C++ is fun – Part Six

at Turbine/Warner Bros.!

Russell Hanson

Syllabus

1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
3) Basic data structures and how to use them, opening files and performing operations on files -- **We're here** April 8-10
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
Project 1 Due – April 17
5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
7) Basic threads models and some simple databases SQLite May 6-8
8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
Project 2 Due May 15
9) How to download an API and learn how to use functions in that API, Windows Foundation Classes May 20-22
10) Designing and implementing a simple game in C++ May 27-29
11) Selected topics – Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC June 3-5
12) Working on student projects - June 10-12
Final project presentations Project 3/Final Project Due June 12

Project 1 due in 1 week

- Therefore, we're going to spend time in class working on the project, last 30 minutes of class

Class Activity: Pointers and memory management!

```
#include "stdafx.h"
// This program uses the & operator to determine a variable's
// address and the sizeof operator to determine its size.
#include <iostream>
using namespace std;
int main()
{
   int x = 25;
   cout << "The address of x is " << &x << endl;
   cout << "The size of x is " << sizeof(x) << " bytes\n";
   cout << "The value in x is " << x << endl;
   system("PAUSE");
   return 0;
}
```

The address of x is 0029FE74
The size of x is 4 bytes
The value in x is 25
Press any key to continue . . .

Pointer Variables

CONCEPT: *Pointer variables*, which are often just called *pointers*, are designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables.

A *pointer variable*, which often is just called a *pointer*, is a special variable that holds a memory address. Just as `int` variables are designed to hold integers, and `double` variables are designed to hold floating-point numbers, pointer variables are designed to hold memory addresses.

Memory addresses identify specific locations in the computer's memory. Because a pointer variable holds a memory address, it can be used to hold the location of some other piece of data. This should give you a clue as to why it is called a pointer: It "points" to some piece of data that is stored in the computer's memory. Pointer variables also allow you to work with the data that they point to.

```
int* ptr;
```

This style of definition might visually reinforce the fact that `ptr`'s data type is not `int`, but pointer-to-`int`. Both definition styles are correct.

Program 9-2 demonstrates a very simple usage of a pointer: storing and printing the address of another variable.

Program 9-2

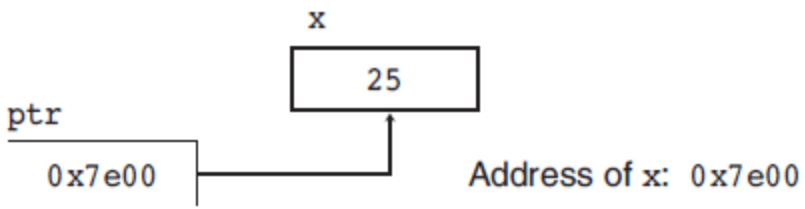
```
1  // This program stores the address of a variable in a pointer.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int x = 25;     // int variable
8     int *ptr;       // Pointer variable, can point to an int
9
10    ptr = &x;       // Store the address of x in ptr
11    cout << "The value in x is " << x << endl;
12    cout << "The address of x is " << ptr << endl;
13    return 0;
14 }
```

Program Output

```
The value in x is 25
The address of x is 0x7e00
```

Figure 9-4 illustrates the relationship between `ptr` and `x`.

Figure 9-4



The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, `ptr` could be used to change the contents of the variable `x`. This is done with the *indirection operator*, which is an asterisk (*). When the indirection operator is placed in front of a pointer variable name, it *dereferences* the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 9-3.

Program 9-3

```
1  // This program demonstrates the use of the indirection operator.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7     int x = 25;     // int variable
8     int *ptr;       // Pointer variable, can point to an int
9
10    ptr = &x;       // Store the address of x in ptr
11
12    // Use both x and ptr to display the value in x.
13    cout << "Here is the value in x, printed twice:\n";
14    cout << x << endl;      // Displays the contents of x
15    cout << *ptr << endl;   // Displays the contents of x
16
17    // Assign 100 to the location pointed to by ptr. This
18    // will actually assign 100 to x.
19    *ptr = 100;
20
21    // Use both x and ptr to display the value in x.
22    cout << "Once again, here is the value in x:\n";
23    cout << x << endl;      // Displays the contents of x
24    cout << *ptr << endl;   // Displays the contents of x
25    return 0;
26 }
```

Class Activity: Use the indirection operator

```
#include "stdafx.h"
// This program demonstrates the use of the indirection operator.
#include <iostream>
using namespace std;
int main()
{
   int x = 25;    // int variable
   int *ptr;      // Pointer variable, can point to an int
   ptr = &x;      // Store the address of x in ptr

   // Use both x and ptr to display the value in x.
   cout << "Here is the value in x, printed twice:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x

   // Assign 100 to the location pointed to by ptr. This
   // will actually assign 100 to x.
   *ptr = 100;

   // Use both x and ptr to display the value in x.
   cout << "Once again, here is the value in x:\n";
   cout << x << endl;     // Displays the contents of x
   cout << *ptr << endl;  // Displays the contents of x
   system("PAUSE");
   return 0;
}
```

9.3 The Relationship Between Arrays and Pointers

CONCEPT: Array names can be used as constant pointers, and pointers can be used as array names.

You learned in Chapter 7 that an array name, without brackets and a subscript, actually represents the starting address of the array. This means that an array name is really a pointer. Program 9-5 illustrates this by showing an array name being used with the indirection operator.

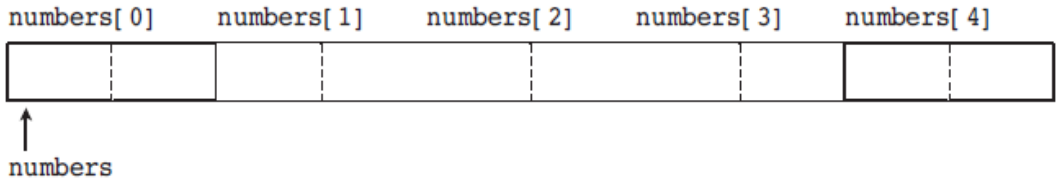
Program 9-5

```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

Program Output

```
The first element of the array is 10
```

Figure 9-5



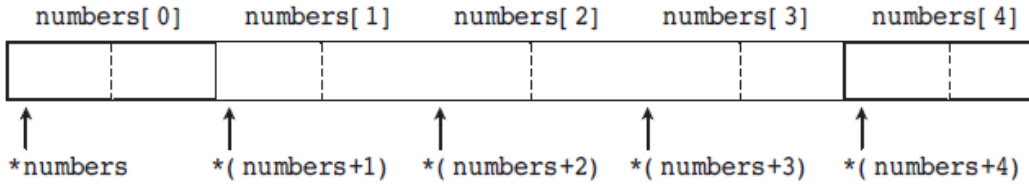
one to `numbers`, you are actually adding `1 * sizeof( short)` to `numbers`. If you add two to `numbers`, the result is `numbers + 2 * sizeof( short)`, and so forth. On a PC, this means the following are true, because `short` integers typically use two bytes:

```
*( numbers + 1) is actually *( numbers + 1 * 2)
*( numbers + 2) is actually *( numbers + 2 * 2)
*( numbers + 3) is actually *( numbers + 3 * 2)
```

and so forth.

This automatic conversion means that an element in an array can be retrieved by using its subscript or by adding its subscript to a pointer to the array. If the expression `*numbers`, which is the same as `*( numbers + 0)`, retrieves the first element in the array, then `*( numbers + 1)` retrieves the second element. Likewise, `*( numbers + 2)` retrieves the third element, and so forth. Figure 9-6 shows the equivalence of subscript notation and pointer notation.

Figure 9-6



NOTE: The parentheses are critical when adding values to pointers. The `*` operator has precedence over the `+` operator, so the expression `*number + 1` is not equivalent to `*( number + 1)`. `*number + 1` adds one to the contents of the first element of the array, while `*( number + 1)` adds one to the address in `number`, then dereferences it.

Class Activity: Access array using pointer notation

```
#include "stdafx.h"
// This program shows an array name being dereferenced with the *
// operator.
#include <iostream>
using namespace std;
int main()
{
   int numbers[] = {10, 20, 30, 40, 50};

   cout << "The first element of the array is ";
   cout << *numbers << endl;
   cout << "The second element of the array is ";
   cout << *(numbers+1) << endl;
   cout << "The third element of the array is ";
   cout << *(numbers+2) << endl;
   cout << "The second element of the array is not ";
   cout << *numbers+1 << endl;
   system("PAUSE");
   return 0;
}
```

When working with arrays, remember the following rule:

array[index] is equivalent to *(*array* + index)

Array names are *pointer constants*

The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, consider the following definitions:

```
double readings[20], totals[20];
double *dptr;
```

These statements are legal:

```
dptr = readings; // Make dptr point to readings.
dptr = totals;   // Make dptr point to totals.
```

But these are illegal:

```
readings = totals;  // ILLEGAL! Cannot change readings.
totals = dptr;      // ILLEGAL! Cannot change totals.
```

Array names are *pointer constants*. You can't make them point to anything but the array they represent.

Pointer arithmetic

Pointer Arithmetic

CONCEPT: **Some mathematical operations may be performed on pointers.**

The contents of pointer variables may be changed with mathematical statements that perform addition or subtraction. This is demonstrated in Program 9-9. The first loop increments the pointer variable, stepping it through each element of the array. The second loop decrements the pointer, stepping it through the array backward.

Not all arithmetic operations may be performed on pointers. For example, you cannot multiply or divide a pointer. The following operations are allowable:

- The ++ and -- operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the + and - operators, or the += and -= operators.
- A pointer may be subtracted from another pointer.

Class Activity: Pointer Arithmetic

```
#include "stdafx.h"
// This program uses a pointer to display the contents of an array.
#include <iostream>
using namespace std;
int main()
{
   const int SIZE = 8;
   int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
   int *numPtr;   // Pointer
   int count;     // Counter variable for loops
   // Make numPtr point to the set array.
   numPtr = set;
   // Use the pointer to display the array contents.
   cout << "The numbers in set are:\n";
   for (count = 0; count < SIZE; count++)
   {
      cout << *numPtr << " ";
      numPtr++;
   }
   // Display the array contents in reverse order.
   cout << "\nThe numbers in set backward are:\n";
   for (count = 0; count < SIZE; count++)
   {
      numPtr--;
      cout << *numPtr << " ";
   }
   system("Pause");
   return 0;
}
```

Pointer to an array vs. array of pointers

```
int main(){
   int (*iptr)[50];
   // int* iptr[50];
   // Error 1           error C2440: '=' : cannot convert from 'int (*)[50]' to 'int *[50]'
   iptr = new int[100000000][50];
   cout << iptr;
   system("Pause");
   return 0;

   /*   The trick is to parenthisize the pointer declaration.
                       This informs C++ that you want a pointer to an array, rather than array of pointers.
                       Writing
                       int (*a)[10];
                       declares that a is a pointer to an array of 10 ints. By constrast,
                       int *a[10];
                       declares that a is an array of 10 pointers to ints.
                       Instead of writing
                       object (*ptr)[10] = new object[5][10];
                       or
                       object (*ptr)[10][15] = new object[5][10][15];
                       you must write
                       typedef object (*ObjectArray)[5][10][15];
                       ObjectArray ptr = (ObjectArray) new object[5][10][15];
                       You would also have to dereference the array whenever you refer to an element. For example,
                       (*ptr)[4][3][2] = 0;
   */
}
```

Or use a vector then you do not need to free the memory, because the std::vector destructor will do it for you.

Checking return value of new

But what if there isn't enough free memory to accommodate the request? What if the program asks for a chunk large enough to hold a 100,000-element array of `floats`, and that much memory isn't available? When memory cannot be dynamically allocated, C++ throws an exception and terminates the program. *Throwing an exception* means the program signals that an error has occurred. You will learn more about exceptions in Chapter 16.

Programs created with older C++ compilers behave differently when memory cannot be dynamically allocated. Under older compilers, the `new` operator returns the address 0, or `NULL` when it fails to allocate the requested amount of memory. (`NULL` is a named constant, defined in the `iostream` file, that stands for address 0.) A program created with an older compiler should always check to see if the `new` operator returns `NULL`, as shown in the following code:

```
iptr = new int[100];
if (iptr == NULL)
{
   cout << "Error allocating memory!\n";
   return;
}
```

The null pointer

NOTE: A pointer that contains the address 0 is called a *null pointer.*

The `if` statement determines whether `iptr` points to address 0. If it does, then the `new` operator was unable to allocate enough memory for the array. In this case, an error message is displayed and the `return` statement terminates the function.

WARNING! The address 0 is considered an unusable address. Most computers store special operating system data structures in the lower areas of memory. Anytime you use the `new` operator with an older compiler, you should always test the pointer for the `NULL` address before you use it.

When a program is finished using a dynamically allocated chunk of memory, it should release it for future use. The `delete` operator is used to free memory that was allocated with `new`. Here is an example of how `delete` is used to free a single variable, pointed to by `iptr`:

```
delete iptr;
```

If `iptr` points to a dynamically allocated array, the `[]` symbol must be placed between `delete` and `iptr`:

```
delete [] iptr;
```

Pointers to Objects, too

Pointers to Objects

You can also define pointers to class objects. For example, the following statement defines a pointer variable named `rectPtr`:

```
Rectangle *rectPtr;
```

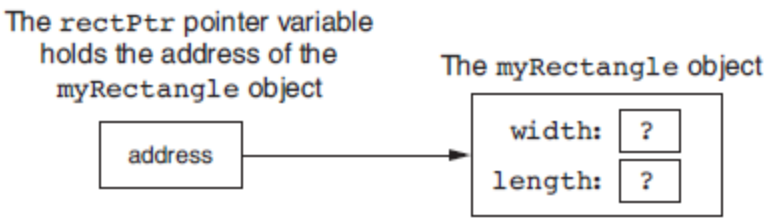
The `rectPtr` variable is not an object, but it can hold the address of a `Rectangle` object. The following code shows an example.

```
Rectangle myRectangle;    // A Rectangle object
Rectangle *rectPtr;       // A Rectangle pointer
rectPtr = &myRectangle;   // rectPtr now points to myRectangle
```

The first statement creates a `Rectangle` object named `myRectangle`. The second statement creates a `Rectangle` pointer named `rectPtr`. The third statement stores the address of the `myRectangle` object in the `rectPtr` pointer. This is illustrated in Figure 13-10.

```
1  // This program demonstrates a simple class.
2  #include <iostream>
3  using namespace std;
4
5  // Rectangle class declaration.
6  class Rectangle
7  {
8     private:
9        double width;
10       double length;
11    public:
12       void setWidth(double);
13       void setLength(double);
14       double getWidth() const;
15       double getLength() const;
16       double getArea() const;
17 };
```

Figure 13-10

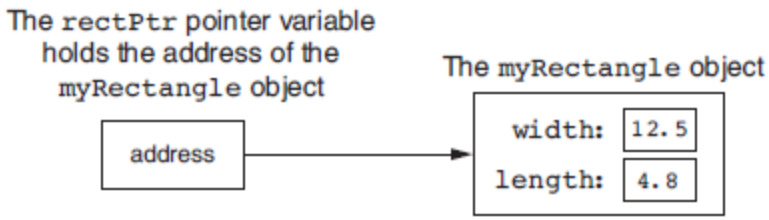


The `rectPtr` pointer can then be used to call member functions by using the -> operator. The following statements show examples.

```
rectPtr->setWidth( 12.5 );
rectPtr->setLength( 4.8 );
```

The first statement calls the `setWidth` member function, passing 12.5 as an argument. Because `rectPtr` points to the `myRectangle` object, this will cause 12.5 to be stored in the `myRectangle` object's `width` variable. The second statement calls the `setLength` member function, passing 4.8 as an argument. This will cause 4.8 to be stored in the

Figure 13-11

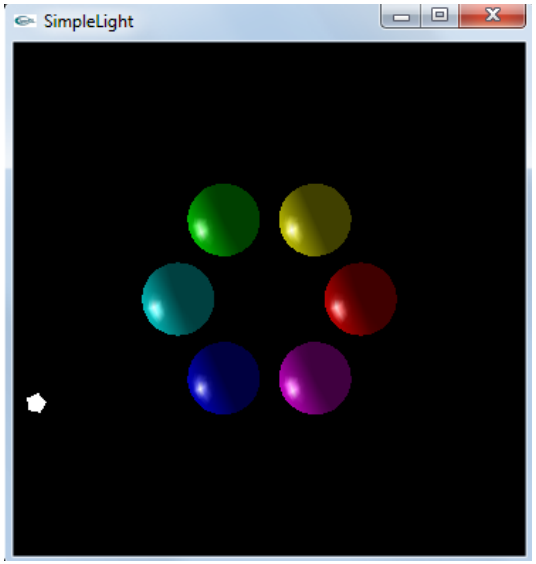


Class object pointers can be used to dynamically allocate objects. The following code shows an example.

```
1  // Define a Rectangle pointer.
2  Rectangle *rectPtr;
3
4  // Dynamically allocate a Rectangle object.
5  rectPtr = new Rectangle;
6
7  // Store values in the object's width and length.
8  rectPtr->setWidth(10.0);
9  rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = 0;
```

```
int main(){
   double number;       // To hold a number
   double totalArea;    // The total area
   Rectangle *kitchen;  // To point to kitchen dimensions
   Rectangle *bedroom;  // To point to bedroom dimensions
   Rectangle *den;      // To point to den dimensions

   // Dynamically allocate the objects.
   kitchen = new Rectangle;
   bedroom = new Rectangle;
   den = new Rectangle;

   // Get the kitchen dimensions.
   cout << "What is the kitchen's length? ";
   cin >> number;                              // Get the length
   kitchen->setLength(number);                  // Store in kitchen object
   cout << "What is the kitchen's width? ";
   cin >> number;                              // Get the width
   kitchen->setWidth(number);                   // Store in kitchen object

   // Get the bedroom dimensions.
   cout << "What is the bedroom's length? ";
   cin >> number;                              // Get the length
   bedroom->setLength(number);                  // Store in bedroom object
   cout << "What is the bedroom's width? ";
   cin >> number;                              // Get the width
   bedroom->setWidth(number);                   // Store in bedroom object
   // Get the den dimensions.
   cout << "What is the den's length? ";
   cin >> number;                              // Get the length
   den->setLength(number);                      // Store in den object
   cout << "What is the den's width? ";
   cin >> number;                              // Get the width
   den->setWidth(number);                       // Store in den object

   // Calculate the total area of the three rooms.
   totalArea = kitchen->getArea() + bedroom->getArea() + den->getArea();

   // Display the total area of the three rooms.
   cout << "The total area of the three rooms is " << totalArea << endl;

   // Delete the objects from memory.
   delete kitchen;
   delete bedroom;
   delete den;
   kitchen = 0;   // Make kitchen point to null.
   bedroom = 0;   // Make bedroom point to null.
   den = 0;       // Make den point to null.
   return 0;
}
```

Class Activity: Install GLUT, and compile and run a simple OpenGL program



http://www.math.ucsd.edu/~sbuss/MathCG/OpenGLsoft/SimpleLight/SimpleLight.html

SimpleLight illustrates the use Phong lighting in OpenGL. It draws six spheres of different colors, and makes a white light revolve around the spheres. It consists of two source files, **SimpleLight.c** and **SimpleLight.h.** There are several options available for downloading this program:

SimpleLightProject.zip: A zip file, including the source files, Microsoft Visual C++ workspace and project files, and Win32 executable.

SimpleLight.zip: A zip file, with sources files and the executable.

SimpleLight.c and SimpleLight.h and SimpleLight.exe: Download the two source files and the executable, one at a time.

Installing GLUT

If GLUT is not installed, you can install it by downloading the
glut zip file (v. 3.7.6) (web site) and copying its files as follows:

•runtime library:
C:\Program Files\Microsoft Visual Studio *\VC\bin\glut32.dll
["Program Files (x86)" for 64-bit Windows; The '*' matches your version of VS: 11.0 for VS2012, 10.0 for VS2010, 9.0 for VS2008]

•header file:
C:\Program Files\Microsoft Visual Studio *\VC\include\GL\glut.h
["Program Files (x86)" for 64-bit Windows; You have to create the "GL" directory]

•linker library:
C:\Program Files\Microsoft Visual Studio *\VC\lib\glut32.lib
["Program Files (x86)" for 64-bit Windows]

Managing keyboard input in GLUT

```
// glutKeyboardFunc is called below to set this function to handle
//                      all "normal" key presses.
void myKeyboardFunc( unsigned char key, int x, int y )
{
            switch ( key ) {
            case 'r':
                        RunMode = 1-RunMode;                // Toggle to opposite value
                        if ( RunMode==1 ) {
                                    glutPostRedisplay();
                        }
                        break;
            case 's':
                        RunMode = 1;
                        drawScene();
                        RunMode = 0;
                        break;
            case 'l':                                                   // Toggle local light mode
                        LightIsPositional = 1 - LightIsPositional;
                        if ( RunMode==0 ) {
                                    drawScene();
                        }
                        break;
            case 27:    // Escape key
                        exit(1);
            }
}
```

Overview of the main() function in GLUT

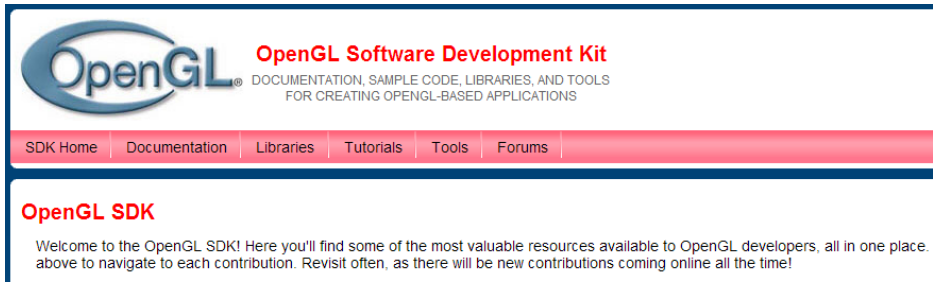
```
// Main routine
// Set up OpenGL, define the callbacks and start the main loop
int main( int argc, char** argv )
{
            glutInit(&argc,argv);
            // We're going to animate it, so double buffer
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH );
            // Window position (from top corner), and size (width% and hieght)
    glutInitWindowPosition( 10, 60 );
    glutInitWindowSize( 360, 360 );
    glutCreateWindow( "SimpleLight" );
            // Initialize OpenGL parameters.
    initRendering();
            // Set up callback functions for key presses
            glutKeyboardFunc( myKeyboardFunc );                    // Handles "normal" ascii symbols
            glutSpecialFunc( mySpecialKeyFunc );                   // Handles "special" keyboard keys
            // Set up the callback function for resizing windows
    glutReshapeFunc( resizeWindow );
            // Call this for background processing
            // glutIdleFunc( myIdleFunction );
            // Call this whenever window needs redrawing
    glutDisplayFunc( drawScene );
            fprintf(stdout, "Arrow keys control speed.  Press \"r\" to run,  \"s\" to single step.\n");

            // Start the main loop.  glutMainLoop never returns.
            glutMainLoop(  );
    return(0);          // This line is never reached.
}
```

3D and 2D graphics and OpenGL provide some nice alternatives to the console…

http://sourceforge.net/projects/ogl-samples/files/latest/download

http://www.cmake.org/files/v2.8/cmake-2.8.10.2-win32-x86.exe

```
$ /cygdrive/c/Program\ Files/CMake\ 2.8/bin/cmake.exe CMakeLists.txt
-- Building for: Visual Studio 11
-- The C compiler identification is MSVC 17.0.50727.1
-- The CXX compiler identification is MSVC 17.0.50727.1
-- Check for working C compiler using: Visual Studio 11
-- Check for working C compiler using: Visual Studio 11 -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler using: Visual Studio 11
-- Check for working CXX compiler using: Visual Studio 11 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found OpenGL: opengl32
-- Configuring done
-- Generating done
-- Build files have been written to: C:/russell/TurbineWarnerBros/ogl-samples-4.3.2.1/ogl-samples-4.3.2.1
```

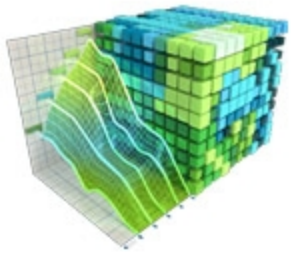


GPU Programming

https://developer.nvidia.com/nvidia-visual-profiler

NVIDIA Visual Profiler

The NVIDIA Visual Profiler is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. First introduced in 2008, Visual Profiler supports all 350 million+ CUDA capable NVIDIA GPUs shipped since 2006 on Linux, Mac OS X, and Windows. The NVIDIA Visual Profiler is available as part of the CUDA Toolkit.

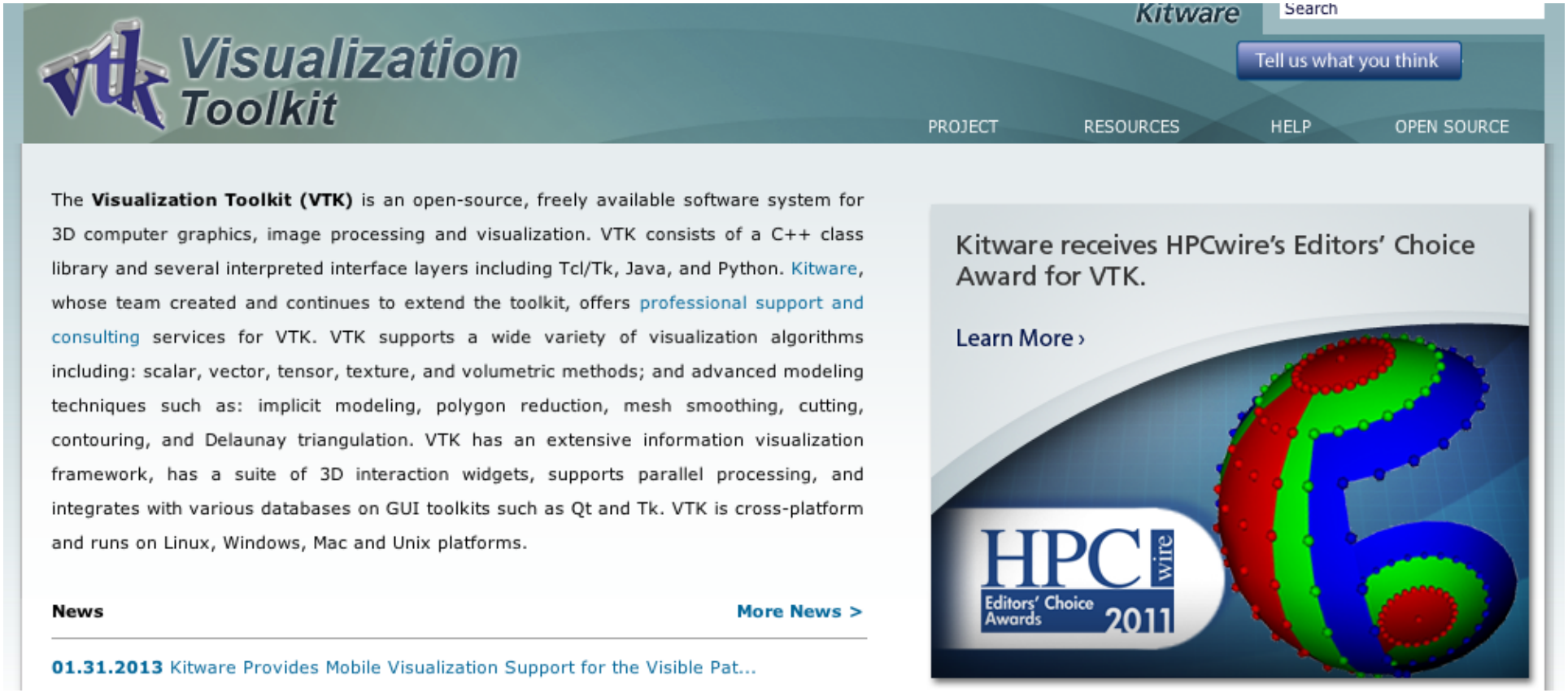


DOWNLOAD
CUDA 5
Production Release

Learn More
CUDA 5
Features and Benefits

Widget toolkits enable GUIs and user interfaces beyond the console

http://www.vtk.org/



wxWidgets

http://www.wxwidgets.org/

Homework #3 Exercises for next Monday (pick 2)

1)

19. Stock Profit

The profit from the sale of a stock can be calculated as follows:

$$\text{Profit} = ((NS \times SP) - SC) - ((NS \times PP) + PC)$$

where *NS* is the number of shares, *SP* is the sale price per share, *SC* is the sale commission paid, *PP* is the purchase price per share, and *PC* is the purchase commission paid. If the calculation yields a positive value, then the sale of the stock resulted in a profit. If the calculation yields a negative number, then the sale resulted in a loss.

Write a function that accepts as arguments the number of shares, the purchase price per share, the purchase commission paid, the sale price per share, and the sale commission paid. The function should return the profit (or loss) from the sale of stock.

Demonstrate the function in a program that asks the user to enter the necessary data and displays the amount of the profit or loss.

2)

49. An application uses a two-dimensional array defined as follows.

```
int days[29][5];
```

Write code that sums each row in the array and displays the results.

Write code that sums each column in the array and displays the results.

3)

Array Allocator

Write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should return a pointer to the array.

4)

True or False

31. T F Each byte of memory is assigned a unique address.
32. T F The * operator is used to get the address of a variable.
33. T F Pointer variables are designed to hold addresses.
34. T F The & symbol is called the indirection operator.
35. T F The & operator dereferences a pointer.
36. T F When the indirection operator is used with a pointer variable, you are actually working with the value the pointer is pointing to.
37. T F Array names cannot be dereferenced with the indirection operator.
38. T F When you add a value to a pointer, you are actually adding that number times the size of the data type referenced by the pointer.
39. T F The address operator is not needed to assign an array's address to a pointer.
40. T F You can change the address that an array name points to.
41. T F Any mathematical operation, including multiplication and division, may be performed on a pointer.
42. T F Pointers may be compared using the relational operators.
43. T F When used as function parameters, reference variables are much easier to work with than pointers.
44. T F The `new` operator dynamically allocates memory.
45. T F A pointer variable that has not been initialized is called a null pointer.
46. T F The address 0 is generally considered unusable.
47. T F In using a pointer with the `delete` operator, it is not necessary for the pointer to have been previously used with the `new` operator.

5) Implement three different pointer casts such as dynamic_cast, __try_cast, static_cast, reinterpret_cast, const_cast, C-Style Casts as described in
http://msdn.microsoft.com/en-us/library/aa712854(v=vs.71).aspx