C++ Tutorial

Rob Jagnow

Overview

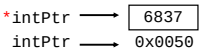
- Pointers
- Arrays and strings
- Parameter passing
- Class basics
- Constructors & destructors
- Class Hierarchy
- Virtual Functions
- Coding tips
- Advanced topics

Pointers

```
int *intPtr;
```
Create a pointer

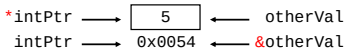
```
intPtr = new int;
```
Allocate memory

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




```
delete intPtr;
```
Deallocate memory

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



Arrays

Stack allocation

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

Heap allocation

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

...

delete[] intArray;
```

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

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;
}
```

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);
```

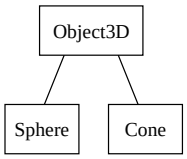
Class Hierarchy

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 }
};
```

Virtual Functions

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);
```

Number of arguments

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 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.

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 few will be required for this course

- `friend` or `protected` class members
- inline functions
- `const` or `static` functions and variables
- compiler directives
- operator overloading

```
Vec3f& operator+(Vec3f &a, Vec3f &b);
```