std::vector<int> int_container;
int_container int_input;

int main() {
  ... // as above
  for (int_container::iterator it=int_input.begin();
       it!=int_input.end(); ++it) {
    sum += *it;
  }
  std::cout << "\nTheir sum is: " << sum;
  return 0;
}
```

Containers (cont.)

- Traversing, or *iterating*, over a container is common
  - For `vector`s, we can do this using subscripting, `[]`
  - Most containers, such as `set`, and `map`, do not support this
- An *iterator* is an object that encapsulates the state and behavior necessary to iterate over a container
- An *iterator* requires just three simple operations
  - *increment* (`operator++`) Move the iterator forward to the next object
  - *dereference* (`operator*`) Fetch the current object the iterator points to
  - *comparison* (`operator== !=`) Compare iterators

Containers (cont.)

- Container have `begin()` and `end()` functions that return iterators for use in comparisons.



Strings

- C strings are just (fixed size) arrays of `char`acters
  - Pointer arithmetic can be used with C strings as with any other array
  - C provides a very unsafe library for C string manipulation
  - However, C strings should <u>not</u> be used in C++ applications, except for literal constants

Strings (cont.)

```
//strings.cpp
#include <string.h>

int main() {
  const char* name1 = "bat";
  char name2[9] = "fruit";
  strcat(name2, name1);
  char* name3;
  strcpy(name3, name2);
  ...
}
```

`"bat"` is a *string literal* with a terminating `'\0'` thus it occupies 4 bytes of storage, not 3

concatenate `"bat"` onto `"fruit"`, requires 10 bytes, so overwrite other stack storage

`name3` is unititialized, could point anywhere thus `strcpy` copies into a random location

Strings (cont.)

- C++ provides `std::string`
  - Dynamically sized strings, with a wide range of functions and operators
  - Similar to Java's `StringBuffer`

```
#include <string>//strings_cont.cpp
#include <iostream>
int main() {
  std::string name1 = "bat";
  std::string name2 = "fruit";
  name2 += name1; // or name2.append(name1)
  std::string name3;
  name3 = name1;
  std::cout << name3;
  ...
}
```

`std::string`'s can be constructed from `char*`'s and converted to `char*`'s

`name3` is a copy of `name1`, C++ assignment defaults to copying values not references

output operator overloaded for `std::string` as well as **`char*`**

  - More to be discussed later…