C++ Refresher

Introduction

As you begin ITP 380, you may be a bit rusty with C++. This document provides a brief refresher on several C++ topics that will prove useful during this semester.

C++ Basics

Due to the prerequisites for this course, we assume you have at least some C++ knowledge.

If not, we'd **highly suggest** reviewing some online tutorials to become familiar with the basic language syntax and features.

(Feel free to skip these if you are already somewhat comfortable with C++)

- Structure of a program
- Variables and types
- Operators
- Control structures
- Functions
- Classes (I)
- Classes (II)
- Friendship and inheritance
- Polymorphism

The remainder of this document highlights additional specific topics that will come in handy this semester.

Pointers

A **pointer** is a variable that stores a memory address as an integer value. The integer stored in the variable is interpreted as a memory address.

A pointer is denoted by the * symbol after the variable type:

```
char val;     // This is a char value.
char* valPtr; // This is a pointer to a char value.
```

While a program is running, all variables, classes, functions, and objects used by your program must be stored in memory somewhere. Each location in memory has an address. You can use a pointer to hold a memory address and access the data stored at that address.

The type of the pointer indicates what type of value lives at the memory address held by the pointer. For example, a float* points to a memory address that will be interpreted as a float value.

Here's another example:

Declaring a Pointer Example

```
int x = 0;
double y = 5.0;

int* z = &x;
```

	Memory	
Variable	**Address**	**Value**
x	0x04	0
y	0x08	5.0
z	0x10	??

In this example, we are using the "&" (address of) operator to assign the address of variable x to be the value held by z. What value will the "z" pointer variable hold after this operation?

Answer: 0x04

Dereferencing a Pointer

A pointer holds a memory address as an integer. The value of a pointer variable is a memory address. But what if you want to access the data stored at that memory address?

If a pointer holds a valid memory address, **dereferencing** a pointer allows you to retrieve the value stored at that memory address.

Dereferencing a Pointer

```
int x = 0;
double y = 5.0;

int* z = &x;

// *z means dereference z
std::cout << *z;
```

	Memory	
Variable	Address	Value
x	0x04	0
y	0x08	5.0
z	0x10	0x04

The int* variable "z" holds the address of "x". By using the dereference operator (*z), we are saying "go to the memory address stored in this pointer (0x04) and retrieve the value there (0)."

If you have a pointer to a class or struct instance, you can access public functions or variables on that object using the "->" operator. This is another syntax for dereferencing a pointer.

Null Pointers

A pointer doesn't always point to a valid memory address. Pointers can be used to represent "null" or "non-existent" objects. When this is the case, the pointer should be set equal to **nullptr**.

```
Actor* actor = nullptr;
// Some code omitted...

// Always check for null before using a pointer!
if(actor != nullptr)
{
    actor->SetScale(Vector3(1, 1, 1));
}
```

Attempting to dereference a null pointer will cause your program to crash. Therefore, you must always check whether a pointer is null unless you are absolutely sure it isn't.

When to Use Pointers?

Pointers are useful, but they should only be used when necessary. They should only be used when dynamic allocation is necessary (see next section), or when you need be able to represent an object where "nothing" is a valid option (because it can be set to **nullptr**).

Dynamic Allocation

In C# or Java, all new object instances are created using the **new** keyword. For example:

```
Game game = new Game(); // C# or Java code
```

As a result, if you are familiar with C# or Java, you may be tempted to use the **new** keyword when creating new object instances in C++. **DO NOT DO THIS!**

In C++, the **new** keyword allows you to dynamically allocate a new object using heap memory (as opposed to stack or global memory). It should be used sparingly and only when necessary.

Dynamic allocation is usually required when you need an object to continue existing outside the scope it was created. Consider this example, which DOES NOT use dynamic allocation:

```
int main(int argc, char** argv)
{
    // No dynamic allocation here
    Game game;
    bool success = game.Initialize();
    if(success)
    {
        game.RunLoop();
    }
    game.Shutdown();
    return 0;
}
```

This notation is simple and safe: it allocates no dynamic memory ("game" is created on the stack), and the object is automatically deallocated when the leaving the function (in other words: no memory leaks). Note that you do not use the **new** or **delete** keywords at all. You should use this type of allocation whenever possible – it is simpler!

Sometimes, however, it is necessary to dynamically allocate objects. To do this, you use the **new** and **delete** keywords:

```
int main(int argc, char** argv)
{
    // A contrived dynamic allocation example
    // You don't really need to use dynamic allocation here though!
    Game* game = new Game();
    bool success = game->Initialize();
    if(success)
    {
        game->RunLoop();
    }
    game->Shutdown();
    delete game;

    return 0;
}
```

Dynamic allocation is less safe and can needlessly complicate your code. Because you're using pointers, you have to worry about checking for **nullptr**. You must also call **delete** when you are done with the object, or else you will leak memory.

When should you use dynamic allocation? There are only a few situations where it is warranted:

- When you want to allocate some memory whose size is not known at compile time.
- When you want to allocate a large amount of persistent/reusable memory.
- When you want an object to persist outside the scope in which it is created. For example, an object created inside a function will be automatically destroyed when the function ends – unless it is dynamically allocated.

Pass by Value, Pointer, and Reference

In C++, there are three ways that we can pass arguments to a function: by value, by pointer, and by reference. This section explains how these methods differ, and the benefits of each.

Pass by Value

When you pass variables to a function, the function creates its own local copies of those variables.

This is generally OK for basic types (int, bool, float, etc.) or very small class objects. But for larger class objects, this process of creating copies can slow down your program.

```
// Declares “add”, which passes lhs and rhs by value
int Add(int lhs, int rhs);

// BAD! Makes unnecessary copies of lhs/rhs
Vector3 Add(Vector3 lhs, Vector3 rhs);
```

Another problem with pass by value: since the function creates copies of the objects you pass in, any modifications to those variables in the function **will not persist** after the function returns (since only the function’s local copies were modified). In some cases, this is actually a good thing, but sometimes you may want to keep the modifications.

Pass by Pointer

By making use of pointers, we can solve the two problems that occur with “pass by value”:

1. To avoid making copies of larger objects, pass a pointer to the object. Then, only the small pointer variable is copied, instead of the entire large object.

2. The function still creates its own copy of the passed pointer, but the pointer still points to our original object. Therefore, if the function modifies the object through the pointer, those changes **will persist** after the function returns!

```
// “actor” is passed by pointer
void AddActor(Actor* actor);
```

The main problem with “pass by pointer” is that we are now using pointers and all the complications that come along with them (null checks). When you are passing pointers around, make sure you keep track of who is responsible for deleting the pointer!

Pass by Reference

First off, what is a **reference**? A reference mostly acts like a pointer, but with one big difference: **a reference can never be null**. You also don't have to delete references. You can very seamlessly get a reference to a value or create a value from a reference.

A reference is denoted using the "&" symbol after the variable type.

```
// lhs and rhs are passed by reference
Vector3 Add(Vector3& lhs, Vector3& rhs);
```

Passing by reference shares the same benefits of passing by pointer: you avoid copying larger objects, and the function is able to modify the passed-in objects.

Pass by reference is most useful when you need to pass non-basic types that aren't dynamically allocated to a function. But also keep in mind that it's possible to convert from value to pointer to reference and back again, if the need arises:

```
int val = 5;
int* valPtr = &val; // Value to pointer (using "address of" operator)
int& valRef = val;  // Value to reference
int val2 = *valPtr; // Pointer to value (using "dereference" operator)
int val3 = valRef;  // Reference to value
```

Const Keyword

The **const** keyword in C++ generally means "this thing cannot change" or "this thing will not cause changes". The keyword itself is used in several different contexts, which we'll review here.

Const Variables

The most straightforward use of **const** is to create a constant variable, which can't be modified.

```
const int PADDLE_WIDTH = 5;
PADDLE_WIDTH = 4; // ERROR! variable can't be modified
```

Use this when you have a value that should never change. Using **const** variables can also improve the readability of your code (instead of "rect.w = 5", you can say "rect.w = BALL_WIDTH").

You can create **const** local variables, class member variables, or global variables.

Const Function Arguments

When using "pass by pointer" or "pass by reference" (see previous section), one "benefit" is that the called function can modify the objects that you pass in. However, what if you **DID NOT** want the function to be able to modify the passed objects?

You can add **const** to function arguments to indicate that the function cannot and will not modify the passed in objects. In fact, the compiler will throw an error if you attempt to modify them.

```
// Add is disallowed from changing lhs or rhs
Vector3 Add(const Vector3& lhs, const Vector3& rhs);
```

You'll encounter scenarios where you want to "pass by pointer" or "pass by reference" to avoid copying the objects, but you don't want the function to accidentally modify the objects. An "Add" function is a great example – it should add the two items and return a new item, but it *should never* modify the two passed in objects!

Const Member Functions

Finally, you may have a **const** variable, but you want to call some function on it that won't modify the object. A good example is a "get" function – it retrieves a value from the object, but it *does not* modify the object.

Consider this example:

```
class Vector3 {
public:
    float GetX(); // Returns X value to caller – no modification
};

void Calculate(const Vector3& v) {
    cout << v.GetX(); // ERROR! GetX might modify a const variable
}
```

The reference variable **v** passed to "Calculate" is **const**, so we can't modify it. As programmers, we understand that calling "GetX()" would not modify the object, but the compiler doesn't know that – you have to tell it.

To do that, simply add the **const** keyword after a function that will not modify the object's data.

```
class Vector3 {
public:
    float GetX() const;
};

void Calculate(const Vector3& v) {
    cout << v.GetX(); // OK – called const function GetX()
}
```

As you write your classes, consider whether a function will modify the object at all. And if not, mark it **const** so it can be called on any **const** pointers or references.

Auto Keyword

The **auto** keyword in C++11 allows the compiler to deduce a variable's type, rather than requiring you to write out the entire thing.

In ITP 380, this mostly comes in handy for iterators.

```
// long and annoying
std::vector<int>::iterator myIter = myVect.begin();

// short and sweet
auto myIter = myVect.begin();
```

Two warnings with the **auto** keyword: first, using **auto** can make your code harder to read and understand, so don't use it everywhere. Second, don't use **auto** just because you don't understand what the return type of a function is!

Don't forget about the difference between auto variables and auto reference variables. If you do not want to make a copy of a variable, you'll want to add the &. For example:

```
auto myThing = thing;            // this is a copy of thing
auto& myThing = thing;           // this is a reference to thing
const auto& myThing = thing;     // this is a const reference to thing
```

Range-Based For Loops

In C++, you'd traditionally need to use a standard "for" loop to iterate over a list:

```
std::vector<int> vec;
vec.push_back(10);
vec.push_back(20);

for(int i = 0; i < vec.size(); i++)
{
    std::cout << i << std::endl;
}
```

Since C++11, you can also use a "range-based for" loop, which is similar to a "foreach" loop in other languages:

```
for(int i : vec) // makes copies of each i
{
    i += 10; // DOES NOT persist outside of loop
    std::cout << i << std::endl;
}
```

Note that we can also use **auto** here, if we don't want to write out the variable's type:

```
for(auto i : vec) // makes copies of each i
{
    i += 10; // DOES NOT persist outside of loop
    std::cout << i << std::endl;
}
```

Additionally, the variable in the for loop is a **copy**, not a **reference**. You need to explicitly add the reference operator (&) if you want changes in the loop to persist.

```
for(auto& i : vec) // Modify the integers in vector
{
    i += 10; // Persists outside of loop
    std::cout << i << std::endl;
}
```

Virtual and Override

In a class hierarchy, a parent class can specify **virtual** functions that can be **overridden** by child classes. However, it is easy to make mistakes when doing this. Consider this example:

```
class A
{
public:
    virtual void TakeDamage(int amount);
};

class B : public A
{
public:
    void TakeDamage();
};
```

Class A defines a virtual function "TakeDamage" which can be overridden by a child class. Child class B overrides "TakeDamage" – but do you see a problem?

The "TakeDamage" function in class B doesn't have the right argument list. This code will compile and run, but it won't behave correctly. You'll be left scratching your head, and potentially wasting hours debugging the issue.

To avoid making this mistake, always use the **override** keyword when you intend to override a base class function. This tells the compiler what your intention is, and it can give you an error if there's a problem.

```
class B : public A
{
public:
    void TakeDamage() override; // ERROR! No matching virtual func!
};
```

This stops your code from even compiling until you resolve the problem. **Always use the override keyword when you intend to override a virtual function!** You'll save yourself a lot of time and headaches.

Type Casting

Occasionally, you need to "cast" basic types between one another, such as converting an **int** to a **float**. Such cases are trivial, and can be accomplished like so:

```
int myInt = 5;
float myFloat = (float)myInt; // "C-style" cast
float myFloat2 = static_cast<float>(myInt); // "static" cast
```

However, there are also times when type casting can be a bit more complicated, especially when dealing with inheritance hierarchies. Consider this example:

```
class Actor
{
public:
    void Update(float deltaTime);
};

class Player : public Actor
{
public:
    void Respawn();
};

void OnPlayerDied(Actor* player)
{
    //TODO: Call Respawn() on player.
}
```

In this example, we have a class, **Player**, that inherits from **Actor**. Class **Player** has a function called "Respawn". We want to call that function on a variable in "OnPlayerDied".

You might try this, but it wouldn't work – since you have an **Actor***, and not a **Player***, you can't call the function "Respawn" directly.

```
void OnPlayerDied(Actor* player)
{
    player->Respawn(); // ERROR! No member "Respawn" in "Actor"
}
```

However, since **Player** is a subclass of **Actor**, you can perform a cast to access the function.

```
void OnPlayerDied(Actor* player)
{
    Player* p = static_cast<Player*>(player);
    p->Respawn(); // OK
}
```