Introduction to C++

Introduction

- C++ improves on many of C's features
- C++ provides object-oriented programming (OOP)
- C++ is a superset to C
- No ANSI standard exists yet (in 1994)

Some C++ Code

Segment.H

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};

#endif // __SEGMENT_HEADER__
```

Segment.C

```
#include "Segment.H"
#include "Point.H"

Segment::Segment()
{
    m_p0 = new Point(0, 0);
    m_p1 = new Point(1, 1);
}
Segment::~Segment()
{
    delete m_p0;
    delete m_p1;
}
```

#include

```
#include "Segment.H"
```

Insert header file at this point.

```
#include <iostream>
```

Use library header.

Header Guards

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

// contents of Segment.H
//...

#endif
```

- To ensure it is safe to include a file more than once.

Header Guards

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

// contents of segment.H
//...

#endif
```

If this variable is not defined…

Define it.

End of guarded area.

C++ Single-Line Comments

- In C,
  /* This is a single-line comment. */
- In C++,
  // This is a single-line comment
- But note that compilers will accept both!

C++ Stream Input/Output

- In C,

```
printf("Enter new tag: ");
scanf("%d", &tag);
printf("The new tag is: %d\n", tag);
```

- In C++,

```
cout << "Enter new tag: ";
cin >> tag;
cout << "The new tag is : " << tag << '\n';
```

An Example

```
// Simple stream input/output
#include <iostream.h>

main()
{
  cout << "Enter your age: ";
  int myAge;
  cin >> myAge;

  cout << "Enter your friend's age: ";
  int friendsAge;
  cin >> friendsAge;
```

An Example .

```
  if (myAge > friendsAge)
    cout << "You are older.\n";
  else
    if (myAge < friendsAge)
      cout << "You are younger.\n";
    else
      cout << "You and your friend are the same age.\n";

  return 0;
}
```

Declarations in C++

- In C++, declarations can be placed anywhere
  - Except in the condition of a while, do/while, for or if structure
- An example

```
cout << "Enter two integers: ";
int x, y;
cin >> x >> y;
cout << "The sum of " << x << " and " << y << " is " << x + y <<
  '\n';
```

- Another example

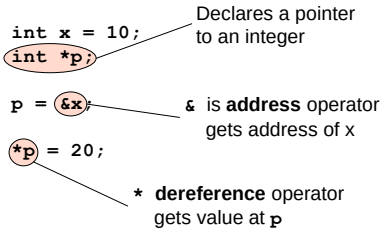
```
for (int i = 0; i <= 5; i++)
  cout << i << '\n';
```

Data Types in C++

```
struct Name {
    char first[10];
    char last[10];
};
```

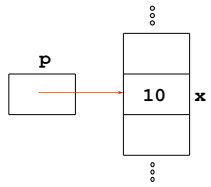
- In C,
  - struct Name stdname;
- In C++,
  - Name stdname;
- The same is true for enums and unions

Pointers



Pointers .

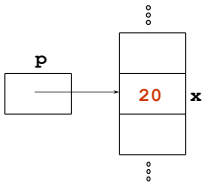
```
int x = 10;
int *p;

p = &x;
```



p gets the address of **x** in memory.

Pointers ..

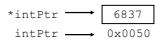
```
int x = 10;
int *p;

p = &x;

*p = 20;
```



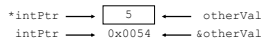
***p** is the value at the address **p**.

Pointers Example

```
int *intPtr;                    Create a pointer

intPtr = new int;               Allocate memory

*intPtr = 6837;                 Set value at given address
```




```
delete intPtr;                  Deallocate memory

int otherVal = 5;               Change intPtr to point to
intPtr = &otherVal;             a new location
```



Allocating memory using new

- Point *p = new Point(5, 5);
  - **new** can be thought of a function with slightly strange syntax
  - **new** allocates space to hold the object
  - **new** calls the object's constructor
  - **new** returns a pointer to that object

Deallocating memory using delete

```
// allocate memory
Point *p = new Point(5, 5);

...
// free the memory
delete p;
```

For every call to new, there must be exactly one call to delete

Arrays

Stack allocation

```
int intArray[10];
Int Array[0] = 6837;
```

Heap allocation

```
int *intArray;
intArray = new int[10];
intArray[0] = 6837;

...

delete[] intArray;
```

More Arrays Examples

int x = 10;
int* nums1 = new int[10]; // ok
int* nums2 = new int[x]; // ok

- Initializes an array of 10 integers on the heap
- C equivalent of
  int* nums = (int*)malloc(x * sizeof(int));

Multidimensional Arrays

int x = 3, y = 4;
int* nums3 = new int [x] [4] [5]; // ok
int* nums4 = new int [x] [y] [5]; // BAD!

- Initializes a multidimensional array
- Only the first dimension can be a variable
  - The rest must be constants
- Use single dimension arrays to fake multidimensional ones

Strings

A string in C++ is an array of characters

```
char myString[20];
strcpy(myString, "Hello World");
```

Strings are terminated with the `NULL` or `'\0'` character

```
myString[0] = 'H';
myString[1] = 'i';
myString[2] = '\0';

printf("%s", myString);
```

output: `Hi`

Parameter Passing

Pass by value

```
int add(int a, int b) {
  return a+b;
}

int a, b, sum;
sum = add(a, b);
```

Make a local copy of `a` and `b`

Pass by reference

```
int add(int *a, int *b) {
  return *a + *b;
}

int a, b, sum;
sum = add(&a, &b);
```

Pass pointers that reference `a` and `b`. Changes made to `a` or `b` will be reflected outside the `add` routine

Parameter Passing .

Pass by reference – alternate notation

```
int add(int &a, int &b) {
  return a+b;
}

int a, b, sum;
sum = add(a, b);
```

Class Basics

```
#ifndef _IMAGE_H_
#define _IMAGE_H_
```
Prevents multiple references

```
#include <assert.h>
#include "vectors.h"
```
Include a library file
Include a local file

```
class Image {

public:
  ...
```
Variables and functions accessible from anywhere

```
private:
  ...
```
Variables and functions accessible only from within this class's functions

```
};

#endif
```

Creating an instance

Stack allocation

```
Image myImage;
myImage.SetAllPixels(ClearColor);
```

Heap allocation

```
Image *imagePtr;
imagePtr = new Image();
imagePtr->SetAllPixels(ClearColor);

...

delete imagePtr;
```

Organizational Strategy

image.h — Header file: Class definition & function prototypes

```
void SetAllPixels(const Vec3f &color);
```

image.C — .C file: Full function definitions

```
void Image::SetAllPixels(const Vec3f &color) {
    for (int i = 0; i < width*height; i++)
      data[i] = color;
}
```

main.C — Main code: Function references

```
myImage.SetAllPixels(clearColor);
```

Constructors & Destructors

```
class Image {
public:
  Image(void) {
    width = height = 0;
    data = NULL;
  }

 ~Image(void) {
    if (data != NULL)
      delete[] data;
  }

  int width;
  int height;
  Vec3f *data;
};
```

Constructor:
Called whenever a new instance is created

Destructor:
Called whenever an instance is deleted

Constructors Specifics

Constructors can also take parameters

```
Image(int w, int h) {
  width = w;
  height = h;
  data = new Vec3f[w*h];
}
```

Using this constructor with stack or heap allocation:

```
Image myImage = Image(10, 10);
```
stack allocation

```
Image *imagePtr;
imagePtr = new Image(10, 10);
```
heap allocation

The Copy Constructor

```
Image(Image *img) {
  width = img->width;
  height = img->height;
  data = new Vec3f[width*height];
  for (int i=0; i<width*height; i++)
    data[i] = img->data[i];
}
```

A default copy constructor is created automatically, but it is often not what you want:

```
Image(Image *img) {
  width = img->width;
  height = img->height;
  data = img->data;
}
```

Destructors Specifics

- Delete calls the object's destructor
- Delete frees space occupied by the object
- A destructor cleans up after the object
- Releases resources such as memory

Destructors – An Example

```
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};
```

Destructors – An Example .

```
Segment::Segment()
{
    m_p0 = new Point(0, 0);
    m_p1 = new Point(1, 1);
}
Segment::~Segment()
{
    delete m_p0;
    delete m_p1;
}
```

Syntactic Sugar "->"

```
Point *p = new Point(5, 5);

// Access a member function:
(*p).move(10, 10);

// Or more simply:
p->move(10, 10);
```

Passing Classes as Parameters

If a class instance is passed by value, the copy constructor will be used to make a copy

```
bool IsImageGreen(Image img);
```

Computationally expensive

It's much faster to pass by reference:

```
bool IsImageGreen(Image *img);
```

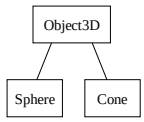
or

```
bool IsImageGreen(Image &img);
```

Class Hierarchy

Child classes inherit parent attributes

```
class Object3D {
  Vec3f color;
};

class Sphere : public Object3D {
  float radius;
};

class Cone : public Object3D {
  float base;
  float height;
};
```



Class Hierarchy .

Child classes can *call* parent functions

```
Sphere::Sphere() : Object3D() {
  radius = 1.0;
}
```

Call the parent constructor

Child classes can *override* parent functions

Superclass

```
class Object3D {
  virtual void setDefaults(void) {
    color = RED; }
};
```

Subclass

```
class Sphere : public Object3D {
  void setDefaults(void) {
    color = BLUE;
    radius = 1.0 }
};
```

Introducing `const`

```
void Math::printSquare(const int& i)
{
    i = i*i;
    cout << i << endl;
}

int main()
{
    int i = 5;
    Math::printSquare(i);
    Math::printCube(i);
}
```

Summary with Header File

header file
begin header guard
forward declaration
class declaration
constructor
destructor
member variables
need semi-colon
end header guard

Segment.H

```
#ifndef __SEGMENT_HEADER__
#define __SEGMENT_HEADER__

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
private:
    Point *m_p0, *m_p1;
};

#endif // __SEGMENT_HEADER__
```

Can also pass pointers to `const`

```
void Math::printSquare(const int* pi)
{
    *pi = (*pi) * (*pi);
    cout << i << endl;
}
int main()
{
    int i = 5;
    Math::printSquare(&i);
    Math::printCube(&i);
}
```

Declaring things `const`

const River nile;

const River* nilePc;

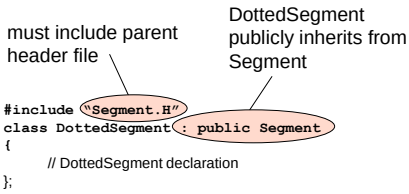
River* const nileCp;

const River* const nileCpc

Read pointer declarations right to left

```
// A const River
const River nile;

// A pointer to a const River
const River* nilePc;

// A const pointer to a River
River* const nileCp;

// A const pointer to a const River
const River* const nileCpc
```

Inheritance



Virtual

- In Java every method invocation is dynamically bound, meaning for every method invocation the program checks if a sub-class has overridden the method
  - You can override this (somewhat) by using the keyword "final" in Java
- In C++ you have to declare the method virtual if you want this functionality
  - So, "virtual" is the same thing as "not final"
- Just like you rarely say things are final in Java, you should rarely not say things are virtual in C++

Virtual Functions in C++

A superclass pointer can reference a subclass object

```
Sphere *mySphere = new Sphere();
Object3D *myObject = mySphere;
```

If a superclass has virtual functions, the correct subclass version will automatically be selected

Superclass

```
class Object3D {
  virtual void intersect(Ray *r, Hit *h);
};
```

Subclass

```
class Sphere : public Object3D {
  virtual void intersect(Ray *r, Hit *h);
};
```

```
myObject->intersect(ray, hit);
```

Actually calls `Sphere::intersect`

Pure Virtual Functions

A *pure virtual function* has a prototype, but no definition. Used when a default implementation does not make sense

```
class Object3D {
  virtual void intersect(Ray *r, Hit *h) = 0;
};
```

A class with a pure virtual function is called a *pure virtual class* and cannot be instantiated

However, its subclasses can

The `main` function

This is where your code begins execution

```
int main(int argc, char** argv);
```

`argc`: Number of arguments

`argv`: Array of strings

`argv[0]` is the program name

`argv[1]` through `argv[argc-1]` are command-line input

Coding tips

Use the `#define` compiler directive for constants

```
#define PI 3.14159265
#define MAX_ARRAY_SIZE 20
```

Use the `printf` or `cout` functions for output and debugging

```
printf("value: %d, %f\n", myInt, myFloat);
cout << "value:" << myInt << ", " << myFloat << endl;
```

Use the `assert` function to test "always true" conditions

```
assert(denominator != 0);
quotient = numerator/denominator;
```

Coding tips .

After you `delete` an object, also set its value to `NULL`
(This is not done for you automatically)

```
delete myObject;
myObject = NULL;
```

This will make it easier to debug memory allocation errors

```
assert(myObject != NULL);
myObject->setColor(RED);
```

Segmentation Faults

Typical causes:

```
int intArray[10];
intArray[10] = 6837;
```

Access outside of array bounds

```
Image *img;
img->SetAllPixels(ClearColor);
```

Attempt to access a `NULL` or previously deleted pointer

These errors are often very difficult to catch and can cause erratic, unpredictable behavior

Common Pitfalls

```
void setToRed(Vec3f v) {
  v = RED;
}
```

Since `v` is passed by value, it will not get updated outside of The `set` function

The fix:

```
void setToRed(Vec3f &v) {
  v = RED;
}
```

or

```
void setToRed(Vec3f *v) {
  *v = RED;
}
```

Common Pitfalls ..

```
Sphere* getRedSphere() {
  Sphere s = Sphere(1.0);
  s.setColor(RED);
  return &s;
}
```

C++ automatically deallocates stack memory when the function exits, so the returned pointer is invalid

The fix:

```
Sphere* getRedSphere() {
  Sphere *s = new Sphere(1.0);
  s->setColor(RED);
  return s;
}
```

It will then be your responsibility to delete the Sphere object later

Advanced topics

- Lots of advanced topics, but a few will be required for this course
  - friend or protected class members
  - inline functions
  - static functions and variables
  - operator overloading
  - compiler directives

Some Useful Links

- C++ Programming
  - http://www.syvum.com/squizzes/cpp/
- Online C/C++ Documentation
  - http://www.thefreecountry.com/documentation/onlinecpp.shtml
- C++ Language Tutorials
  - http://www.cs.wustl.edu/~schmidt/C++/
- The C++ Programming Language
  - http://www.research.att.com/~bs/C++.html