|  |
| --- |
| C++ From the Fire Hose |
| An Introduction to C++ |
| Christopher Cantrell |

2012

Copyright © 2012 by Christopher Cantrell

All Rights Reserved

topherCantrell@gmail.com

Contents

[Day 1 1](#_Toc341936487)

[Creating a New Project in Visual C++ 2010 3](#_Toc341936488)

[Hello World 6](#_Toc341936489)

[Common VC++ Keyboard Shortcuts 6](#_Toc341936490)

[Functions (chapter 2) 7](#_Toc341936491)

[Includes (chapter 2) 9](#_Toc341936492)

[Parameters, Returns, and Substitution (chapter 2) 10](#_Toc341936493)

[Prototypes and Files (chapter 2) 11](#_Toc341936494)

[Namespaces (chapter 2) 12](#_Toc341936495)

[Variables and Expressions (chapter 3) 13](#_Toc341936496)

[Global Variables 14](#_Toc341936497)

[Local Variables 15](#_Toc341936498)

[Types and Sizes (chapter 3) 17](#_Toc341936499)

[Literals (chapter 3) 18](#_Toc341936500)

[Boolean Values (chapter 3) 19](#_Toc341936501)

[Operators (chapter 3) 20](#_Toc341936502)

[Flow (if/else) (chapter 3) 21](#_Toc341936503)

[Comparison Operators (evaluate to 1 or 0) 22](#_Toc341936504)

[Logic Operators (evaluate to 1 or 0) 22](#_Toc341936505)

[Short Circuit 22](#_Toc341936506)

[Complex Expressions 22](#_Toc341936507)

[Be Careful 22](#_Toc341936508)

[While and Do/While (chapter 3) 23](#_Toc341936509)

[Break and Continue (chapter 3) 23](#_Toc341936510)

[The (nearly) Infinite Loop 24](#_Toc341936511)

[Lab 1: HiLo 25](#_Toc341936512)

[Step 1 26](#_Toc341936513)

[Step 2 26](#_Toc341936514)

[Step 3 26](#_Toc341936515)

[Step 4 26](#_Toc341936516)

[Step 5 27](#_Toc341936517)

[Step 6 27](#_Toc341936518)

[Day 2 28](#_Toc341936519)

[Arrays (single dimension) (chapter 4) 29](#_Toc341936520)

[Pointers (chapter 4) 30](#_Toc341936521)

[Algorithm Primitives 32](#_Toc341936522)

[References 34](#_Toc341936523)

[Exercise 2: TicTacToe 37](#_Toc341936524)

[Step 1 and 2 38](#_Toc341936525)

[Step 3 38](#_Toc341936526)

[Step 4 39](#_Toc341936527)

[Step 5 39](#_Toc341936528)

[Step 6 39](#_Toc341936529)

[Final 40](#_Toc341936530)

# Day 1

**Objectives**

* Learn to create an empty project in VC++
* Learn to compile and run a “Hello World” program
* Learn to use includes
* Learn to declare local variables and pass parameters
* Learn to use prototypes
* Learn to use namespaces
* Learn to use primitive types
* Learn to use local and global variables
* Learn to use operators and expressions
* Learn about code flow (if,do,while,for,switch)

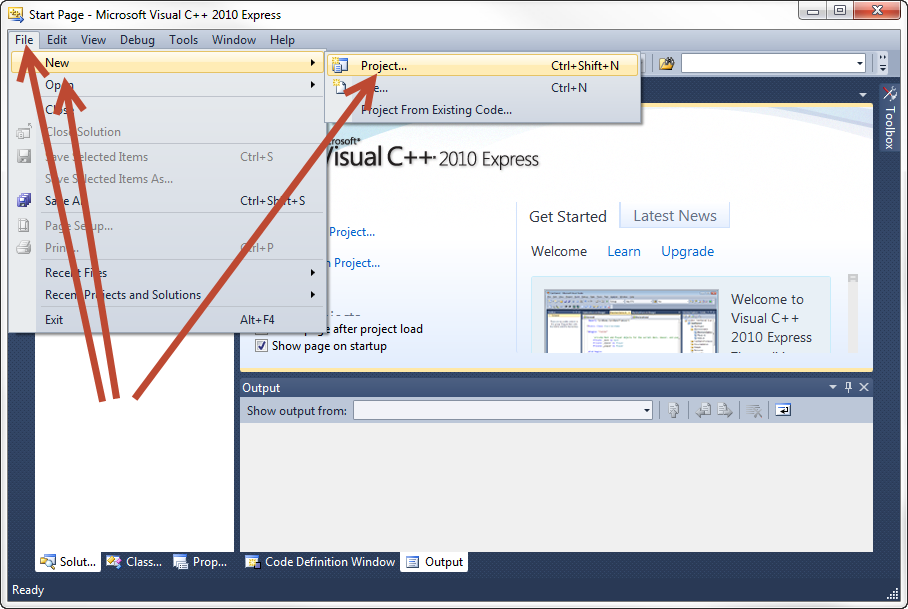
**Lab**

## Creating a New Project in Visual C++ 2010

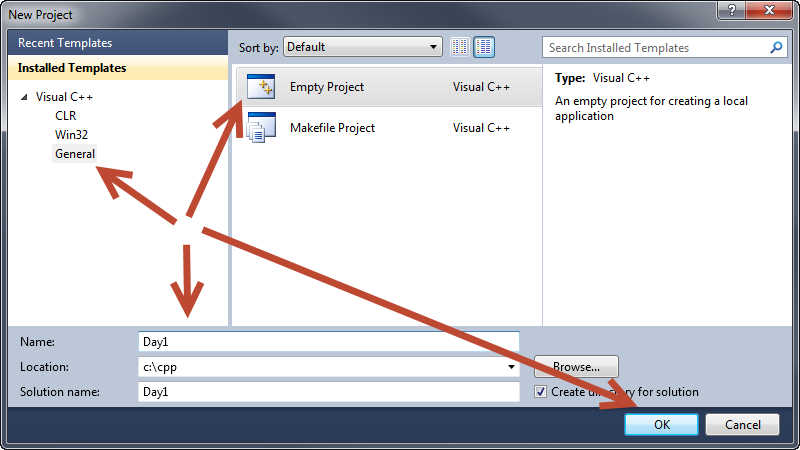
Launch Microsoft Visual C++ 2010 from the start menu or shortcut on the desktop.

Close the “start page” if it opens.

Select File -> New -> Project.



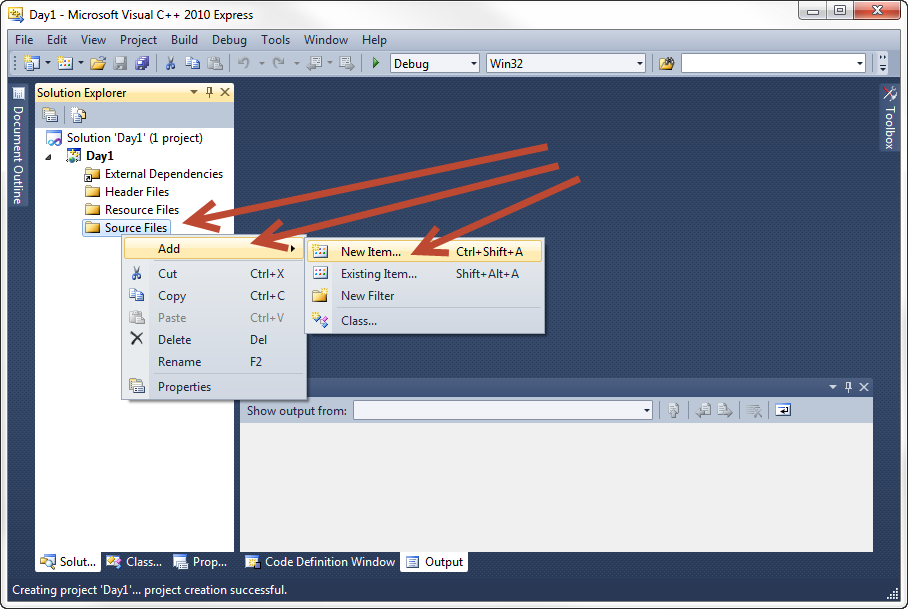
Use an “Empty Project” named “Day1” in the “c:\cpp” directory.

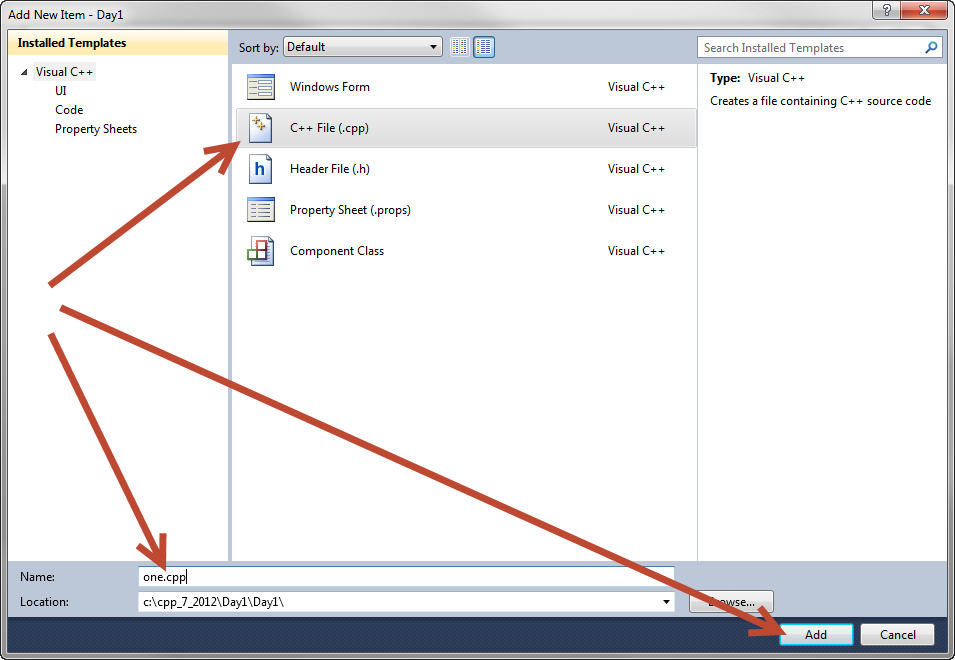


Add a “main” source file to the project. Right click on “Source Files”.

Source Files-> Add -> New Item.

Add a “C++ File (.cpp)” named “one.cpp”.

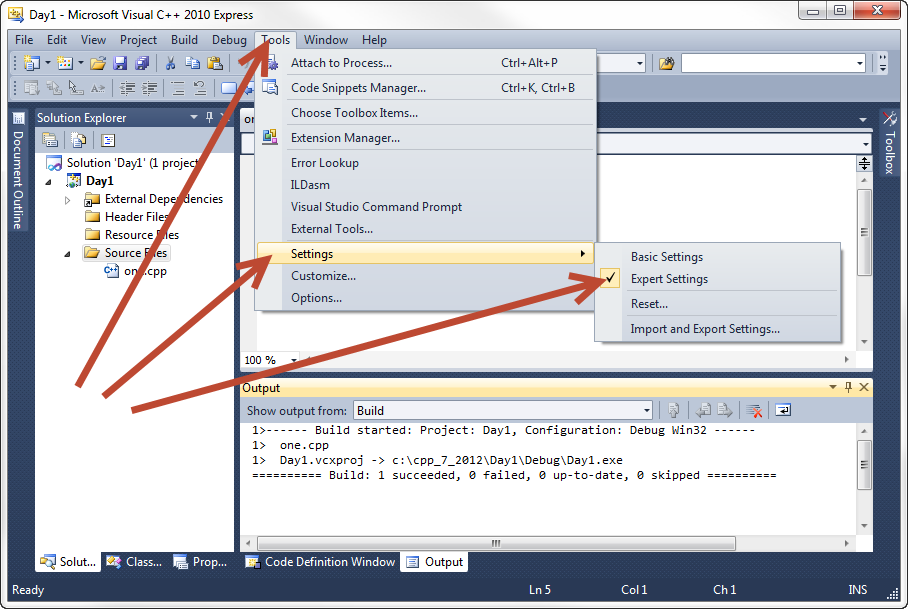


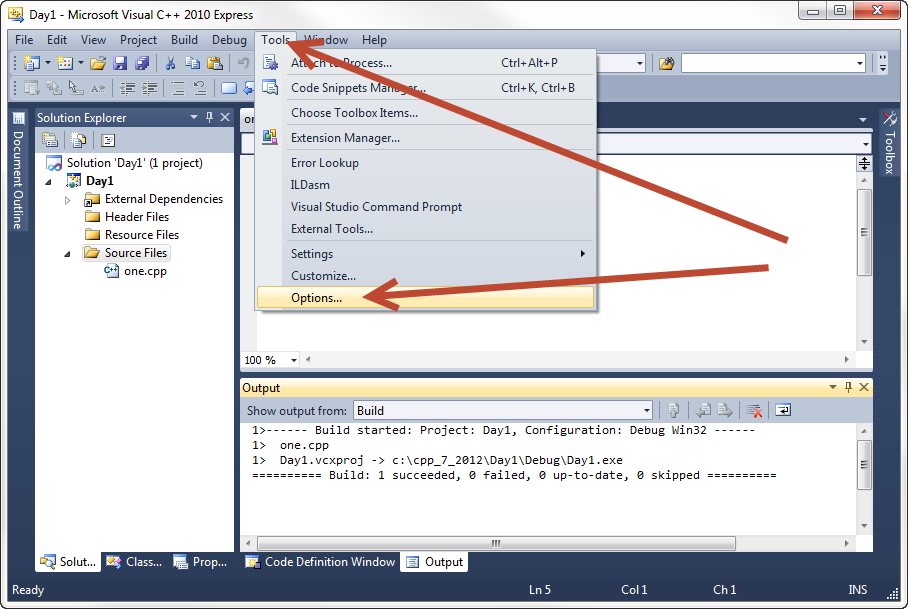


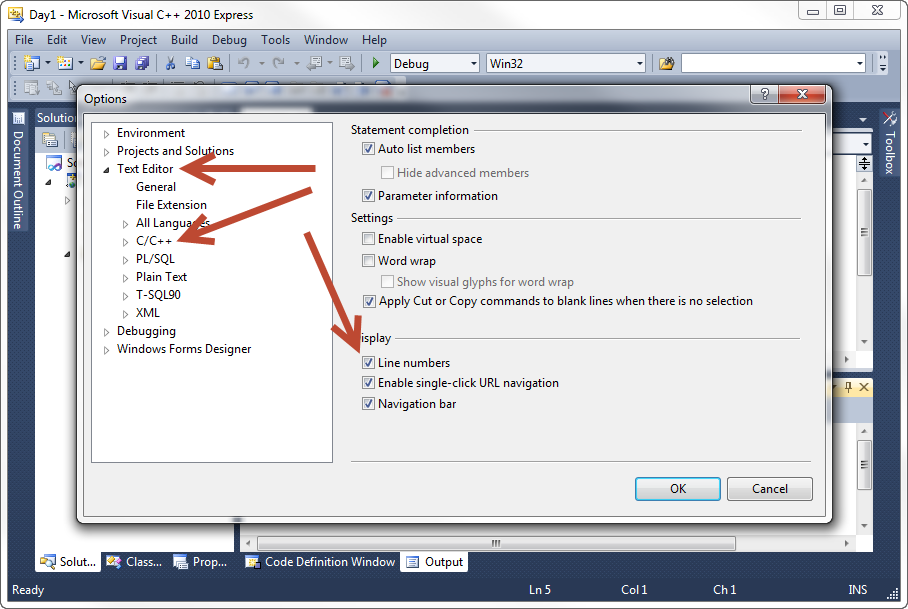
Turn on line numbering and expert settings if they are not already on:

Tools -> Settings -> Expert Settings (Only for Express users)

Tools-> Options -> Text Editor -> C/C++ -> Line Numbers



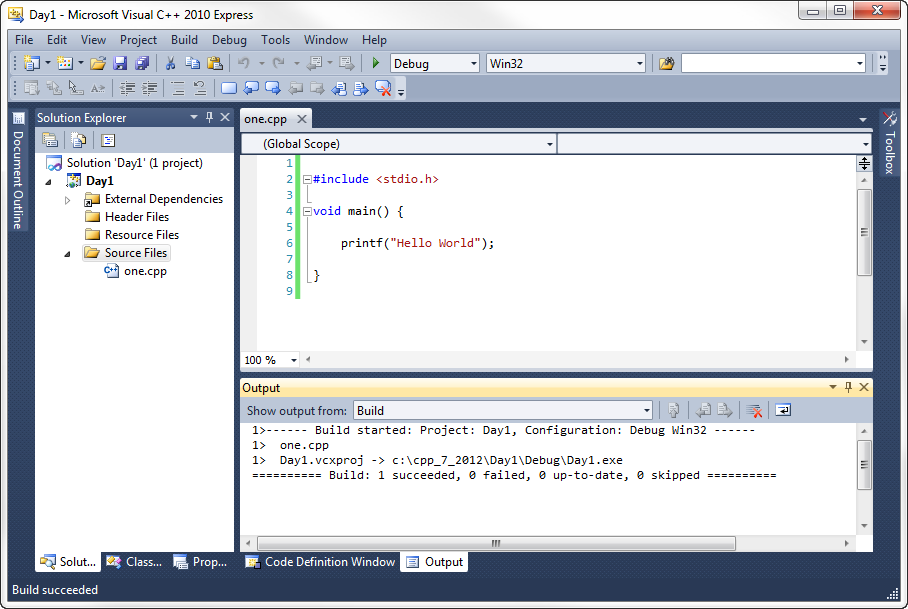




### Hello World

Type in the “Hello World” program into the “one.cpp” file.

Press F7 to build it or pick Build -> Build Solution.



### Common VC++ Keyboard Shortcuts

F7 Build

Shift-F5 Run without debugging

F5 Debug/continue

Shift-F5 Stop debugging

F10 Step over

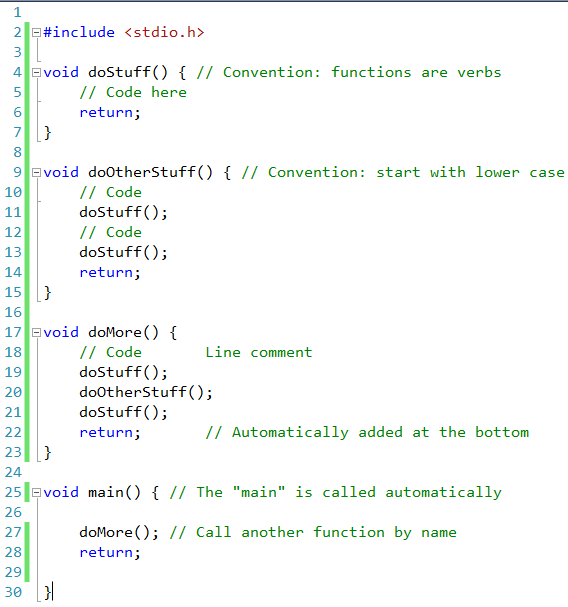
F11 Step into

Right-click on line to add/remove breakpoint

If you run the program the window will open and close too quickly to see. Instead, put a breakpoint on line 8 (the last line of the program) and press F5 to debug. Press F5 to continue to end the program.

## Functions (chapter 2)

The “function” is the basic unit of code for most languages. A “function” is a list of computer instructions that flow from top to bottom … first to last.



**Stack Pointer**

**Stack**

Return to 28

Return to 7

A function has a single return type (void here) and a name (like doStuff) and a list of parameters you pass to it (none here … the empty parentheses). Braces define the code block.

Spacing (and tabs and linefeeds) generally do not matter. The compiler ignores them.

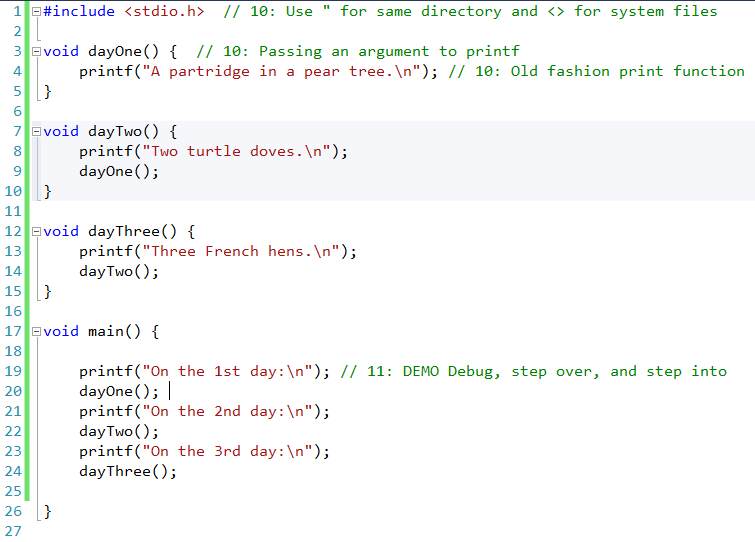
Convention: function names are verbs and start with lower case.

The “main” is the first function and is called automatically when your program starts.

There are “block” and “line” comments.

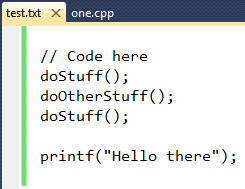
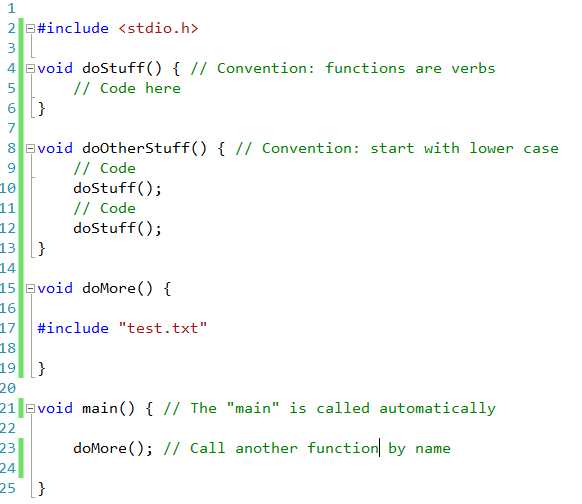
Run in debugger. Email “exe”. Launch with explorer.

Use this example to illustrate debugging.



## Includes (chapter 2)

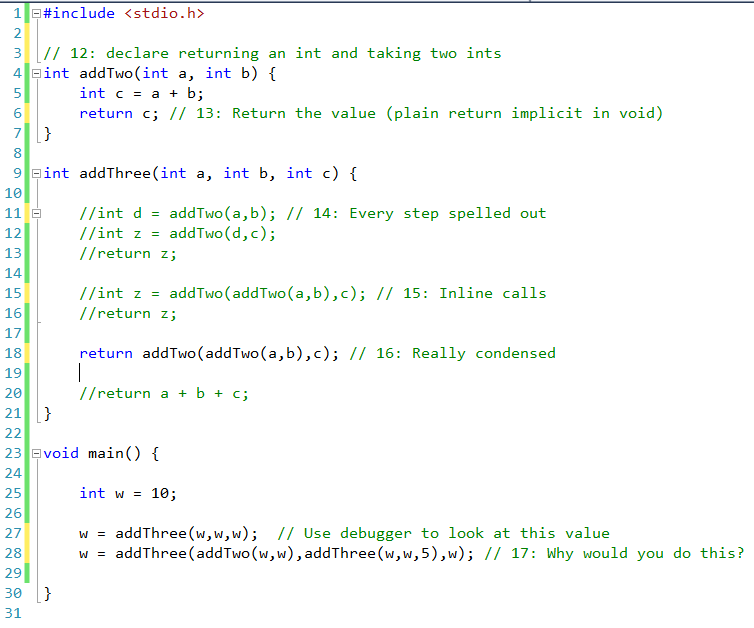
The “#include” is a pure text substitution. It is “pre-compile” step that blindly pastes the code without looking. There are several “pre-processor” directives that start with “#”.



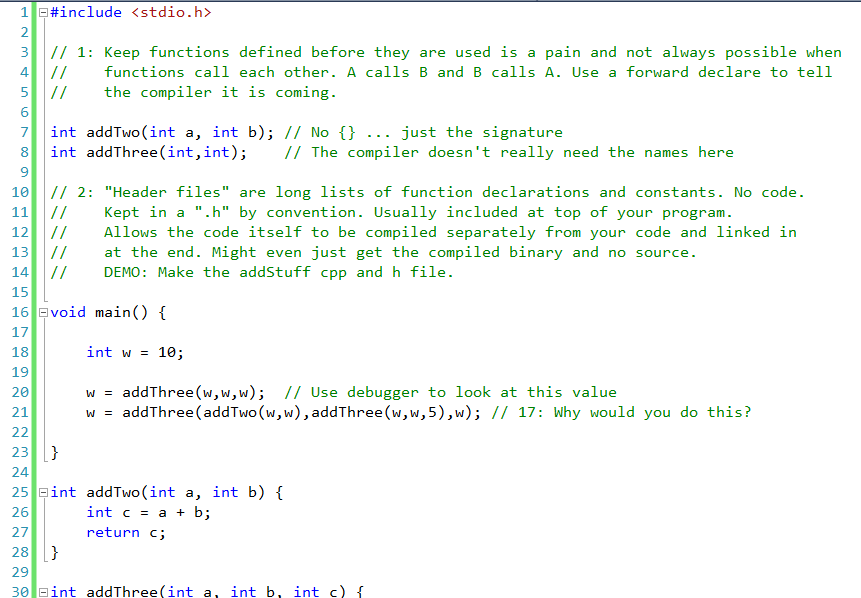
Using quotes means look for the file in the current directory. Using “<” means look for the file in the system library.

You can right-click and VC will take you to the file.

## Parameters, Returns, and Substitution (chapter 2)

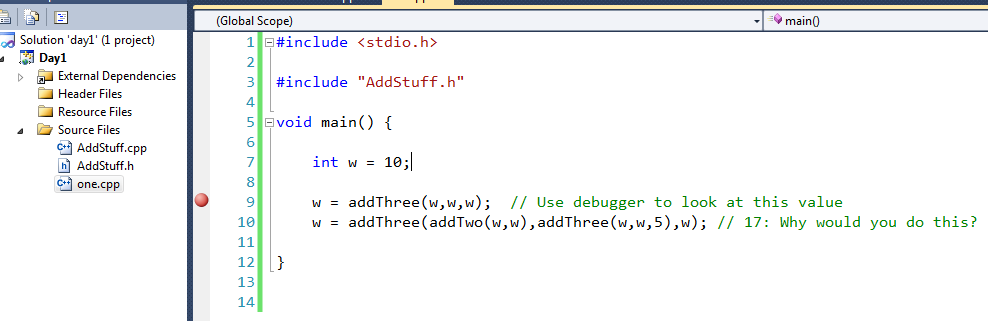


## Prototypes and Files (chapter 2)



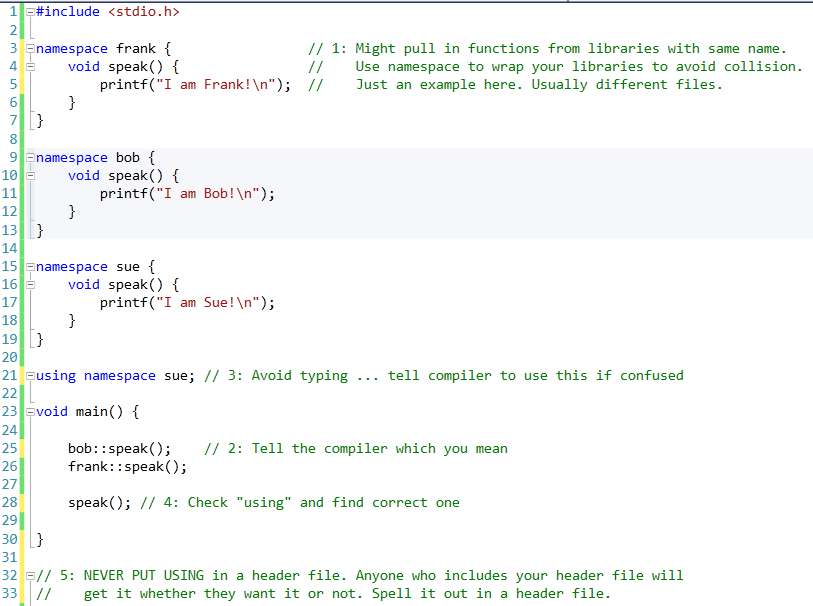
Include the .h in the .cpp if nothing else but to make sure they are in sync. If one changes and the other doesn’t then it breaks at compile.

Discuss recompiling only what is needed and buying binary libraries in detail.



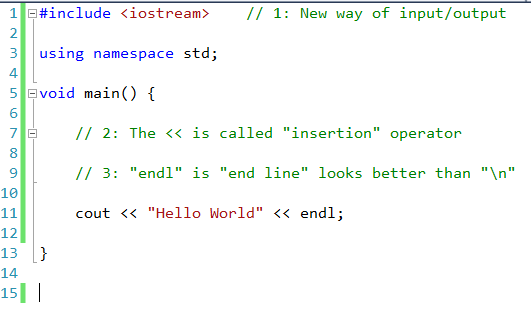
## Namespaces (chapter 2)

New with C++

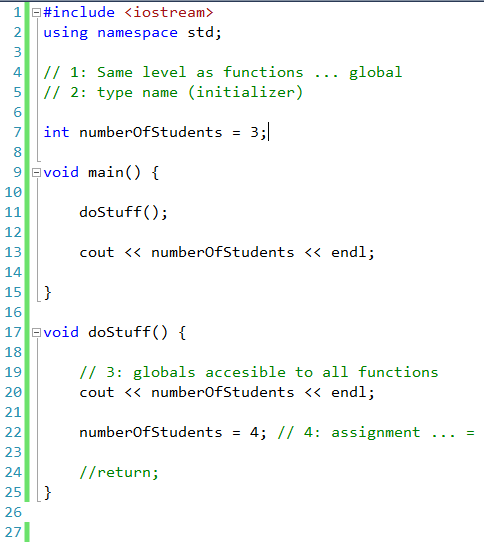


## Variables and Expressions (chapter 3)

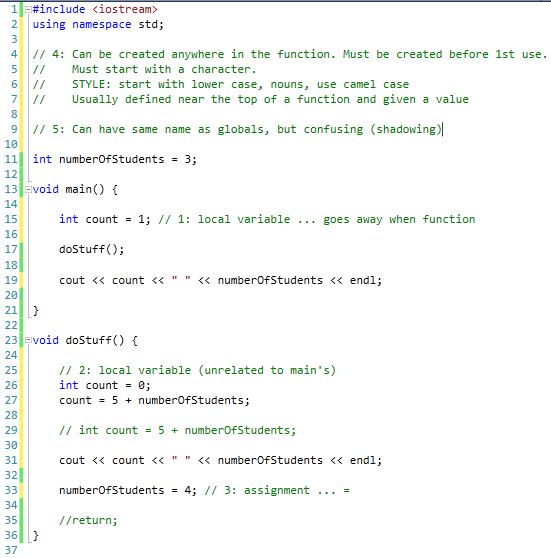
New way of input/output in C++. Uses operator overloading (in a few days).



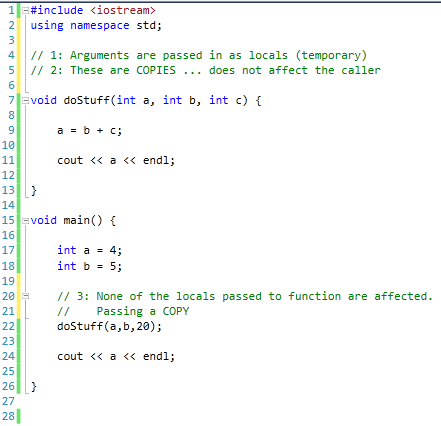
### Global Variables



### Local Variables



Local variables are on the stack. They “go away” when the function returns. Parameters are passed to functions as copies. The return value is a copy.



Return to DONE

a = 4

c = 20

b = 5

a = 4

Return to 23

b = 5

## Types and Sizes (chapter 3)

#include <iostream>

using namespace std;

#include <climits>

void main() {

// 1: Examples of types (page 72)

char a; // at least 8 bits (not unicode)

short b; // at least 16 bits

int c; // at least as big as a short

long d; // at least 32 bits and at least as big as an int

long long e; // at least 64 bits and at least as big as a long

float f; // unspecified (usually 32 bits)

double g; // unspecified (usually 64 bits)

long double h; // unspecified (usually 128 bits)

cout << sizeof(double) << endl; // 2: "sizeof" evaluated at compile time

cout << INT\_MAX << endl; // 3: Using constants from climits

}

The size of “int” is different for different machines. The size is what is “natural” for the CPU architecture. Your code might work on one machine but not another.

The upper bit of a value is its sign bit. You can use “unsigned” in front of a type to treat the value as unsigned.

When math overflows the storage size of a value it will wrap. Very negative numbers will suddenly be very positive.

#include <iostream>

using namespace std;

void main() {

unsigned int x = 0;

x = x - 1;

cout << x << endl;

// 0000

// - 0001

// ----

// 1111 (borrow is 1)

}

## Literals (chapter 3)

void main() {

float f = 1/3;

//float f = 1.0/3.0 1: 1 is integer, 1.0 is float

cout << f << endl; // 1: What gets printed here?

}

You can add a postfix to constants to tell the compiler how to treat them.

void main() {

int a = 123; // Nothing is "int"

long b = 123L; // L is "long", UL is "unsigned long"

float c = 3.14F; // F is "float"

double d = 2.0L; // L on a "float" is a double

char e = 'Z'; // In ticks ... character literal

char \*p = "HELLO"; // String constants

p = "HELLO \"WORLD"; // Quotes within quotes are "escaped" ... page 85

cout << p << endl;

}

The compiler will convert between types for you automatically, but data could be lost. The compiler will give you a warning unless you explicitly cast.

void doStuff(int x) {

cout << x << endl;

}

void main() {

float a = 1.0;

doStuff(a); // warning C4244: 'argument' : conversion from 'float' to 'int', possible loss of data

doStuff((int)a);

}

## Boolean Values (chapter 3)

C++ has a Boolean type that uses keywords “true” and “false”. Expressions still use the old “0 means false and anything else is true”.

#include <iostream>

using namespace std;

#include <climits>

void main() {

bool a = true;

bool b = false;

cout << a << " " << b << endl;

}

They are still treated as numerical (0 and 1) at runtime.

## Operators (chapter 3)

void main() {

int a = 2+3\*4-2;

// int a = 2+(3\*4)-2; // 1: Use parenthesis. They are free.

cout << a << endl;

}

Look at these in code:

+, -, \*, / Familiar math operations

% Modulo … remainder of integer division

&, |, ~, ^ Bitwise AND, OR, NOT, XOR

<<, >> Bitwise shifts

++, -- Increment and decrement

void main() {

int a = 4;

//int b = ++a;

int b = a++;

// a++ In a standalone the ++a and a++ are the same

cout << a << " " << b << endl;

}

## Flow (if/else) (chapter 3)

These are all forward-branches. The code flow progresses from top to bottom, but the if/else allows you to skip forward over instructions.

void main() {

int a; // 1: Skip initialization because ...

cin >> a; // 2: Read a value from the keyboard

// 3: if syntax. Keyword, expression, block of code

if(a == 4) {

cout << "a is 4" << endl;

}

cout << "Done" << endl;

}

Optional “else”:

if(a == 4) {

cout << "a is 4" << endl;

} else {

cout << "a is NOT 4" << endl;

}

You can chain them like this:

if(a == 4) {

cout << "a is 4" << endl;

} else if(a==5) {

cout << "a is 5" << endl;

} else {

cout << "a is something else" << endl;

}

Without the braces the if/else works with the next statement.

int b;

int c=1;

if(a == 4) b=1;

else b=2;c=0;

But use the braces. In this case “c” is always set to 0.

### Comparison Operators (evaluate to 1 or 0)

// == operator evaluates to 1 if things are the same or 0 if not

// != evaluates to 1 if things are different

// <, > less-than, greater-than

// <=, >= less-than-equal, greater-than-equal

### Logic Operators (evaluate to 1 or 0)

if(a==1 && b==2) {}

if(a==0 || b==0 || c==3) {

}

### Short Circuit

The language definition guarantees that expressions are evaluated left to right and short circuited (true or false) as soon as possible. In the above, if a==0 then b==0 and c==3 are never checked.

This is important for null pointer checks (see later).

### Complex Expressions

if(!(a==4 || a==5)) {} // The "!" is a NOT.

if( (doSomething(2)==4) && !(a==5 && b==6) ) {}

### Be Careful

int a=1;

if(a=0) {

cout << "a is 1" << endl;

}

## While and Do/While (chapter 3)

Loops allow you to jump the code flow backwards … to go back and repeat a block of instructions. Use braces to define the block of code and an expression to control the looping.

int a=0;

while(a<=5) {

++a; // Start example without the ++

cout << "Here" << endl;

}

The expression is evaluated at the “top” of the loop (first thing). If it evaluates to non-zero then the block of code executes. At the bottom the loop comes back to the expression at the top.

The “do/while” moves the expression check to the bottom of the loop.

int a=0;

do {

++a;

cout << a << endl;

} while (a<=5);

The “do” loop is ALWAYS executed at least once. The WHILE loop may never execute at all.

We usually want a “do” loop but almost always use a “while” loop. We like to put the controlling expression at the top where we see it first and think about it first.

## Break and Continue (chapter 3)

The “break” keyword takes the CPU to the next instruction after the bottom of the loop. It “breaks” the CPU out of the loop. The “continue” keyword takes the CPU to the top of the loop.

int a=0;

while(a<10) {

cout << a << endl;

++a;

if(a==5) {

break;

}

}

int a=0;

while(a<10) {

++a;

if(a==5) {

continue;

}

cout << a << endl;

}

### The (nearly) Infinite Loop

while(true) {

int a;

cin >> a;

cout << "You entered " << a << endl;

if(a<0) {

break;

}

}

## Lab 1: HiLo

Write a program that picks a random number from 1 to 100 and prompts the player to guess it. With each guess the program prints “higher” or “lower” allowing the user to home in on the secret number.

Use these two lines to generate a random number from 1 to 100:

srand(time(0));

int number = (rand()%100)+1;

You must include <time.h>:

#include <time.h>

Work this first exercise in small pieces adding functionality little by little.

Suggestions:

1. Start with a file “hilo.cpp” with a “main” that prints HelloWorld.
2. Add local variables in “main” for “number” and “guess”.
3. Make an “if” check in “main” to compare the two numbers and print “higher” or “lower” or “equal”. Initialize the two variables with test values and test each “if” case.
4. Add “cin” to get the “guess”. Print a prompt before getting the number. Test the cases again.
5. Put the whole input/output code in an infinite loop. Add a “break” statement in the “equals” case.
6. Add the code fragment to make “number” random.
7. Spruce up the game if you want with some of the following:
   1. Add a “Guess count” to keep score.
   2. Add random “higher” and “lower” wording.
   3. Print directions at the start of the game.

### Step 1

#include <iostream>

using namespace std;

void main() {

cout << "Hello World" << endl;

}

### Step 2

void main() {

int number = 20;

int guess = 1;

cout << "Hello World" << endl;

}

### Step 3

void main() {

int number = 20;

int guess = 1;

if(guess<number) {

cout << "Higher!" << endl;

}

if(guess>number) {

cout << "Lower!" << endl;

}

if(guess==number) {

cout << "You got it!" << endl;

}

}

### Step 4

int number = 20;

int guess;

cout << "Your guess: ";

cin >> guess;

if(guess<number) {

cout << "Higher!" << endl;

}

...

### Step 5

while(true) {

cout << "Your guess: ";

cin >> guess;

if(guess<number) {

cout << "Higher!" << endl;

}

...

}

### Step 6

#include <iostream>

#include <time.h>

using namespace std;

void main() {

srand(time(0));

int number = (rand()%100)+1;

int guess;

while(true) {

cout << "Your guess: ";

cin >> guess;

if(guess<number) {

cout << "Higher!" << endl;

}

if(guess>number) {

cout << "Lower!" << endl;

}

if(guess==number) {

cout << "You got it!" << endl;

break;

}

}

}

# Day 2

**Objectives**

* Learn to use arrays
* Learn to use pointers
* Learn to use references

**Lab**

* Tic Tac Toe

## Arrays (single dimension) (chapter 4)

void main() {

int counter; // create one int named "counter" in local memory (stack)

int stuff[20]; // create 20 ints in local memory (stack)

//int stuff[counter]; Must be created with constant at compile time. Other ways

// to do dynamic

stuff[0] = 1; // : 0 is the "index"

stuff[1] = 2;

stuff[20] = 20;

cout << stuff[0] << " " << stuff[1] << " " << stuff[20] << endl;

cout << stuff[-1] << endl;

// :Unitialized stuff ... whatever was in memory

// :The size is NOT kept in the data structure. The compiler does not check.

// :If you are lucky it will write to an invalid memory location and your program will die.

// :Likely you'll change another piece of data. Difficult to debug.

system("pause");

}

int stuff[6] = {1,2,3,1,4,1}; // Can be initialized like other vars

//int stuff[] = {1,2,3,4}; // Let the compiler count for you

// You can only use the {} initializer at creation time.

//

//int more[5];

//more = {1,2,3,4,5}; does not work

// Write a program to count the 1s in the array (

int count = 0;

int x = 0; // Start with first element

while(x<5) { // Elements 0 through 5 NOT 6, right??

if(stuff[x]==1) {

count = count + 1;

}

x=x+1;

}

// Show as a for loop

cout << count << endl;

## Pointers (chapter 4)

There is one other built in type called a “pointer” that holds a numeric memory address.

You can read and write from the address “pointed” to by the pointer. But you need to know how many bytes are at the address. You might want to read an “int” or a “long long” or something else.

There is no way to know for sure what is in memory, but you can tell the COMPILER what you THINK is at that location and it will generate the code to move the right number of bytes for you.

void main() {

int a = 5; // Reserve storage on stack. Name "a". Put 5 in it.

// Concept:

//pointer p = 0x1234; // Reserve storage on stack named "p" for an address and put 01234 in it

// What is at 0x1234? No way to check and see. But you tell the compiler what you think

//int\* p = 0x1234; // Reserve storage on stack named "p" for an address and put 01234 in it

// The compiler will treat "p" as the address of an "int" ... not a "float" or "char" etc

// When you "chase" or "dereference" the pointer it will move the correct number of bytes

// You program runs in different memory every time. The value "0x1234" is not something

// useful. Instead you can get the address of other things:

int\* p = &a; // The "&" is the "address of" variable a. This is COMPILE TIME.

cout << a << endl;

cout << p << endl; // This is the ADDRESS ... not what is at the address

int b = \*p; // This is the "chase" or "dereference" operator.

// Don't use the address ... use what the address points to (and we told you it was an "int")

cout << b << endl;

system("pause");

}

void main() {

int a = 5;

int\*p = &a;

int \*p = &a; // Which do you like?

int\* p = &a;

int \* p = &a;

system("pause");

}

void main() {

int a = 5;

int\* p = &a;

\*p = 20; // Change what "p" points to (we said p was an int pointer)

cout << a << endl;

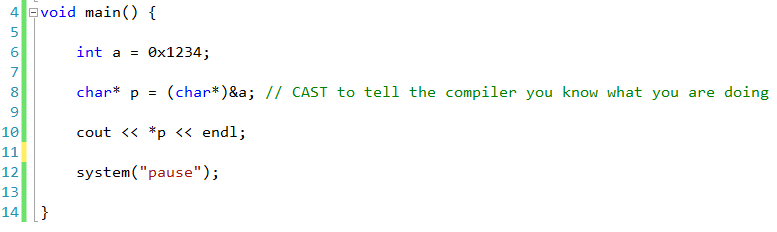
int\* q = p; // Create local storage named "q" and copy address from p to q

\*q = 30;

cout << a << endl;

system("pause");

}



4

3

2

1

8000

a

0

0

0

8

p

8004

int data[] = {7,8,8,3,1,2,6,4};

int\* p = &data[0]; // Address of 1st element

cout << \*p << endl;

p = p + 1; // Magic pointer math! Adds 4 to address (you said int)

cout << \*p << endl;

cout << \*(p+2) << endl; // Looks odd

cout << p[2] << endl; // But this is the same

p = data; // Without the [] you get the address-of (1st element)

// Mechanically, arrays and pointers are the same thing

## Algorithm Primitives

Now that we have arrays we can look at some basic algorithms and looping over arrays.

2

1

1

4

1

// What is being passed here? Not all 6 ints ... the function is

// changing the values in what is passed.

void sortArray(int data[], int size) {

bool changed;

do {

changed = false;

for(int x=0;x<5;++x) {

int a = data[x];

int b = data[x+1];

if(b<a) {

data[x] = b;

data[x+1] = a;

changed = true;

}

}

} while(changed);

}

void printArray(int data[], int size) {

for(int x=0;x<size;++x) {

cout << data[x];

if(x!=(size-1)) {

cout << ", ";

}

}

cout << endl;

}

void main() {

int data[] = {3,2,1,1,3,1};

sortArray(data,6);

printArray(data,6);

/\*

//for(int w=0;w<1000;++w) {

for(int x=0;x<5;++x) {

int a = data[x];

int b = data[x+1];

if(b<a) {

data[x] = b;

data[x+1] = a;

}

}

//}

cout << data[0] << "," << data[1] << "," << data[2] << ","

<< data[3] << "," << data[4] << "," << data[5] << ","

<< endl;

\*/

system("pause");

}

## References

void swap(int a, int b) {

// Incoming parameters are copied

int c = a;

a = b;

b = /\*a\*/ c;

}

void main() {

int a=1;

int b=2;

swap(a,b); // Does not change my locals

cout << a << " " << b << endl;

system("pause");

}

How to fix this?

void swap2(int\* a, int\* b) {

int c = \*a;

\*a = \*b;

\*b = /\* \*a \*/ c;

}

swap2(a,b); // What do we do here?

swap2(&a,&b); // Very obvious that my locals are in danger

You rarely use the pointer without dereferencing it. You almost always have the “\*” with it. If you do a lot of pointer work you quickly tire of the syntax.

When would you not? Keep these in mind for later.

* Pointer math (p = p + 1)
* Array access (p[23]) which is really (\*p+23) pointer math
* Initializing (p = &a)

New to C++ is a way to tell the compiler that a variable should always be treated as an address.

void swap3(int& a, int& b) {

// The compiler knows a and b are addresses and treats them

// differently from "c". You have to keep it in mind without

// the visual clue.

int c = a;

a = b;

b = /\* a \*/ c;

}

swap3(a,b); // Compiler does the “address of”. Not obvious at all!

But you shouldn’t call a function without knowing the signature anyway. You can look at the header file and tell if it address-of or copy.

This is for your convenience (or confusion) the disassembly is identical.

int a=4;

int\* p; // What value is here?

int& q = a; //You must initialize when you create

q[0] = 4; // Can't use index (arrays are passed by REFERENCE anyway)

q=q+1; // Can't use pointer math

int doCube(int a) {

a = a\*a\*a;

cout << a << endl;

return a;

}

void main() {

int a=4;

int b;

b = doCube(a);

cout << "Cube of " << a << " is " << b << endl;

system("pause");

}

Add the “&” to the function and rerun. The function is changing the incoming parameter, which isn’t a copy.

int doCube(const int& a) // "const"

The “const” means you promise not to change “a”. The compiler won’t let you. You, the caller, feel safer. But you aren’t:

int doCube(const int& a) {

int& b = (int &)a;

b = b\*b\*b;

cout << b << endl;

return b;

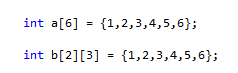
}

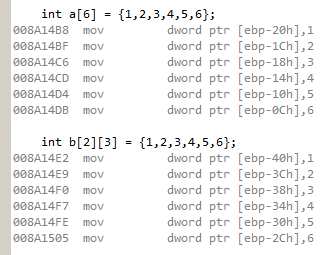
The “const” is just a helper at compile time. It does nothing at the runtime and does not contribute to the assembly. But USE IT. Let the compiler find what it can!

## Arrays (multi-dimensional)

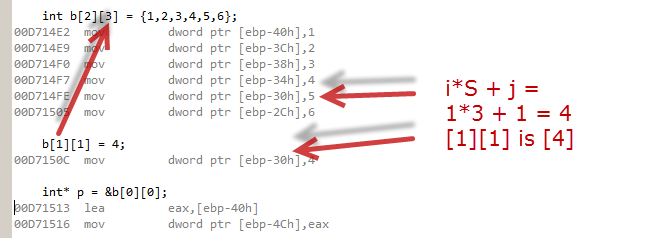
You can learn a lot by looking under the covers at the assembly. While debugging, right click on the code and select “go to disassembly”.

Multi-dimensional arrays are stored as flat (one dimensional) array. The compiler uses the sizes to do the offset math for you.





In the generated code, a multi-dimensional array is flattened to a single dimension array. The compiler generates the offset-math for you.



// You can use braces to organize the values, but this is

// just visual. You still have to tell the compiler the

// sizes in the []

// I J

int c[2][3] = {{1,2,3},{4,5,6}};

// i j

c[1][1] = 4;

// The math to get the offset into the flat array is:

// pos = i\*J + j

// Notice that the compiler doesn't need the value I to

// do the math. You can leave off the first size since

// the compiler doesn't need it

int d[][3] = {{1,2,3},{4,5,6}};

Arrays are passed as pointers. A multidimensional array is passed as a flat array pointer. You have to tell the compiler the sizes to do the math at the receiving function.

void printArray(int data[2][3]) {

// The compiler needs the "3" to do the i\*J+j

// The "2" is ignored. You can leave it out.

}

## Exercise 2: TicTacToe

Write a two-player tic-tac-toe program. Prompt the players one at a time for their move (a number 1 through 9 for squares left to right and top to bottom). Print the board after each move. Make sure moves are legal and detect a winning move and congratulate the winner!

Suggestions:

1. Use a global variable array of 9 ints to represent the 9 squares left to right and top to bottom. Initialize the array to all 0’s. The values of the squares are the states of the squares:
   1. 0 means empty
   2. 1 means X
   3. 2 means Y
2. Create a “printBoard” function that prints the state of the board on the screen. Use “|” and “-“ to make an ascii-art image of the screen. Use a helper function to translate 0,1, and 2 to SPACE, X, and Y.
3. Create a “getLegalMove” function that asks the player for a square number and returns the move if the square is valid. Otherwise the function prints an error and asks until the move is legal.
4. Add an infinite loop to main and ask player 1 for a move and then player 2 for a move. Set the state of the board array after every move.
5. Create a “checkForWin” function that returns true if the board is in a “win” state and false if not. You can be as fancy as you like here or make a brute-force 16-if routine.
6. Add check-for-win after every move.

### Step 1 and 2

#include <iostream>

using namespace std;

int board[] = {0,0,0, 0,0,0, 0,0,0};

char getPrintCharacter(int contents) {

if(contents==0) return ' ';

if(contents==1) return 'X';

if(contents==2) return 'O';

return '?';

}

void printBoard() {

cout << getPrintCharacter(board[0]) << " | "

<< getPrintCharacter(board[1]) << " | "

<< getPrintCharacter(board[2]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(board[3]) << " | "

<< getPrintCharacter(board[4]) << " | "

<< getPrintCharacter(board[5]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(board[6]) << " | "

<< getPrintCharacter(board[7]) << " | "

<< getPrintCharacter(board[8]) << endl;

}

void main() {

printBoard();

system("pause");

}

### Step 3

int getLegalMove() {

int move;

while(true) {

cout << "Pick a square: ";

cin >> move;

if(move>=1 && move<=9 && board[move]==0) {

return move-1; // Return 0 to 8

}

cout << "Invalid move" << endl;

}

}

### Step 4

void main() {

while(true) {

cout << endl;

printBoard();

cout << "Your move player 1 (X)." << endl;

int m = getLegalMove();

board[m] = 1; // 1 for X

cout << endl;

printBoard();

cout << "Your move player 2 (O)." << endl;

m = getLegalMove();

board[m] = 2; // 2 for O

}

system("pause");

}

### Step 5

bool checkForWin() {

for(int p=1;p<3;++p) {

// Horizontal

if(board[0]==p && board[1]==p && board[2]==p) return true;

if(board[3]==p && board[4]==p && board[5]==p) return true;

if(board[6]==p && board[7]==p && board[8]==p) return true;

// Vertical

if(board[0]==p && board[3]==p && board[6]==p) return true;

if(board[1]==p && board[4]==p && board[7]==p) return true;

if(board[2]==p && board[5]==p && board[8]==p) return true;

// Diagonal

if(board[0]==p && board[4]==p && board[8]==p) return true;

if(board[2]==p && board[4]==p && board[6]==p) return true;

}

return false;

}

### Step 6

cout << endl;

printBoard();

cout << "Your move player 1 (X)." << endl;

int m = getLegalMove();

board[m] = 1; // 1 for X

if(checkForWin()) {

cout << endl << "X wins!" << endl << endl;

break;

}

### Final

#include <iostream>

using namespace std;

// Use a flat array for the board. 0=empty, 1=X, 2=O

int board[] = {0,0,0, 0,0,0, 0,0,0};

/\*\*

\* This fuction maps a board-square state to a printed

\* character.

\* @param contents the contents of the square

\* @return the character representation to print

\*/

char getPrintCharacter(int contents) {

if(contents==0) return ' ';

if(contents==1) return 'X';

if(contents==2) return 'O';

return '?';

}

/\*\*

\* This function prints the tic-tac-toe game board on the screen

\* in ASCII art.

\*/

void printBoard() {

cout << getPrintCharacter(board[0]) << " | "

<< getPrintCharacter(board[1]) << " | "

<< getPrintCharacter(board[2]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(board[3]) << " | "

<< getPrintCharacter(board[4]) << " | "

<< getPrintCharacter(board[5]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(board[6]) << " | "

<< getPrintCharacter(board[7]) << " | "

<< getPrintCharacter(board[8]) << endl;

}

/\*\*

\* This function gets a legal move from the player. The function prints an

\* error and loops until the player enters a valid move.

\* @return the move 0-8

\*/

int getLegalMove() {

int move;

while(true) {

cout << "Pick a square: ";

cin >> move;

// TODO more detailed reporting here:

// - The move is not valid (must be 1-8)

// - The square is taken

if(move>=1 && move<=9 && board[move-1]==0) {

return move-1; // Return 0 to 8

}

cout << "Invalid move" << endl;

}

}

/\*\*

\* This method checks the game board for a win configuration. If

\* the board is a "win" the last player to move just won.

\* @return true if won or false if not

\*/

bool checkForWin() {

for(int p=1;p<3;++p) { // Check player 1 then player 2

// Horizontal

if(board[0]==p && board[1]==p && board[2]==p) return true;

if(board[3]==p && board[4]==p && board[5]==p) return true;

if(board[6]==p && board[7]==p && board[8]==p) return true;

// Vertical

if(board[0]==p && board[3]==p && board[6]==p) return true;

if(board[1]==p && board[4]==p && board[7]==p) return true;

if(board[2]==p && board[5]==p && board[8]==p) return true;

// Diagonal

if(board[0]==p && board[4]==p && board[8]==p) return true;

if(board[2]==p && board[4]==p && board[6]==p) return true;

}

return false;

}

void main() {

while(true) {

cout << endl;

printBoard();

cout << "Your move player 1 (X)." << endl;

int m = getLegalMove();

board[m] = 1; // 1 for X

if(checkForWin()) {

cout << endl << "X wins!" << endl << endl;

break;

}

cout << endl;

printBoard();

cout << "Your move player 2 (O)." << endl;

m = getLegalMove();

board[m] = 2; // 2 for O

if(checkForWin()) {

cout << endl << "O wins!" << endl << endl;

break;

}

}

printBoard();

system("pause");

}

# Day 3

**Objectives**

* Learn to make your own classes from primitives
* Learn the C++ mechanics: methods, this, and permissions
* Learn to use constructors and destructors
* Learn to build new types by inheritance

There is a lot of material here. Likely it will roll over into Day 4.

**Lab**

* Tic Tac Toe as objects

## Composition

You can create new types from primitive types. The compiler treats the new types just like built-ins.

#include <iostream>

using namespace std;

// This defines the memory foot-print (the structure) of a

// complex type. It does NOT create a new "instance" of the type.

struct Point {

int x; // = 0 You can't initialize because this is just the

int y; // structure ... not an instance

}; // Note semicolon required here

void main() {

int a = 4; // Create an "int" on the stack (4 bytes)

Point p; // Create a "Point" on the stack (8 bytes)

cout << sizeof(p) << endl;

p.x = 10; // Use the "dot" membership operator to get to pieces

p.y = 20;

cout << "X " << p.x << " Y " << p.y << endl;

system("pause");

}

Show Line using Point. We say the Line “has-a” point in it.

## Passing Parameters

Passing new types by value and pointer.

void initialize(Point p, int a, int b) { // 8-byte copy of p

p.x = a;

p.y = b;

}

void initialize2(Point \*p, int a, int b) {

(\*p).x = a; // This is clunky

p->x = a; // New pointer-membership operator

p->y = b;

}

void main() {

Point p;

initialize(p,10,20); // 8-byte copy of p

initialize2(&p,10,20); // 4-byte address of p

cout << "X " << p.x << " Y " << p.y << endl;

system("pause");

}

You could use references for “initialize2” and get the same code with less syntax.

The compiler treats your type like a built in. It will copy byte by byte.

int a; // Create storage

int b = a; // Create storage and copy 4 bytes

Point p; // Create storage

Point q = p; // Create storage and copy 8 bytes

## Behavior

Unlike the primitive types, a complex type usually has a set of functions that go with it to perform complex behavior.

These functions all take a pointer to the instance of the structure.

struct Point {

int x;

int y;

};

void initialize(Point \*p, int a, int b) {

p->x = a;

p->y = b;

}

void print(Point \*p) {

cout << "X " << p->x << " Y " << p->y << endl;

}

int getSumOfCoordinates(Point \*p) {

return p->x + p->y;

}

void main() {

Point p;

initialize(&p,10,20);

print(&p);

cout << getSumOfCoordinates(&p) << endl;

system("pause");

}

## Syntactic Sugar

C++ provides a compiler trick to make this relationship easier to define. You simple move the functions inside the structure.

The compiler adds (and hides) the pointer for you. It names the pointer variable “this”.

struct Point {

int x;

int y;

// The pointer is ALWAYS there and named "this"

void initialize(/\* Point \*this, \*/ int a, int b) {

this->x = a;

this->y = b;

}

// The compiler will insert "this->" automatically

void print(/\* Point \*this \*/) {

cout << "X " << x << " Y " << y << endl;

}

// We call this "functions with automatic this" a "method"

int getSumOfCoordinates(/\* Point \*this \*/) {

return x + y;

}

};

void main() {

Point p;

// New calling syntax (produces same code as before)

// You are thinking about calling a function "on" an instance

// Now we say "p" is an object of type "Point". The object

// has behaviors we can call.

p.initialize(10,20);

p.print();

cout << p.getSumOfCoordinates() << endl;

system("pause");

}

This syntax produces the exact same runtime code.

This is a new way to express what C programmers discovered: complex data and behavior go together to make “objects”.

You think in terms of asking an object to do something and not in terms of calling a function and passing a pointer to the data.

It is the “object” that is doing something and not the function.

There is still just one “copy” of the function though. The memory footprint of an object is ONLY its data.

## Encapsulation

The C programmers also discovered that the data itself should be private. OO programming is all about telling objects to do things and not dipping into the structure directly.

I want to hide the details of what is inside me. I want to hide my private parts. Users should only use my methods and never my data directly.

C++ adds the “private” and “public” keywords:

struct Point {

private:

int x;

int y;

public:

void initialize(int a, int b) {

x = a;

y = b;

}

void print() {

cout << "X " << x << " Y " << y << endl;

}

int getSumOfCoordinates() {

return x + y;

}

};

Point p;

p.x = 4; // Compile time error

If I want to allow changing the data I need to provide “accessors”:

int getX() {

return x;

}

void setX(int a) {

x = a;

}

Is it better to use these than to make X public? Why or why not?

* Disadvantage: slower runtime performance
* Advantage: can change details in the future without breaking code

You need a crystal ball. If you never change the details then the performance is wasted. There is intuition here. You may sacrifice performance today for time-to-market tomorrow.

The “inline” keyword suggests that the compiler expand the code in place without making a function call. In the future when the code changes and you recompile the compiler would use other inline code or not inline at all. You get the performance and future flexibility at the same time.

inline int getX() {

return x;

}

This is just a suggestion! The compiler is free to ignore it. The optimizer is also free to inline things you haven’t suggested.

## Class

By default everything in a “struct” is PUBLIC, though now you can add the “private” as you wish.

Usually you want “private” to be the default and add “public” as you wish.

The “structDefaultPrivate” keyword is klunky. Instead you use “class”

In C++ the difference between “struct” and “class” is that “class” is default “private” if you don’t do anything.

You can replace the word “class” in your code with “struct” and it will compile to the same runtime. You can replace “struct” with “class” too as long as you add “public:” to the top.

class Point {

int x; // "private" by default

int y;

public:

void initialize(int a, int b) {

x = a;

y = b;

}

void print() {

cout << "X " << x << " Y " << y << endl;

}

int getSumOfCoordinates() {

return x + y;

}

};

## Constructors and Destructors

What if you wanted to make the Point unchangeable? What could you do?

You want to initialize it at creation but then make it read only.

Creation is a very special time in an object’s lifecycle.

The compiler gives you a special method to call at creation:

Point(int a, int b) { // Same name as class, no return type

cout << "I am here in constructor" << endl;

x = a;

y = b;

}

Point p(1,2); // Automatically calls constructor

Point q; // Doesn't compile. You MUST use a constructor

You can provide many constructors to create objects in different ways:

Point() {

x=0;

y=0;

}

Point(int a) {

x = a;

y = a;

}

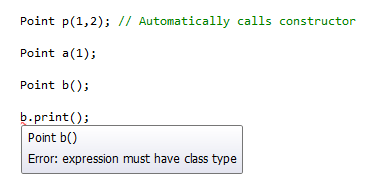
Point(int a, int b) { // Same name as class, no return type

cout << "I am here in constructor" << endl;

x = a;

y = b;

}



This is a gotcha.

The compiler thinks that “Point b();” is a function prototype! To use the “no args” constructor you must leave off the parenthesis.



Maybe change the language design so that function prototypes cannot be placed in a function?

You ALWAYS have a constructor. If you don’t code one then you get the “do nothing no args” generated for you. If you specify any constructor then you don’t get the auto-generated.

The compiler calls a special method when your object is destroyed. There is only one form:

~Point() { // Same name with "~" in front. No return. No arguments.

cout << "In destructor" << endl;

}

You ALWAYS have a destructor. If you don’t code one you get the “do nothing” generated for you.

void doSomething(Point q) {

// Code here

}

void demo() {

Point p(1,2);

doSomething(p);

}

void main() {

//Point p(1,2); // Automatically calls constructor

demo();

system("pause");

// "p" is destroyed HERE at the end

}

The call to “doSomething” creates a new object. How? There was no constructor call?

## Copy Constructor

The compiler uses a special constructor to make a copy.

// More commonly Point(const Point &other)

Point(Point &other) {

cout << "Making a copy" << endl;

x = other.x;

y = other.y;

}

If you do not code a copy constructor you get the “memory copy” for free. It just copies memory. If you have pointers then both objects point to the same places, which may or may not be right.

You also get an automatic function for assignment (q = p). We will look at that later.

## Inheritance

Want to add a color value to Point. Want to pick up everything in Point and just add more.

You want to “extend” the definition of Point.

class ColorPoint : public Point { // Syntax for "inheritance" ... for "extending"

int color;

};

void main() {

ColorPoint p; // Put print in base constructor

p.print();

system("pause");

}

You get all the data and functions of “Point” for free. In fact, you cannot prevent it. You can’t take anything away. You simply add to it. “Point” is the base class. ColorPoint is the derived class.

class ColorPoint : public Point { // Syntax for "inheritance" ... for "extending"

int color;

public:

int getColor() {

return color;

}

};

void main() {

ColorPoint p; // Put print in base constructor

p.print();

cout << p.getColor() << endl;

system("pause");

}

## Inherited Constructors and Destructors

Notice that the base class constructor is called here. Your constructor will ALWAYS call the base class constructor. If you don’t specify one, you’ll get a call to the no-args of the base class.

If it doesn’t have one, it won’t compile.

// You must call the base-class constructor before anything else happens. Even

// before this function begins to execute. C++ adds the "Initializer" list

// for this:

ColorPoint(int a, int b, int c) : Point(a,b) // Pick a base constructor ... any one

{

color = c;

}

// You can optionally put everything in the initializer list. For primitives

// it doesn't matter. For object members it saves time. For "const" members

// it is required.

ColorPoint(int a, int b, int c) : Point(a,b) , color(c)

{

}

The last thing your destructor does is call the baseclass destructor. There is just one, and the code is automatic. You do NOT get to put it in manually.

## Inherited Members with the Same Name

class ColorPoint : public Point {

int color;

public:

ColorPoint(int a, int b, int c) : Point(a,b) , color(c)

{

}

// Same name as the baseclass. That's OK. It's like a namespace.

void print() {

Point::print(); // Calling the baseclass print

cout << "And I have color " << color << endl;

}

};

The compiler has no trouble keeping these separate.

ColorPoint p(1,2,10); // Put print in base constructor

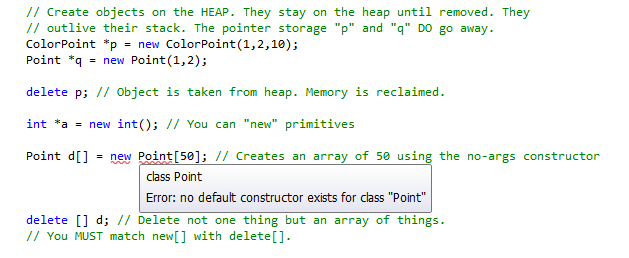
Point q(3,4);

p.print(); // Compiler knows to call ColorPoint::print

q.print(); // Compiler knows to call Point::print

p.Point::print(); // You can tell the compiler which you mean

## Dynamic Memory (Likely Day 4)



If you forget to delete something then you “leak” the memory.

## Separate files

Usually objects are defined in their own CPP and HEADER files. Show the “Point” example.

Explain #ifdef

## Exercise 3: TicTacToe In Objects

Write a two-player tic-tac-toe program. Prompt the players one at a time for their move (a number 1 through 9 for squares left to right and top to bottom). Print the board after each move. Make sure moves are legal and detect a winning move and congratulate the winner!

Use objects for the Board and Player. Their interfaces should be:

class Board {

public:

/\*\*

\* This method clears the board in preparation for a new game.

\*/

void clear();

/\*\*

\* This method returns the contents of the given cell.

\* @param cellNumber the cell number [0..8]

\* @return the contents: 0 (empty), 1 (player1), 2 (player2)

\*/

int getCell(int cellNumber);

/\*\*

\* This method registers a players move on the board and returns

\* true if the move was a win.

\* @param cellNumber the cell number [0..8]

\* @return true if the move was a win

\*/

bool setCell(int cellNumber, bool isPlayer1);

/\*\*

\* This method prints the board to the screen in ASCII art.

\*/

void print();

};

class Player {

public:

/\*\*

\* This method identifies the player as Player1 or Player2

\* @param isPlayer1 true if player1 or false if player2

\*/

void setPlayerNumber(bool isPlayer1);

/\*\*

\* This method gets the players next valid move.

\* @param board the game board for move checking

\* @return the player's input [0..8]

\*/

int getMove(Board& board);

/\*\*

\* This method prints the "you won" message.

\*/

void printWin();

};

Suggestions:

1. Create “Board.h”, “Player.h”, “Board.cpp”, “Player.cpp”, and “Game.cpp” files. Use #ifdefs in the headers and #includes in the cp files.
2. Type in the given interfaces in the header files. Make stub methods in the cpp files and put print statements in them. Put test code in main to call all the object methods to test.
3. Flesh out the Board methods and data using the code from last time. Use test code in main to set some cells and print the board.
4. Flesh out the Player methods and data using the code from last time. The “playerNumber” is used to control the wording in prompts and messages.
5. Flesh out the game loop in main.

### Board.h

#ifndef BOARD\_H\_

#define BOARD\_H\_

class Board {

int cells[9];

// Helper functions

char Board::getPrintCharacter(int contents);

bool checkForWin();

public:

/\*\*

\* This method clears the board in preparation for a new game.

\*/

void clear();

/\*\*

\* This method returns the contents of the given cell.

\* @param cellNumber the cell number [0..8]

\* @return the contents: 0 (empty), 1 (player1), 2 (player2)

\*/

int getCell(int cellNumber);

/\*\*

\* This method registers a players move on the board and returns

\* true if the move was a win.

\* @param cellNumber the cell number [0..8]

\* @return true if the move was a win

\*/

bool setCell(int cellNumber, bool isPlayer1);

/\*\*

\* This method prints the board to the screen in ASCII art.

\*/

void print();

};

#endif

### Board.cpp

#include "Board.h"

#include <iostream>

using namespace std;

void Board::clear() {

for(int x=0;x<9;++x) {

cells[x] = 0;

}

}

int Board::getCell(int cellNumber) {

return cells[cellNumber];

}

bool Board::setCell(int cellNumber, bool isPlayer1) {

if(isPlayer1) {

cells[cellNumber] = 1;

} else {

cells[cellNumber] = 2;

}

return checkForWin();

}

char Board::getPrintCharacter(int contents) {

if(contents==0) return ' ';

if(contents==1) return 'X';

if(contents==2) return 'O';

return '?';

}

void Board::print() {

cout << getPrintCharacter(cells[0]) << " | "

<< getPrintCharacter(cells[1]) << " | "

<< getPrintCharacter(cells[2]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(cells[3]) << " | "

<< getPrintCharacter(cells[4]) << " | "

<< getPrintCharacter(cells[5]) << endl;

cout << "---------" << endl;

cout << getPrintCharacter(cells[6]) << " | "

<< getPrintCharacter(cells[7]) << " | "

<< getPrintCharacter(cells[8]) << endl;

}

bool Board::checkForWin() {

for(int p=1;p<3;++p) {

// Horizontal

if(cells[0]==p && cells[1]==p && cells[2]==p) return true;

if(cells[3]==p && cells[4]==p && cells[5]==p) return true;

if(cells[6]==p && cells[7]==p && cells[8]==p) return true;

// Vertical

if(cells[0]==p && cells[3]==p && cells[6]==p) return true;

if(cells[1]==p && cells[4]==p && cells[7]==p) return true;

if(cells[2]==p && cells[5]==p && cells[8]==p) return true;

// Diagonal

if(cells[0]==p && cells[4]==p && cells[8]==p) return true;

if(cells[2]==p && cells[4]==p && cells[6]==p) return true;

}

return false;

}

### Player.h

#ifndef PLAYER\_H\_

#define PLAYER\_H\_

#include "Board.h"

class Player {

bool player;

public:

/\*\*

\* This method identifies the player as Player1 or Player2

\* @param isPlayer1 true if player1 or false if player2

\*/

void setPlayerNumber(bool isPlayer1);

/\*\*

\* This method gets the players next valid move.

\* @param board the game board for move checking

\* @return the player's input [0..8]

\*/

int getMove(Board& board);

/\*\*

\* This method prints the "you won" message.

\*/

void printWin();

};

#endif

### Player.cpp

#include "Player.h"

#include <iostream>

using namespace std;

void Player::setPlayerNumber(bool isPlayer1) {

player = isPlayer1;

}

int Player::getMove(Board& board) {

int move;

while(true) {

if(player) {

cout << "Pick a square Player 1: ";

} else {

cout << "Pick a square Player 2: ";

}

cin >> move;

if(move>=1 && move<=9 && board.getCell(move-1)==0) {

return move-1; // Return 0 to 8

}

cout << "Invalid move" << endl;

}

}

void Player::printWin() {

if(player) {

cout << "You win player 1!" << endl;

} else {

cout << "You win player 1!" << endl;

}

}

### Game.cpp

#include <iostream>

using namespace std;

#include "Board.h"

#include "Player.h"

void main() {

Board board;

Player player1;

Player player2;

// TODO: The Board constructor should do this first time

board.clear();

// TODO: Would be better to pass to constructor

player1.setPlayerNumber(true);

player2.setPlayerNumber(false);

while(true) {

board.print();

int move = player1.getMove(board);

bool win = board.setCell(move,true); // TODO: Use player.isPlayer1()

if(win) {

player1.printWin();

break;

}

// TODO: Use an array of 2 players instead of separately

board.print();

move = player2.getMove(board);

win = board.setCell(move,false); // Use player.isPlayer2() instead

if(win) {

player2.printWin();

break;

}

}

system("pause");

}

# Day 4

**Objectives**

* Learn to use function overloads
  + Global (function)
  + Class members
* Learn to use operator overloads
  + Operators in a class (== and =)
  + Global functions (<< and >>)
* Learn to use streams and files
  + Opening a file
  + Reading and writing text (primitive types)
  + Reading and writing binary
  + Stream functions
* Learn to use the std::string object
  + Basic substitutions and searches
  + Overview of all methods

**Lab**

* MadLibs

## Function Overloads

You can’t have two functions with the same name. The compiler doesn’t know how to pick between them.

The C++ compiler is smarter. Functions can have the same name but different arguments and it can pick among them based on how you call them.

int add(int a, int b) {

return a+b;

}

int add(int a, int b, int c) {

return a+b+c;

}

void main() {

int c = add(1,2);

int d = add(1,2,3);

system("pause");

}

The return-type does NOT count since you aren’t required to use the return type when you call a function. Only the name and the arguments contribute.

These functions are said to be “overloaded”.

You can overload methods too.

The compiler will make a best guess. If you think it might guess wrong then explicitly cast:

int c = add( (int)1, (int)2 );

## Default Parameter Values

C++ adds “default” values.

int add(int a, int b, int c=0, int d=0, int e=0) {

return a+b+c+d+e;

}

int c = add(1,2);

int d = add(2,3,4);

Defaults go on the right and must all go together.

## Operator Overloads (functions)

class Point {

public:

int x;

int y;

Point(int a, int b) {

x = a;

y = b;

}

};

Point addPoints(Point &a, Point &b) {

Point ret(0,0);

ret.x = a.x + b.x;

ret.y = a.y + b.y;

return ret;

}

void printPoint(Point &a) {

cout << "X:" << a.x <<" Y:" << a.y <<endl;

}

void main() {

Point i(2,3);

Point j(4,5);

i = addPoints(i,j);

printPoint(i);

system("pause");

}

Wouldn’t it be nice if you could tell the compiler to use “addPoints(i,j)” when it sees “i + j” ?

You can! You have to name you add function in a way it knows to look for:

Point operator+(Point &a, Point &b) {

Point ret(0,0);

ret.x = a.x + b.x;

ret.y = a.y + b.y;

return ret;

}

Point i(2,3);

Point j(4,5);

i = operator+(i,j);

printPoint(i);

When the compiler sees an operator on types it doesn’t know about it looks for an “operator+” function that matches and replaces it:

“left N right” becomes “operatorN(left,right)”:

Point i(2,3);

Point j(4,5);

i = i + j;

i = j + j + i + j;

printPoint(i);

The order left/right DOES matter (hint … think about subtraction):

Point operator\*(int s, Point&b) {

Point ret(0,0);

ret.x = b.x\*s;

ret.y = b.y\*s;

return ret;

}

i = 4\*i;

i = i\*4; // Does not compile

## Stream operators

Return now to:

int a = 2;

a = a << 2; // operator << on 2 integer ... bit shift

cout << a; // operator << on left=ostream and right=int

operator<<(cout,a); // Produces this code

There are many overloaded functions for “operator<<(ostream, …)” for built in types.

You can write your own for Point:

void operator<<(ostream &os, Point &a) { // POINT by reference or stack? You pick!

os << "X:" << a.x << " Y:" << a.y << endl;

}

void main() {

Point i(2,3);

Point j(4,5);

cout << i;

system("pause");

}

Point i(2,3);

Point j(4,5);

cout << i;

operator<<(cout,i); // Compiler does this

cout << i << j; // Does not compile

operator( operator<<(cout,i) , j); // Becomes this

But our operator<< function doesn’t return anything. We need to return the “cout” so that it can pass through to the next call in the chain:

ostream& operator<<(ostream &os, Point &a) { // POINT by reference or stack? You pick!

os << "X:" << a.x << " Y:" << a.y << endl;

return os;

}

## Operator Overloads (class members)

This looks like the stuff we were doing when we moved the functions inside the class:

class Point {

public:

int x;

int y;

Point(int a, int b) {

x = a;

y = b;

}

};

Point operator+(Point &a, Point &b) {

Point ret(0,0);

ret.x = a.x + b.x;

ret.y = a.y + b.y;

return ret;

}

Point operator\*(int s, Point&b) {

Point ret(0,0);

ret.x = b.x\*s;

ret.y = b.y\*s;

return ret;

}

class Point {

public:

int x;

int y;

Point(int a, int b) {

x = a;

y = b;

}

Point operator+( Point &b) {

Point ret(0,0);

ret.x = x + b.x;

ret.y = y + b.y;

return ret;

}

Point operator\*(int s) {

Point ret(0,0);

ret.x = x\*s;

ret.y = y\*s;

return ret;

}

};

Point i(2,3);

Point j(4,5);

i = i + j;

i = operator+(i,j); // This STILL exists. Remember the automatic "this".

// The compiler looks for "+" in the left argument

You can’t put the “operator<<” inside Point. The left argument is the object called. You would have to modify “cout” to include your Point, which you can’t.

These “operator overloads” are not very object-oriented. In an OO environment you would tell the Point object to do the work:

i.toString();

i.toStream(cout);

i.addTo(j);

Permissions might be a problem. We can solve them using a “friend”:

class Point {

friend void operator<<(ostream&, Point&);

friend class Line; // Line methods can now access Point privates

int x;

int y;

public:

};

void operator<<(ostream& os, Point& p) {

os << p.x << " " << p.y;

}

## Streams and Files

A “stream” is a stream of bytes flowing in (as in CIN) or out (as in COUT). The stream operators “<<” and “>>” push the builtins to and from bytes as ascii.

These are TEXT processors … to and from TEXT.

int a = 12; // An "int" is 4 bytes

cout << a; // TWO bytes are sent here: '1' and '2'.

There are methods you can do to all byte streams.

There are lots of sources of byte streams. You have interacted with these before:

* Keyboard Stream of bytes to read from
* Monitor Stream of bytes to write to
* Socket TWO streams … one to read and one to write
* File TWO streams … one to read and one to write

Once you know how to get hold of a particular stream you can talk to anything using these common operations.

Streams you write to “<<” are “ostream”. Streams you write to “>>” are “istream”. The specific types of stream inherit from these.

A file stream, for instance, has all of “ostream” plus extra methods to open and close the stream.

## Getting a File Stream

#include <iostream>

#include <fstream>

using namespace std;

int main() {

fstream someFile; // Local object manages the stream

someFile.open("c:\\cpp\_7\_2012\\test.txt", fstream::out);

int a = 12;

//cout << a;

someFile << a << " is a value " << endl << a << endl;

someFile.flush(); // Make sure nothing is in the buffer

someFile.close(); // Close the file

system("pause");

}

Create a stream … use it … then close it.

someFile.open("c:\\cpp\_7\_2012\\test.txt",

fstream::out | fstream::in);

Or the mode constants together:

* in Allow input operations on the stream
* out Allow output operations
* app Set position to end before EACH output operation
* ate Set the position to the end when opening
* binary Consider the stream as binary rather than text
* trunc Discard any existing content when opening

You can set the stream to use EXCEPTIONS when something bad happens. But by default you have to check the fail status:

if(!someFile.is\_open()) { // someFile.fail()

cout << "Could not open the file" << endl;

}

### ostream

// There are several of these modifier functions:

// endl ... print endl

// hex ... switch output format to hexadecimal

// oct ... octal

// dec ... back to decimal

int a = 100;

cout << a << hex << a << oct << a << dec << a << endl;

endl is a function!

endl(cout);

What is “cout << endl;” ??

cout.operator<<(endl);

operator<<(&cout,endl); --> this function calls the function you pass passing cout

// http://www.cplusplus.com/reference/iostream/manipulators/

// There are methods you can call on cout to control text formatting:

cout.width(12); // Next output is 12 characters at least. Then reset width for next output.

cout.fill('\*'); // Set the padding character

cout << 1 << endl; // \*\*\*\*\*\*\*\*\*\*1

// Control of the floating point format

cout.setf(ios::scientific, ios::floatfield);

cout.precision(3); // Digits after the decimal point here

cout << 1.2 << endl;

// Binary in the form of pointer to bytes and a number of bytes.

char\* p = new char[1024];

// Fill up the data

cout.write(p,1024);

Point pnt;

cout.write((char \*)&pnt,sizeof(Point));

### istream

int a;

cin >> hex >> a;

If you type in “-123Z45” then “-123” is read to a. Stops at invalid character, which is next to be read.

char c;

cin.get(ch); // gets a single character

char p[1024];

cin.getline(p,20); // Read a line up to CR but not more than 20

We will discuss “string” shortly … there is a helper function to read a string from a stream with no length requirement.

cin.read(char\*,int size)

Seeking around:

<http://www.cplusplus.com/reference/iostream/istream/seekg/>

## Thoughts on Reading Data

These methods require you know exactly what is coming next in the stream. Text is often dynamic.

* Line-at-a-time into a string and use string methods to parse the string
* XML … use a library to read the file into a model (Document Object Model … DOM)
* Binary … warning! Sizes, endianess, and padding changes. Look at bytes and convert.

## std::string

The compiler continues to support null terminated strings.

char\*p = “Hello”; // 6 bytes … letters plus a ZERO on the end to mark the end

C string library to work with these.

Have to keep up with the zero on the end and remember when to delete the pointer.

Common errors: overflowing by one byte because of the terminator.

If there is time, do an exercise where you make a string class.

void main() {

string one = "Hello World"; // Constructor that takes a char\*

string two = one; // Show where things live

one[0] = 'P'; // Operator overload to change characters

cout << one << endl;

cout << two << endl; // Two "P" didn't change. This is a copy.

int len = one.size(); // Get the length

// Operator+ overload for appending and assignement

// Space is created ... old space deleted.

one = one + two + two;

if(one == two) {

// == is overloaded to do the string compare

}

int i = one.find('o'); // First 'o'

int j = one.find('o',5); // First 'o' starting with one[5]

string sub = one.substr(6,3); // Starting index and count ... not end index

system("pause");

}

void main() {

string arr[4]; // Create FOUR objects using their default constructor

arr[1] = "One";

arr[3] = "Last";

for(int x=0;x<4;++x) {

cout << ":" << arr[x] << ":" << endl;

}

system("pause");

}

void main() {

// Some string processing

string text = "I am a %replaceMe% test.";

string rep = "rep";

int i = text.find('%');

int j = text.find('%',i+1);

cout << i << " " << j << endl;

int repLength = j-i+1; // Why plus one?

string firstPart = text.substr(0,i);

string secondPart = text.substr(j+1);

text = firstPart + rep + secondPart;

cout << text << endl;

system("pause");

}

include <iostream>

#include <string>

#include <sstream>

using namespace std;

//void main(int argumentCount, char\*\* arguments) { // usually argc, argv

void main(int argumentCount, char\* arguments[]) {

for(int x=0;x<argumentCount;++x) {

cout << arguments[x] << endl;

}

// First is the name of the program

int a = atoi(arguments[1]); // For C strings

string one = "1234";

int b = atoi(one.c\_str());

// You already know a text to int converter .. the ">>" with int.

// Need a way to make a stream out of a string.

stringstream ss(one);

int i;

ss >> i;

// From <string> ... read a line from an istream into a string object.

// Don't have to mess

// with size since the string can grow and grow.

getline(cin,one);

}

## Exercise 4: MadLibs

Write a program that reads a MadLib story description from a text file and prompts the user to enter the fill-ins. Then it prints the story filling in the inputs in the text.

A sample story is shown below. A single “%” on a line separates the inputs from the story. You should ignore blank lines in the input section.

AliensInClass.txt

%person% Tell me the name of someone you know.

%animal% Name an animal.

%compLanguage% What is your favorite programming language?

%animal2% Name a small animal.

%

\*\*\*\* ALIENS IN THE CLASSROOM!!! \*\*\*\*\*

\*\*\*\*\*\* a madlib story \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It started out an uneventful %compLanguage% class that evening.

But then %person% pulled a live %animal% out of a bag and

began to eat it. We tried to stop %person% but it was too late

for the %animal%.

"TAKE ME TO YOUR LEADER," %person% said pulling out a %animal2%

and pointing it like a weapon.

Suggestions:

1. Create a main that opens a hardcoded file name and reads/prints the lines one by one. Ignore any line that does not start with “%”. Break out of the loop with a lone “%”.
2. Parse the line into a key and a prompt. Print the prompt and read a string from the user.
3. Store the key and value in a global array with fixed size. 1000 should be plenty.
4. Add a second loop after the first to print the remaining lines of the story as-is.
5. Make a helper function that looks up the value of a key. Return “??” if the key is not found.
6. Make a loop that finds all “%....%” in the string and replaces them with the lookup value.
7. Use a command-line argument for the name of the story file (launch with CMD prompt).

### Step 1

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main() {

fstream storyFile;

storyFile.open("c:\\cpp\_7\_2012\\Day4\\AliensInClass.txt", fstream::in);

string s;

while(!storyFile.eof()) {

getline(storyFile,s);

if(s.size()==0 || s[0]!='%') continue;

if(s=="%") break;

cout << "::" << s << "::" <<endl;

}

storyFile.close();

system("pause");

}

### Step 2

while(!storyFile.eof()) {

getline(storyFile,s);

if(s.size()==0 || s[0]!='%') continue;

if(s=="%") break;

int secondPercent = s.find('%',1);

string key = s.substr(0,secondPercent+1);

string prompt = s.substr(secondPercent+2);

cout << prompt << " :";

string value;

cin >> value;

}

### Step 3

string keys[1000];

string values[1000];

int numberOfKeys = 0;

int main() {

...

while(!storyFile.eof()) {

...

cin >> value;

keys[numberOfKeys] = key;

values[numberOfKeys] = value;

++numberOfKeys;

}

...

}

### Step 4

cout << endl << endl << endl;

while(!storyFile.eof()) {

getline(storyFile,s);

cout << s << endl;

}

### Step 5

string lookupKeyValue(string key) {

for(int x=0;x<numberOfKeys;++x) {

if(keys[x] == key) {

return values[x];

}

}

return "??";

}

### Step 6

while(!storyFile.eof()) {

getline(storyFile,s);

while(true) {

int per = s.find('%');

if(per<0) break;

int per2 = s.find('%',per+1);

int keyLength = per2-per;

string first = s.substr(0,per);

string key = s.substr(per,keyLength+1);

string second = s.substr(per2+1);

s = first+lookupKeyValue(key)+second;

}

cout << s << endl;

}

### Step 7

int main(int argc, char\*\* argv) {

fstream storyFile;

storyFile.open(argv[1], fstream::in);

string keys[100];

string values[100];

int numberOfKeys = 0;

string lookupKeyValue(string key) {

for(int x=0;x<numberOfKeys;++x) {

if(keys[x] == key) {

return values[x];

}

}

return "??";

}

int main(int argc, char\*\* argv) {

fstream storyFile;

storyFile.open(argv[1], fstream::in);

string s;

while(!storyFile.eof()) {

getline(storyFile,s);

if(s.size()==0 || s[0]!='%') continue;

if(s=="%") break;

int secondPercent = s.find('%',1);

string key = s.substr(0,secondPercent+1);

string prompt = s.substr(secondPercent+2);

cout << prompt << " :";

string value;

cin >> value;

keys[numberOfKeys] = key;

values[numberOfKeys] = value;

++numberOfKeys;

}

cout << endl << endl << endl;

while(!storyFile.eof()) {

getline(storyFile,s);

while(true) {

int per = s.find('%');

if(per<0) break;

int per2 = s.find('%',per+1);

int keyLength = per2-per;

string first = s.substr(0,per);

string key = s.substr(per,keyLength+1);

string second = s.substr(per2+1);

s = first+lookupKeyValue(key)+second;

}

cout << s << endl;

}

storyFile.close();

}

# Day 5

**Objectives**

* Learn to use dynamic memory (heap)
  + new and delete
  + Array new and delete
  + Memory leaks and premature frees
* Develop dynamic array class
  + Design class interface
  + Implement and unit test
* Learn to use templates
* Learn to use the STL containers
  + Vector
  + List
  + Map
  + Iterators

**Lab**

* MadLibs Refactored (Objects and STL)

## Array of ints class

In the last lab we used a fixed array with a large max value to hold a variable sized list. Work an example of making this a formal class.

#include <iostream>

using namespace std;

void main() {

int data[100]; // Fixed size ... max size

int dataSize = 0; // Current size

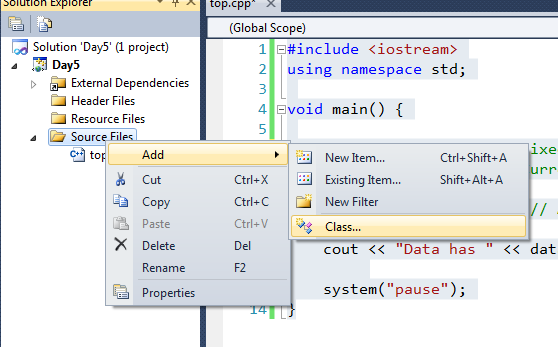
data[dataSize++] = 43; // Adding an item to the end

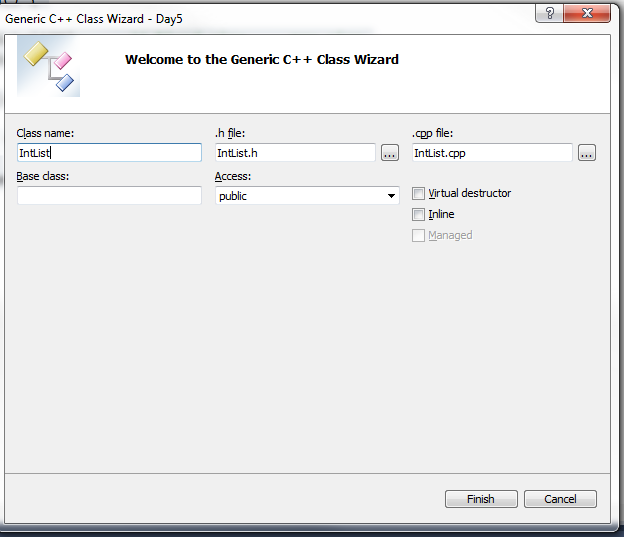
cout << "Data has " << dataSize << " elements.";

system("pause");

}

The IDE will stub out headers/source for you. In Express it looks like this.





You get a “.h” and a “.cpp” with the constructor and destructor already stubbed out.

Start with the interface. What kinds of things do you want to be able to do to this object? Add those to the header file.

#pragma once // Microsoft specific ... better than #ifndef stuff

class IntList

{

public:

IntList(void);

~IntList(void);

int getSize();

void addElementToEnd(int value);

int getElementAt(int index);

void setElementAt(int index, int value);

void insertElementAt(int index, int value);

void removeElementAt(int index);

};

Work the “main” unit test at the same time.

For now just a fixed size array. Add data and constructor.

class IntList

{

int data[1000]; // For now a fixed size

int size; // How many elements are in the list

public:

IntList::IntList(void) {

size = 0;

}

int IntList::getSize() {

return size;

};

#include <iostream>

using namespace std;

#include "IntList.h"

void main() {

IntList data;

cout << "Data has " << data.getSize() << " elements.";

system("pause");

}

Notice that the compiler doesn’t complain that you haven’t defined the other methods. It won’t care until you actually try and use them.

Some thoughts are to design the entire class and write the code completely then test. I prefer a more “extreme” (a type of programming methodology) technique by adding/testing/adding/testing incrementally. If you are working in an embedded environment this may not be practical.

What if you only use 3 or 4? What if you need more than 99? Fixed array is limited.

## Dynamic Memory (the Heap)

Local variables (created on the stack) are automatically managed. The compiler generates code to create them when the function is called. The compiler generates code to destroy them when the function returns.

Constructors and destructors are automatically called.

#include <iostream>

using namespace std;

class Point {

public:

Point() {

cout << "Create a Point" << endl;

}

~Point() {

cout << "Destroy a Point" << endl;

}

int x;

int y;

};

void doSomething() {

//Point p();

Point p; // Create Point on the stack (8 bytes)

// Code here

// End of function ... stack frame is closed ... Point goes away

}

void main() {

doSomething();

cout << "Hello World" << endl;

system("pause");

}

Global variables are created when your program starts. The compiler generates the code to call their constructors and destructors before and after your main.

Point a;

void main() {

//doSomething();

cout << "Hello World" << endl;

system("pause");

}

Put a breakpoint in the destructor to see it.

All these objects are “static”. You can predict the memory requirements at compile time.

In a large “real” application objects come and go in response to user inputs and the contents of parsed data files.

In “C” you used “malloc” and “free” to create dynamic memory areas. In C++ you use “new” and “delete” which takes care of the constructors and destructors.

void main() {

Point \*p = new Point(); // Create an object on the heap. Address in p.

Point \*q = new Point(1,2); // Create an object on the heap. Address in q.

cout << "Hello World" << endl;

delete p; // Destroy the object pointed to by p

delete q; // And q

system("pause");

}

You MUST keep up with the memory management yourself. If you forget to delete something then you “leak” it. It will remain on the heap for the life of the program. If you leak memory in a loop then you will slowly run out of memory and your program will crash.

Point \*p = new Point(); // Create an object on the heap. Address in p.

Point \*q = new Point(1,2); // Create an object on the heap. Address in q.

q = p; // Address of 1st Point now in p and q. 2nd point is forgotten (leaked).

delete p; // Destroy the object pointed to by p

q->x = 4; // This object has been deleted ...

p->y = 2; // ... we say it was "prematurely deleted"

delete p; // This is a "double free" or "double delete"

Point \*p = 0; // A "null" pointer.

// Useful for saying "p points to nothing".

// The language ignores delete of the null pointer. The

// IF check here is not needed.

if(p!=0) {

delete p;

}

int \* i = new int; // Create ONE int on the heap

int \* j = new int[100]; // Create 100 ints on the heap

delete i; // Delete ONE int

delete [] j; // Delete 100 ints

// The [] delete is needed for an object where the

// many destructors must be called.

Point \* p = new Point();

Point \* q = new Point[100];

delete p; // Call ONE destructor

delete [] q; // Call all the destructors

// You MUST match the [] new with [] delete

## Array of Ints with Dynamics

The data is private (encapsulated). The public API does not change.

class IntList

{

int \* data;

int size;

public:

IntList(void);

~IntList(void);

int getSize();

void addElementToEnd(int value);

int getElementAt(int index);

void setElementAt(int index, int value);

void insertElementAt(int index, int value);

void removeElementAt(int index);

};

IntList::IntList(void) {

data = new int[0];

size = 0;

}

IntList::~IntList(void) {

delete [] data;

}

void IntList::insertElementAt(int index, int value) {

int \* tempList = new int[size+1];

for(int x=0;x<index;++x) {

tempList[x] = data[x];

}

tempList[index] = value;

for(int x=index;x<size;++x) {

tempList[x+1] = data[x];

}

delete [] data;

data = tempList;

}

Work remainder of methods and unit test.

## Generic Code and Templates

Want to make a list of “doubles” … of “float” … of ANYTHING else. The logic is the same for all, just different type.

The logic is GENERIC … works with a GENERIC type.

I can use a word processor to search/replace the type for me. Mark the type:

class List

{

TOPHER \* data;

int size;

public:

List();

~List();

int getSize();

void addElementToEnd(TOPHER value);

TOPHER getElementAt(int index);

void setElementAt(int index, TOPHER value);

void insertElementAt(int index, TOPHER value);

void removeElementAt(int index);

};

void List::addElementToEnd(TOPHER value) {

if(size==99) return;

data[size++] = value;

}

Search TOPHER with “double” or “float” or “long” or whatever.

C++ adds a mechanism to do this code generation for you. You can tell the compiler to use your “template” to make new classes as needed. The compiler does the copy/paste/mangle for you.

The compiler needs all the code together in the template (not separate h and cpp).

template <class TOPHER>

class List

{

TOPHER data[1000]; // For now a fixed size

int size; // How many elements are in the list

public:

To “instantiate” the template you use the following syntax:

List<int> data;

List<double> more;

Does it work for “string”? Does it work for “Point”?

Try this simple implementation of Point:

class Point {

int x;

int y;

public:

Point(int nx,int ny) {

x = nx;

y = ny;

}

};

1>c:\cpp\_7\_2012\day5\day5\intlist.h(11): error C2512: 'Point' : no appropriate default constructor available

* The “new data[size+1]” uses the no-arg constructor. Must have one.
* “data[index] = value” used the assignment operator
* Usually a copy-constructor and ==

In order for a class to work in place of a built-in it must behave like a built in and include copy, assignment, and comparison operators.

Any “free/automatic” operators may or may not do what you want.

Most of the library classes have these operators.

List<String> data; // Works fine

Can have regular template functions (that aren’t classes).

Change getElement and setElement to use []

Make the class look like it is an array. Good or bad?

## STL

### vector

#include <iostream>

#include <vector>

#include <string>

using namespace std;

int main(int argc, char\*\* argv) {

// Vectors are dynamic arrays. Fast random access but takes a little

// time to grow/shrink.

vector<int> ages; // Vector of ints

vector<string> names; // Vector of strings

//vector<Point> points; // Vector of a user-defined type

names.push\_back("Hello"); // Add to end

names.push\_back("There");

names.push\_back("World");

int s = names.size();

bool empt = names.empty(); // Why not "size==0" ?

cout << names.at(0) << endl; // Get element at

cout << names[0] << endl; // Overloaded [] returns reference

string a = names.front(); // First element

a = names.back(); // Last element

names[0] = "HELLO"; // Use reference to change the value

for(int x=0;x<names.size();++x) {

cout << x << ":" << names[x] << endl;

}

system("pause");

}

### Iterators

Iterators are cursors. They point to elements in a collection. You can quickly access the element pointed to. You can advance the iterator to the next element. Changing the collection might invalidate the iterator and you have to start it again.

Containers provide “.start()” to get a pointer to the first and “.end()” to get ONE-PAST the end.

int main(int argc, char\*\* argv) {

vector<string> names; // Vector of strings

names.push\_back("Hello"); // Add to end

names.push\_back("There");

names.push\_back("World");

vector<string>::iterator i = names.begin(); // Point to first element

cout << \*i << endl; // Use pointer style \* to access (operator overload)

++i; // Next element

cout << \*i << endl;

--i;

cout << \*i << endl;

i = i + 2;

cout << \*i << endl;

i = names.begin();

++i;

names.insert(i,"NEW STRING");

// Typical for-loop over all elements;

for(vector<string>::iterator j=names.begin();j!=names.end();++j) {

cout << \*j << endl;

}

system("pause");

}

### list

#include <iostream>

#include <list>

#include <string>

using namespace std;

int main(int argc, char\*\* argv) {

// List maintains a linked-list making it super-fast to move things around

// and add to end/beginning. But getting to a particular element takes

// time.

list<string> names;

names.push\_back("Hello");

names.push\_back("There");

names.push\_back("World");

// front(), back(), insert(), erase()

// No "at" or [] operator. You have to iterate to the desired element.

// Typical for-loop over all elements;

for(list<string>::iterator j=names.begin();j!=names.end();++j) {

cout << \*j << endl;

}

system("pause");

}

names.push\_back("ZZZ");

names.push\_back("AAA");

names.push\_back("BBB");

names.sort(); // Uses the "operator<" you define for a class or built-in "<".

// You can also define a function that takes two elements and returns bool true

// if first goes before second.

### map

int main(int argc, char\*\* argv) {

map<string,string> names;

// Put/Get ... using STL "pair".

names.insert(pair<string,string>("123","Bob"));

cout << names.at("123");

// Use the overloaded []

names["123-45-6789"] = "Chris";

names["000-11-2222"] = "John";

cout << names["000-11-2222"] << endl;

cout << names["99"] << endl;

// Must return something if not found. If it were pointers we could test

// for a null pointer to know the find failed.

// The thing returned is a new object of the type using default constructor.

// In this case ... an empty string

map<string,string>::iterator i = names.find("888");

if(i==names.end()) {

cout << "NOT FOUND" << endl;

}

system("pause");

}

int main(int argc, char\*\* argv) {

map<string,string> names;

names["123-45-6789"] = "Chris";

names["000-11-2222"] = "John";

map<string,string>::iterator i = names.begin();

// The iterator moves over pair<string,string>. The "pair" class

// has a "first" and a "second":

cout << i->first << endl;

cout << i->second << endl;

system("pause");

}

## Algorithms and Sorting

## Exercise 5: MadLibs Refactored

Refactor the last exercise. Write a program that reads a MadLib story description from a text file and prompts the user to enter the fill-ins. Then it prints the story filling in the inputs in the text.

This time use the STL and a map<string,string> to capture the user fill-ins.

Your first challenge is to get the old code working with the new STL changes.

Now create a MadLib class to hold the game logic. Add such methods as “readStory” to parse a given file (by name) and “printStory” to show it on the screen. Try reading the input section and story into memory instead of prompting the user as you read the story file.

Advanced: try making a “MadLibInput” class that includes three strings: the prompt, the name, and the user input. Change your map to a map<string,MadLibInput>. You will need an “==” operator for the map to work.

A sample story is shown below. A single “%” on a line separates the inputs from the story. You should ignore blank lines in the input section.

AliensInClass.txt

%person% Tell me the name of someone you know.

%animal% Name an animal.

%compLanguage% What is your favorite programming language?

%animal2% Name a small animal.

%

\*\*\*\* ALIENS IN THE CLASSROOM!!! \*\*\*\*\*

\*\*\*\*\*\* a madlib story \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It started out an uneventful %compLanguage% class that evening.

But then %person% pulled a live %animal% out of a bag and

began to eat it. We tried to stop %person% but it was too late

for the %animal%.

"TAKE ME TO YOUR LEADER," %person% said pulling out a %animal2%

and pointing it like a weapon.

# Day 6

**Objectives**

* Casting up and down the hierarchy
  + IS-A
  + Automatic conversion

**Lab**

## Is-A

#include <iostream>

using namespace std;

class Point {

public:

int x;

int y;

};

class ColorPoint : public Point {

public:

int color;

};

void showPoint(Point \* p) {

cout << "x:" << p->x << " y:" << p->y << endl;

}

void showColor(ColorPoint \* p) {

cout << "x:" << p->x << " y:" << p->y << " color:" << p->color << endl;

}

void main() {

Point a; // 2 ints on the stack: x,y

ColorPoint b; // 3 ints on the stack: x,y,color

showPoint(&a); // Pass pointer to 2 ints

showColor(&b); // Pass pointer to 3 ints

// This works because the memory for "ColorPoint" is the

// same as the memory for "Point" with more added. You can

// tell the compiler to forget about that "more added".

showPoint(&b);

system("pause");

}

The ColorPoint can be safely used everywhere the code uses Point because it has all the memory and all the methods.

We say that ColorPoint IS-A Point … plus more. If the more doesn’t interest you then you can forget about it.

This conversion to a base class always works and is perfectly safe. The compiler does it automatically.

showPoint(&b);

Point \* x = &a;

ColorPoint \* y = &b;

Point \* z = &a; // Automatic

x = y; // Automatic

y = (ColorPoint \*)x; // NOT automatic, but you can insist

The conversion to a derived class does NOT always work, but it may be perfectly correct. You have to tell the compiler to make the conversion. Hopefully you are right.

## Program to the Interface – Not the Implementation

You know better than to write code this way! You should call methods on an object and not access the data directly.

If you access directly as in “x = p.x” then the compiler does a memory copy. If you call a method as in “x = p.getX()” then the developer can insert code in place of the memory copy (or rather just before).

Data is almost always private. The implementation can change without you knowing.

You should think of objects in terms of what they can do and not what they are. The “Point” can respond to “sayHi”. The “Point” can respond to “getX”.

The fact that “Point” has an “int x” inside is nothing you care about. That is totally its private business.

## Polymorphism

When you get down to it, my code doesn’t really care if the “thing” I am using is a Point or ColorPoint or SomeNewPoint. All my code really cares about is that it has a set of specific methods. For instance, my code only cares that the “thing” can respond to “sayHi”. It doesn’t care what “sayHi” actually does. In fact, it gets great power from not knowing.

The power is that it can work with all kinds of “things” that can “sayHi” … different kinds of things I select in my code and even “things” that have yet to be created.

The “streams” are a familiar example. The “stream operator” works on a stream of bytes, but it doesn’t care about what the exact stream is. When your code does “os << 24” do the bytes go to the screen or a file or a socket? It doesn’t know … or care. The program can even change the specific type at runtime.

Surprisingly, this power isn’t easy to code. Imagine you have objects “User” and “ColorPoint” both with “sayHi” methods. Write a function that calls “sayHi” on whatever you pass it. Hint: what are you going to accept as an incoming parameter for the “thing that you can call sayHi on” ?

C does have “functors” … pointers to functions. The code would load up an array of function-pointers. The first would be “sayHi”. The second would be “sayBye”. The third would be “getX”. And so on.

Then you pass the table of pointers to the function. Instead of calling a specific “sayHi” function the code would get the first pointer from the table and call THAT function indirectly.

The syntax is cumbersome.

C++ uses the same mechanism (a table of pointers) but hides the nastiness for you.

First you need to define a group of functions that go in the table. You already have a grouping of functions … you put them in a class.

Then another object can override those methods with their own behavior.

#include <iostream>

using namespace std;

class Point {

int x;

int y;

public:

void sayHi() {

cout << "Point Hi" << endl;

}

void sayBye() {

cout << "Point Bye" << endl;

}

};

class ColorPoint : public Point {

int color;

public:

void sayHi() {

cout << "ColorPoint Hi" << endl;

}

void sayBye() {

cout << "ColorPoint Bye" << endl;

}

};

void greet(Point \* p) {

p->sayHi();

p->sayBye();

}

void main() {

Point a;

ColorPoint b;

greet(&a);

greet(&b);

system("pause");

}

Why doesn’t this work?

Think back to the memory footprint. There is no way for the function to know what the object in memory really is.

You must tag the methods in your class that you want to be part of this table-of-functions. You use the “virtual” keyword.

class Point {

int x;

int y;

public:

virtual void sayHi() {

cout << "Point Hi" << endl;

}

virtual void sayBye() {

cout << "Point Bye" << endl;

}

};

You should put the keyword “virtual” on the derived class, but it is understood if you don’t.

If ANY methods in your class are virtual then the compiler generates a table of function pointers for you. This is called the “virtual table” or “vtable”.

The compiler stores the address of the table in the first four bytes of memory for your object. Your object is larger now.

Now for “p->sayHi()” the compiler generates different code. Before it just put in the name of the exact function to call. Now it generates code to look up the function pointer in the table and use it instead.

The code looks the same, but the compiler does something completely different.

Now the example works.

## Virtuals

If it is so powerful then why not always make functions virtual?

It adds to the memory footprint and it adds to the performance. You get to pick and choose which methods you want to be called from a table and which you don’t.

Object Oriented programming proper is really about virtual methods and programming to the interface.

OO applications create lots of objects on the heap. OO applications use pointers (rather, references) to access them. OO applications depend on “overriding” these virtual methods to implement behavior that the using code knows nothing about.

Graphics objects: they all have “draw” and “erase”, but circles and squares implement differently. The drawing library works with “things” that can “draw” and “erase”.

You’ll see a common base class and code to use it and a bunch of specialized objects that derive from the base class.

If you have a virtual table you need to decide which methods you want to go in it. You should ALWAYS put the destructor in there. If someone deletes the object you want the derived constructor to run. Just remember … ANY virtual methods means ALWAYS virtual destructor. And you have to manually make it virtual.

## Random Point Example

(show magnitude and calling virtual getters)

## Pure Virtual

If the base class is just a grouping of methods then you don’t need a “default” implementation for the methods. You can make the method a pure virtual.

virtual void saySomething() = 0;

The syntax means to put a null-pointer in the virtual table. If an object has a pure virtual then you can’t create it. You can only create a derived class that provides an implementation.

A base class used only to list a number of methods is called an “interface”. Java actually has a separate keyword for interfaces. In C++ there is no difference.

Work this example.

class Speaker { // An "interface"

public:

virtual ~Speaker() {}

virtual void sayHi() = 0; // Must be overriden

virtual void sayBye() = 0;

};

class Feed {

public:

virtual ~Feed() {}

virtual void eat() = 0;

};

class Cat : public Speaker, public Feed {

public:

};

## Design Patterns

Discussion if time permits. Decorator is the easiest.