Algorithms and Analysis

Lesson 4: *C++ 101*

C with classes, new, overloading, templates

Outline

1. **C with Classes**
2. New
3. Overloading
4. Templates

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient, but not safe

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient, but not safe or easy to use

Keeping Things Together

- As soon as you start programming bigger systems you want to keep information together
- C facilitated this through C structures `struct`

```
struct MyStructure {    // Structure declaration
  int myNum;            // Member (int variable)
  char myLetter;        // Member (char variable)
}; // End the structure with a semicolon

int main() {
  struct MyStructure s1;

  s1.myNum = 13;
  s1.myLetter = 'B';

  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);
  return 0;
}
```

Keeping Things Together

- As soon as you start programming bigger systems you want to keep information together

- C facilitated this through C structures `struct`

```
struct MyStructure {    // Structure declaration
  int myNum;            // Member (int variable)
  char myLetter;        // Member (char variable)
}; // End the structure with a semicolon

int main() {
  struct MyStructure s1;

  s1.myNum = 13;
  s1.myLetter = 'B';

  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);
  return 0;
}
```

Keeping Things Together

- As soon as you start programming bigger systems you want to keep information together

- C facilitated this through C structures `struct`

```
struct MyStructure {    // Structure declaration
  int myNum;            // Member (int variable)
  char myLetter;        // Member (char variable)
}; // End the structure with a semicolon

int main() {
  struct MyStructure s1;

  s1.myNum = 13;
  s1.myLetter = 'B';

  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);
  return 0;
}
```

Estimated Errors in the Mean

- When working with empirical data, $\{X_i, i = 1, 2, \ldots, n\}$, we want to compute the *mean* and *variance* (from which we can estimate the error in the mean)

- We can do this on the fly by storing

$$n, \qquad \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad Q_n = \sum_{i=1}^{n}(X_i - \hat{\mu}_n)^2$$

- Given X_{n+1} we can update our data using: $\Delta = \frac{X_{n+1} - \hat{\mu}_n}{n+1}$

$$\hat{\mu}_{n+1} = \hat{\mu}_n + \Delta \qquad Q_{n+1} = Q_n + n\,\Delta\,(X_{n+1} - \hat{\mu}_n)$$

Estimated Errors in the Mean

- When working with empirical data, $\{X_i, i = 1, 2, \ldots, n\}$, we want to compute the $mean$ and $variance$ (from which we can estimate the error in the mean)

- We can do this on the fly by storing

$$n, \qquad \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad Q_n = \sum_{i=1}^{n}(X_i - \hat{\mu}_n)^2$$

- Given X_{n+1} we can update our data using: $\Delta = \frac{X_{n+1}-\hat{\mu}_n}{n+1}$

$$\hat{\mu}_{n+1} = \hat{\mu}_n + \Delta \qquad Q_{n+1} = Q_n + n\,\Delta\,(X_{n+1} - \hat{\mu}_n)$$

Estimated Errors in the Mean

- When working with empirical data, $\{X_i, i = 1, 2, \ldots, n\}$, we want to compute the *mean* and *variance* (from which we can estimate the error in the mean)

- We can do this on the fly by storing

$$n, \qquad \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad Q_n = \sum_{i=1}^{n}(X_i - \hat{\mu}_n)^2$$

- Given X_{n+1} we can update our data using: $\Delta = \frac{X_{n+1} - \hat{\mu}_n}{n+1}$

$$\hat{\mu}_{n+1} = \hat{\mu}_n + \Delta \qquad Q_{n+1} = Q_n + n\,\Delta\,(X_{n+1} - \hat{\mu}_n)$$

Estimated Errors in the Mean

- When working with empirical data, $\{X_i, i = 1, 2, \ldots, n\}$, we want to compute the *mean* and *variance* (from which we can estimate the error in the mean)

- We can do this on the fly by storing

$$n, \qquad \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad Q_n = \sum_{i=1}^{n}(X_i - \hat{\mu}_n)^2$$

- Given X_{n+1} we can update our data using: $\Delta = \frac{X_{n+1} - \hat{\mu}_n}{n+1}$

$$\hat{\mu}_{n+1} = \hat{\mu}_n + \Delta \qquad Q_{n+1} = Q_n + n\,\Delta\,(X_{n+1} - \hat{\mu}_n)$$

this requires the back of an envelop to verify

Second Order Statistics in C

- In C we can use a `struct` to keep this data together

```
struct Sos {
  unsigned n;
  double mu;
  double Q;
};
```

- We can write functions that update the Sos structure

```
void add(struct Sos& sos, double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}
```

- **`struct`**s help keep information together, but can we do better?

Second Order Statistics in C

- In C we can use a `struct` to keep this data together

```
struct Sos {
  unsigned n;
  double mu;
  double Q;
};
```

- We can write functions that update the Sos structure

```
void add(struct Sos& sos, double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}
```

- **`struct`**s help keep information together, but can we do better?

Second Order Statistics in C

- In C we can use a `struct` to keep this data together

```
struct Sos {
  unsigned n;
  double mu;
  double Q;
};
```

- We can write functions that update the Sos structure

```
void add(struct Sos& sos, double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}
```

- **`struct`**s help keep information together, but can we do better?

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing, except C++ is a lot more elegant than python

Classes

- Classes are richer user defined datatype than structs
- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing, except C++ is a lot more elegant than python
- C++ has grown since 1985, adding templates and a lot of nice functionality

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                         // encapsulate
  int n;
  double mu;
  double Q;

public:                          // interface
  Sos();                         // constructor
  void add(double x);            // add data
  double mean();                 // return mean
  double var();                  // unbiased estimate of variance
  double error();                // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                    // encapsulate
  int n;
  double mu;
  double Q;

public:                     // interface
  Sos();                    // constructor
  void add(double x);       // add data
  double mean();            // return mean
  double var();             // unbiased estimate of variance
  double error();           // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                    // encapsulate
  int n;
  double mu;
  double Q;

public:                      // interface
  Sos();                     // constructor
  void add(double x);        // add data
  double mean();             // return mean
  double var();              // unbiased estimate of variance
  double error();            // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                    // encapsulate
  int n;
  double mu;
  double Q;

public:                      // interface
  Sos();                     // constructor
  void add(double x);        // add data
  double mean();             // return mean
  double var();              // unbiased estimate of variance
  double error();            // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                        // encapsulate
  int n;
  double mu;
  double Q;

public:                         // interface
  Sos();                        // constructor
  void add(double x);           // add data
  double mean();                // return mean
  double var();                 // unbiased estimate of variance
  double error();               // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                    // encapsulate
  int n;
  double mu;
  double Q;

public:                     // interface
  Sos();                    // constructor
  void add(double x);       // add data
  double mean();            // return mean
  double var();             // unbiased estimate of variance
  double error();           // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                      // encapsulate
  int n;
  double mu;
  double Q;

public:                        // interface
  Sos();                       // constructor
  void add(double x);          // add data
  double mean();               // return mean
  double var();                // unbiased estimate of variance
  double error();              // estimated error in mean
}

#endif
```

Classes by Example

- Define interface in header file `sos.h`

```
#ifndef __SOS_H__
#define __SOS_H__

class Sos {
private:                    // encapsulate
  int n;
  double mu;
  double Q;

public:                      // interface
  Sos();                     // constructor
  void add(double x);        // add data
  double mean();             // return mean
  double var();              // unbiased estimate of variance
  double error();            // estimated error in mean
}

#endif
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return Q/(n-1.0);
}

double error() const
{
  return sqrt(var()/n);
}
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return Q/(n-1.0);
}

double error() const
{
  return sqrt(var()/n);
}
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return Q/(n-1.0);
}

double error() const
{
  return sqrt(var()/n);
}
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return Q/(n-1.0);
}

double error() const
{
  return sqrt(var()/n);
}
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return Q/(n-1.0);
}

double error() const
{
  return sqrt(var()/n);
}
```

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Libraries

- C++ comes with a lot of in built libraries
- You include libraries using include statements

```
#include <iostream>
#include <vector>
```

- Same as C, but the C++ built in libraries don't have ".h"
- These are known as the standard library or the standard template library (STL)
- There is a naming convention, that libraries built into the system are called `<library>` while libraries you write are called `"library.h"`

Libraries

- C++ comes with a lot of in built libraries
- You include libraries using include statements

```
#include <iostream>
#include <vector>
```

- Same as C, but the C++ built in libraries don't have ".h"
- These are known as the standard library or the standard template library (STL)
- There is a naming convention, that libraries built into the system are called `<library>` while libraries you write are called `"library.h"`

Libraries

- C++ comes with a lot of in built libraries
- You include libraries using include statements

```
#include <iostream>
#include <vector>
```

- Same as C, but the C++ built in libraries don't have ".h"
- These are known as the standard library or the standard template library (STL)
- There is a naming convention, that libraries built into the system are called `<library>` while libraries you write are called `"library.h"`

Libraries

- C++ comes with a lot of in built libraries
- You include libraries using include statements

```
#include <iostream>
#include <vector>
```

- Same as C, but the C++ built in libraries don't have ".h"
- These are known as the standard library or the standard template library (STL)
- There is a naming convention, that libraries built into the system are called `<library>` while libraries you write are called `"library.h"`

Libraries

- C++ comes with a lot of in built libraries
- You include libraries using include statements

```
#include <iostream>
#include <vector>
```

- Same as C, but the C++ built in libraries don't have ".h"
- These are known as the standard library or the standard template library (STL)
- There is a naming convention, that libraries built into the system are called `<library>` while libraries you write are called `"library.h"`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Print

- Rather than pesky `printf` statements C++ allows us to use the operator <<

- When you get used to it, you will love if

```
#include <iostream>      // header file the defines library
using namespace std;

void main() {
  int i = 5;
  double x = 3.3;

  cout << "hello there" << i << ' ' << x << endl;
}
```

Print

- Rather than pesky `printf` statements C++ allows us to use the operator `<<`
- When you get used to it, you will love if

```
#include <iostream>      // header file the defines library
using namespace std;

void main() {
  int i = 5;
  double x = 3.3;

  cout << "hello there" << i << ' ' << x << endl;
}
```

Print

- Rather than pesky `printf` statements C++ allows us to use the operator <<

- When you get used to it, you will love if

```
#include <iostream>      // header file the defines library
using namespace std;

void main() {
  int i = 5;
  double x = 3.3;

  cout << "hello there" << i << ' ' << x << endl;
}
```

Outline

1. C with Classes
2. **New**
3. Overloading
4. Templates

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;      // creates an object a with value 5
int* b = &a;    // b is the memory address of object a
*b = 6          // *b is now a pseudonym for a
```

- b is called a pointer

- The *dereferencing* operator $\star$ turns the pointer back into the object

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;      // creates an object a with value 5
int* b = &a;    // b is the memory address of object a
*b = 6          // *b is now a pseudonym for a
```

- `b` is called a pointer

- The *dereferencing* operator $\star$ turns the pointer back into the object

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;     // creates an object a with value 5
int* b = &a;   // b is the memory address of object a
*b = 6         // *b is now a pseudonym for a
```

- b is called a pointer

- The *dereferencing* operator $\star$ turns the pointer back into the object

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;     // creates an object a with value 5
int* b = &a;   // b is the memory address of object a
*b = 6         // *b is now a pseudonym for a
```

- `b` is called a pointer

- The *dereferencing* operator $\star$ turns the pointer back into the object

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;     // creates an object a with value 5
int* b = &a;   // b is the memory address of object a
*b = 6         // *b is now a pseudonym for a
```

- `b` is called a pointer

- The *dereferencing* operator $\star$ turns the pointer back into the object

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                      // w is an instance of class Widget
Widget* wpt = new Widget(args);  // pointer to instance of class Widget
```

- To call a member function, `func()`, of class `w` you use

```
    w.func()
```

- To call a member function of `wpt` use either

```
(*wpt).func();   // dereference object and call member function
```

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                      // w is an instance of class Widget
Widget* wpt = new Widget(args);     // pointer to instance of class Widget
```

- To call a member function, `func()`, of class `w` you use

```
    w.func()
```

- To call a member function of `wpt` use either

```
(*wpt).func();   // dereference object and call member function
```

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                          // w is an instance of class Widget
Widget* wpt = new Widget(args); // pointer to instance of class Widget
```

- To call a member function, `func()`, of class `w` you use

```
    w.func()
```

- To call a member function of `wpt` use either

```
(*wpt).func();  // dereference object and call member function
```

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                    // w is an instance of class Widget
Widget* wpt = new Widget(args); // pointer to instance of class Widget
```

- To call a member function, `func()`, of class `w` you use

```
    w.func()
```

- To call a member function of `wpt` use either

```
(*wpt).func();  // dereference object and call member function
```

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                    // w is an instance of class Widget
Widget* wpt = new Widget(args);   // pointer to instance of class Widget
```

- To call a member function, `func()`, of class `w` you use

```
w.func()
```

- To call a member function of `wpt` use either

```
(*wpt).func();   // dereference object and call member function
wpt->func();     // easy to type
```

Inheritence

- C++ allows classes to inherit from other classes
- Suppose `Square` and `Circle` inherits from `Shape`
- If `Shape` has a (virtual) member function `area` then `Square` and `Circle` can redefine this

```
class Square: public Shape {
  private:
    double l;

  public:
    Square(double len) {l=len;}   // constructor
    double area() {return l*l;}   // define area
}
```

Inheritence

- C++ allows classes to inherit from other classes
- Suppose `Square` and `Circle` inherits from `Shape`
- If `Shape` has a (virtual) member function `area` then `Square` and `Circle` can redefine this

```
class Square: public Shape {
  private:
    double l;

  public:
    Square(double len) {l=len;}   // constructor
    double area() {return l*l;}   // define area
}
```

Inheritence

- C++ allows classes to inherit from other classes
- Suppose `Square` and `Circle` inherits from `Shape`
- If `Shape` has a (virtual) member function `area` then `Square` and `Circle` can redefine this

```
class Square: public Shape {
  private:
    double l;

  public:
    Square(double len) {l=len;}   // constructor
    double area() {return l*l;}   // define area
}
```

Inheritence

- C++ allows classes to inherit from other classes
- Suppose `Square` and `Circle` inherits from `Shape`
- If `Shape` has a (virtual) member function `area` then `Square` and `Circle` can redefine this

```
class Square: public Shape {
  private:
    double l;

  public:
    Square(double len) {l=len;}   // constructor
    double area() {return l*l;}   // define area
}
```

Polymorphism

- Polymorphism is a way of using inheritance where we instantiate a parent pointer with a child class

```
Shape* shape = new Square(2.5);

cout << shape->area() << endl;
```

- This provides a clean way of choosing a behaviour depending on the object type

- It is used in *iterators* which we will come to later in the course

Polymorphism

- Polymorphism is a way of using inheritance where we instantiate a parent pointer with a child class

```
Shape* shape = new Square(2.5);

cout << shape->area() << endl;
```

- This provides a clean way of choosing a behaviour depending on the object type

- It is used in *iterators* which we will come to later in the course

Polymorphism

- Polymorphism is a way of using inheritance where we instantiate a parent pointer with a child class

```
Shape* shape = new Square(2.5);

cout << shape->area() << endl;
```

- This provides a clean way of choosing a behaviour depending on the object type

- It is used in *iterators* which we will come to later in the course

Polymorphism

- Polymorphism is a way of using inheritance where we instantiate a parent pointer with a child class

```
Shape* shape = new Square(2.5);

cout << shape->area() << endl;
```

- This provides a clean way of choosing a behaviour depending on the object type

- It is used in *iterators* which we will come to later in the course

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

```
int* pt = new int[20];
```

  creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]`

- We can free this up with

```
delete[] pt;
```

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

```
int* pt = new int[20];
```

  creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]`

- We can free this up with

```
delete[] pt;
```

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

  ```
  int* pt = new int[20];
  ```

  creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]` (which is equivalent to `*(pt+i)`)

- We can free this up with

  ```
  delete[] pt;
  ```

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

  ```
  int* pt = new int[20];
  ```

  creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]` (which is equivalent to `*(pt+i)`)—this is the same as C

- We can free this up with

  ```
  delete[] pt;
  ```

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

```
int* pt = new int[20];
```

  creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]` (which is equivalent to `*(pt+i)`)—this is the same as C

- We can free this up with

```
delete[] pt;
```

References

- C and C++ also provides references

```
int a = 5;          // create a memory location called a
int& b = a;         // b is a pseudonym for a
b = 6               // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;            // create a memory location called a
int& b = a;           // b is a pseudonym for a
b = 6                 // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;             // create a memory location called a
int& b = a;            // b is a pseudonym for a
b = 6                  // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;              // create a memory location called a
int& b = a;             // b is a pseudonym for a
b = 6                   // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;          // create a memory location called a
int& b = a;         // b is a pseudonym for a
b = 6               // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;           // create a memory location called a
int& b = a;          // b is a pseudonym for a
b = 6                // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;           // create a memory location called a
int& b = a;          // b is a pseudonym for a
b = 6                // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;              // create a memory location called a
int& b = a;             // b is a pseudonym for a
b = 6                   // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}    // define function f

void g(int& x) {x += 2;}   // define function g

int a = 5;

f(a);                       // does nothing a=5
g(a);                       // now a=7
```

References

- C and C++ also provides references

```
int a = 5;             // create a memory location called a
int& b = a;            // b is a pseudonym for a
b = 6                  // both b and a are now 6
```

- References are like dereferenced pointers

- There are many uses of references, one is so we can make functions change the value of their arguments

```
void f(int x) {x += 6;}   // define function f

void g(int& x) {x += 2;}  // define function g

int a = 5;

f(a);                        // does nothing a=5
g(a);                        // now a=7
```

Saving Copying

- When we declare a function `f(Widget w)` then widget `w` is copied to the function (this is known as passed by value)

- If widget is big, even if we don't want to change it we might **not** want to copy it

```
void f(const Widget& w);
void g(Widget w);
```

- In both cases `w` is a Widget, but function `f` avoids copying its input

Saving Copying

- When we declare a function `f(Widget w)` then widget `w` is copied to the function (this is known as passed by value)

- If widget is big, even if we don't want to change it we might **not** want to copy it

```
void f(const Widget& w);
void g(Widget w);
```

- In both cases `w` is a Widget, but function `f` avoids copying its input

Saving Copying

- When we declare a function `f(Widget w)` then widget `w` is copied to the function (this is known as passed by value)

- If widget is big, even if we don't want to change it we might **not** want to copy it

```
void f(const Widget& w);
void g(Widget w);
```

- In both cases `w` is a Widget, but function `f` avoids copying its input

Saving Copying

- When we declare a function `f(Widget w)` then widget `w` is copied to the function (this is known as passed by value)
- If widget is big, even if we don't want to change it we might **not** want to copy it

```
void f(const Widget& w);
void g(Widget w);
```

- In both cases `w` is a Widget, but function `f` avoids copying its input

Outline

1. C with Classes
2. New
3. **Overloading**
4. Templates

Overloading

- C and C++ allow you to define different functions with the same name but different arguments

```
void func(int a);       // called if argument is an int
void func(double a);    // called if argument is a double
```

- Needs to be used sensibly, but provides flexibility

Overloading

- C and C++ allow you to define different functions with the same name but different arguments

```
void func(int a);       // called if argument is an int
void func(double a);    // called if argument is a double
```

- Needs to be used sensibly, but provides flexibility

Example

- In the second order statistics class we could define a member function

```
void add(const Sos& rhs);
```

- With an implementation

```
void Sos::add(const Sos& rhs)
{
  double total = n + rhs.n;
  double diff = rhs.mu-mu;
  mu += rhs.n*diff/total;
  Q += rhs.Q + n*rhs.n*diff*diff/total;
  n = total;

  return rhs;
}
```

Overloading Continued

- This allows us to add second order statistics

```
Sos total;
for(int i=0; i<10; ++i) {
  Sos local;
  for(int j=0; j<100; ++j) {
    // compute X
    cout << local.mean() << ',' << local.error() << endl;
    local.add()
  }
  total.add(local)
  cout << total.mean() << ',' << total.error() << endl;
}
```

Opeartor Overloading

- C++ like python allows us to overload operators
- Rather than using add I might prefer to use

```
class Sos {
  ...
  double operator+=(double x) { add(x); return(x); }
}
```

- Then we can write

```
Sos sos;
sos += X;
```

Opeartor Overloading

- C++ like python allows us to overload operators
- Rather than using add I might prefer to use

```
class Sos {
  ...
  double operator+=(double x) { add(x); return(x); }
}
```

- Then we can write

```
Sos sos;
sos += X;
```

Opeartor Overloading

- C++ like python allows us to overload operators
- Rather than using add I might prefer to use

```
class Sos {
  ...
  double operator+=(double x) { add(x); return(x); }
}
```

- Then we can write

```
Sos sos;
sos += X;
```

Overloading <<

- To print an object of type `Sos` we define

```
ostream& operator<<(ostream& out, const Sos& d)
{
  out << d.mean() << " " << d.error();
  return(out);
}
```

- We can then print

```
Sos sos;
...

cout << sos << endl;
```

- I've made `sos.h` and `sos.cc` available on the web site

Overloading <<

- To print an object of type `Sos` we define

```
ostream& operator<<(ostream& out, const Sos& d)
{
  out << d.mean() << " " << d.error();
  return(out);
}
```

- We can then print

```
Sos sos;
...

cout << sos << endl;
```

- I've made `sos.h` and `sos.cc` available on the web site

Overloading <<

- To print an object of type `Sos` we define

```
ostream& operator<<(ostream& out, const Sos& d)
{
  out << d.mean() << " " << d.error();
  return(out);
}
```

- We can then print

```
Sos sos;
...

cout << sos << endl;
```

- I've made `sos.h` and `sos.cc` available on the web site

Overloading <<

- To print an object of type `Sos` we define

```
ostream& operator<<(ostream& out, const Sos& d)
{
  out << d.mean() << " " << d.error();
  return(out);
}
```

- We can then print

```
Sos sos;
...

cout << sos << endl;
```

- I've made `sos.h` and `sos.cc` available on the web site—I use them a lot, you might want to keep them around

Outline

1. C with Classes
2. New
3. Overloading
4. **Templates**

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```

Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

```
myclass<int> instance;
```

  the C++ compiler takes your template and substitutes the `T` with **`int`**

- This is both simple and powerful

Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

```
myclass<int> instance;
```

  the C++ compiler takes your template and substitutes the `T` with **`int`**

- This is both simple and powerful

Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

  ```
  myclass<int> instance;
  ```

  the C++ compiler takes your template and substitutes the `T` with **`int`**
- This is both simple and powerful

Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

  ```
  myclass<int> instance;
  ```

  the C++ compiler takes your template and substitutes the `T` with **`int`**
- This is both simple and powerful

Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

  ```
  myclass<int> instance;
  ```

  the C++ compiler takes your template and substitutes the `T` with **`int`**
- This is both simple and powerful

Template Functions

- As well as classes I can create template functions

```
template <typename T>
T accumulate(const vector<T>& vec) {
  T sum = 0;
  for(int i=0; i<vec.size(); ++i) {
    sum += vec[i];
  }
  return sum
}
```

- This will work with `vector<int>`, `vector<double>`

Template Functions

- As well as classes I can create template functions

```
template <typename T>
T accumulate(const vector<T>& vec) {
  T sum = 0;
  for(int i=0; i<vec.size(); ++i) {
    sum += vec[i];
  }
  return sum
}
```

- This will work with `vector<int>`, `vector<double>`

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++