C++ Tutorial

Emily Whiting & Zhunping Zhang
Overview

• Basic Syntax
• Pointers
• Dynamic Memory
• Parameter passing
• Class basics
• Constructors & destructors
• Class Hierarchy
• Virtual Functions
• Organizational Strategy
• Coding tips
• Compiling
#include <iostream>
using namespace std;

float c(float x) {
    return x*x*x;
}

int main() {
    float x;
    cin >> x;
    cout << c(x) << endl;
    return 0;
}
The **main** function

This is where your code begins execution

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

<table>
<thead>
<tr>
<th>↑</th>
<th>↑</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of arguments</td>
<td>Array of strings</td>
</tr>
</tbody>
</table>

`argv[0]` is the program name

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

int *intPtr;

intPtr = new int;

*intPtr = 6837;

delete intPtr;

int otherVal = 5;
intPtr = &otherVal;

Create a pointer
Allocate memory
Set value at given address
Deallocate memory
Change intPtr to point to a new location
Dynamic Memory

Fixed size array

```cpp
int intArray[10];
intArray[0] = 6837;
```

```cpp
#include <iostream>

int main() { 
    int n;
    cin >> n;
    int intArray[n];
    intArray[0] = 6837;

    return 0;
}
```

Arrays must have known sizes at compile time

This doesn’t compile
#include <iostream>

int main() {
    int n;
    cin >> n;

    int *intArray;
    intArray = new int[n];
    intArray[0] = 6837;

    ...

    delete[] intArray;
    return 0;
}
STL vector

```cpp
#include <vector>
using namespace std;

int func(int n) {
    vector<float> f(n);
    f[0] = 6837;
}
```

`vector` is a resizable array with dynamic memory handled for you

If you can, use the STL and avoid dynamic memory

---

**alternative method**

```cpp
int func(int n) {
    vector<float> f(n);
    f.push_back(6837);
}
```

Methods are called with the dot operator (same as Java)
Parameter Passing

pass by value

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

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

Make a local copy of a and b

pass by reference

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

```c
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);
Parameter Passing

doesn’t work

```c
int bar = 0;
AddTwo(bar);

void AddTwo(int val) {
    val += 2;
}
```

Since `bar` is passed by value, it will not get updated outside of the `AddTwo` function

works

```c
int* bar;
*bar = 0;
AddTwo(bar);

void AddTwo(int* val) {
    *val += 2;
}
```
Parameter Passing

doesn’t work

```cpp
vector<int> v;
PushTwo(v);

void PushTwo(vector<int> v) {
    v.push_back(2);
}
```

works

```cpp
vector<int> v;
PushTwo(&v);

void PushTwo(vector<int>* v) {
    v->push_back(2);
}
```
Parameter Passing

works

int* bar;
*bar = 0;
AddTwo(*bar);

void AddTwo(int& val) {
    val += 2;
}

also works

vector<int> v;
PushTwo(v);

void PushTwo(vector<int>& v) {
    v.push_back(2);
}
#ifndef _IMAGE_H_
#define _IMAGE_H_

#include <assert.h>
#include "vectors.h"

class Image {

public:
    ...

private:
    ...

};

#endif
Creating an instance

**Stack allocation**

Image myImage;
myImage.SetAllPixels(ClearColor);

**Heap allocation**

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

...  
delete imagePtr;
Constructors & Destructors

```cpp
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

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
```
Passing Classes as Parameters

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

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

*Computationally expensive*

It’s much faster to pass by reference:

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

or

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

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

Child classes can *override* parent functions

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

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

Call the parent constructor
Virtual Functions

A superclass pointer can reference a subclass object

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

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

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

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).
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);
Coding tips

Use the `define` compiler directive for constants

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

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

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

Use the `assert` function to test “always true” conditions

```c
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 fault (core dumped)

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.
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.
Compiling

```
g++ -c main.cc
|   ↓   |
|   ↓   |
|  main.o |

|   ↓   |
|  g++ -c mydraw.cc |
|   ↓   |
|  mydraw.o |

|   ↓   |
|  g++ -c mymath.cc |
|   ↓   |
|  mymath.o |

|   ↓   |
|  g++ -o myprogram main.o mathstuff.o drawstuff.o |
|   ↓   |
|  myprogram |
```
// This is main.cc
#include <GL/glut.h>
#include <iostream>
using namespace std;

int main() {
    cout << "Hello!" << endl;
    glVertex3d(1,2,3);
    return 0;
}

% g++ -c main.cc
% g++ -o myprogram -lglut main.o
% ./myprogram

Libraries

Include OpenGL functions
Include standard IO functions
Long and tedious explanation

Calls function from standard IO
Calls function from OpenGL

Make object file
Make executable, link GLUT
Execute program
Most assignments include makefiles, which describe the files, dependencies, and steps for compilation.

You can just type `make`

So you don’t have to know the stuff from the past few slides.

But it’s nice to know.
Resources

• The C++ Programming Language
  – A book by Bjarne Stroustrup, inventor of C++

• The STL Programmer’s Guide
  – Contains documentation for the standard template library

• Java to C++ Transition Tutorial
  – Probably the most helpful, since you’ve all taken 6.170
  – http://www.cs.brown.edu/courses/csci1230/javatoc.htm