Templates

Generic functions and classes.

Normal Functions

- A normal function is:
  - A list of inputs/outputs.
  - A set of instructions for manipulating data.

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

Normal Functions

- If we want a function that does the same thing with a different data-type, we have to make an overload that accepts the new data-type.

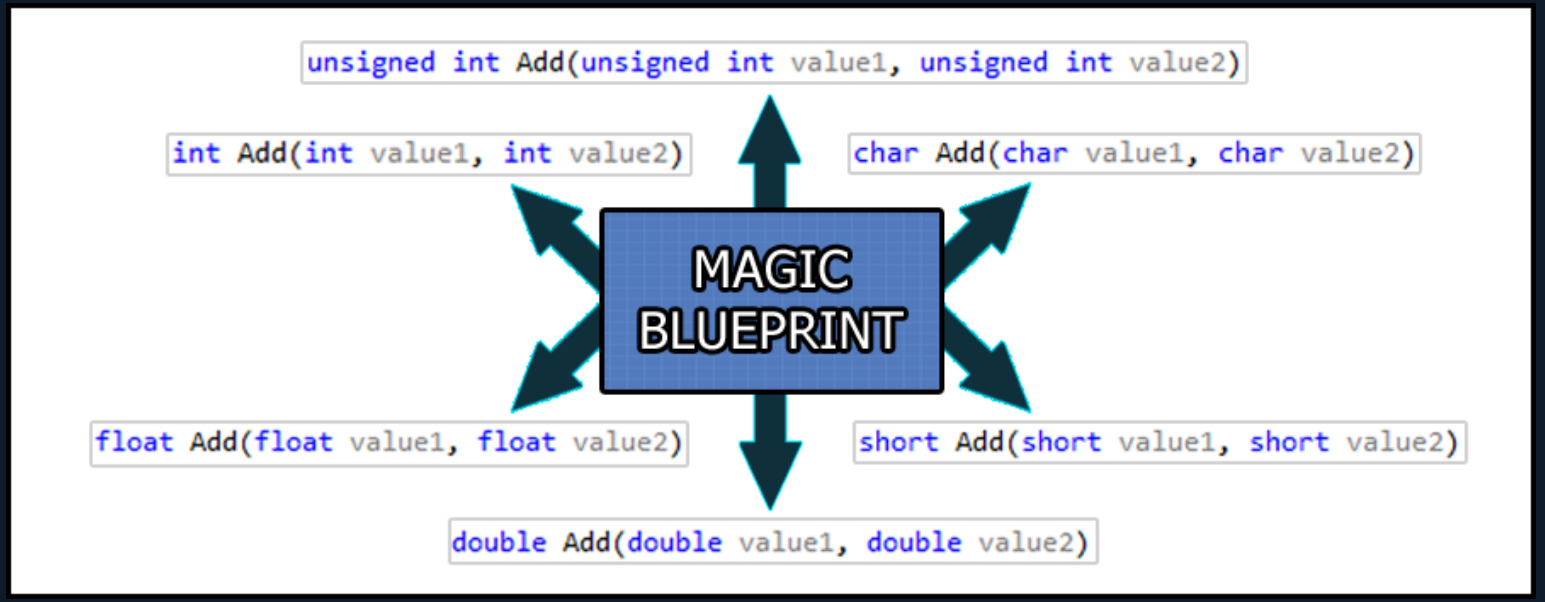
```
int Add(int a, int b)
{
    return a + b;
}
```

```
float Add(float a, float a)
{
    return a + b;
}
```

Template Functions

- Template functions solve this problem.
  - Template functions are not functions.
  - Template functions are blueprints for normal functions.
  - Template functions use generic/place-holder data-types.
  - When a previously unused data-type is given to a template function, the compiler generates a new ‘normal’ function that handles it.

Template Functions



Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Template Functions

- To make a template function we first flag the function with the ‘template’ keyword.

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

```
template
int Add(int a, int b)
{
    return a + b;
}
```

Template Functions

- We then create a generic data-type with the ‘typename’ keyword (i.e. template parameters).

```
template
int Add(int a, int b)
{
    return a + b;
}
```

```
template<typename T>
int Add(int a, int b)
{
    return a + b;
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Template Functions

- Finally we use the generic data-type in place of our otherwise static data-types (where appropriate).

```
template<typename T>
int Add(int a, int b)
{
    return a + b;
}
```

```
template<typename T>
T Add(T a, T b)
{
    return a + b;
}
```

Template Functions

- We can then invoke the method like normal.
- The compiler will look at the argument types and generate a copy of the template function, replacing any ‘T’s it finds with the correct data-type (e.g. int).

```
void main()
{
    int   i =  Add(5, 7);         //works!
    float f =  Add(5.0f, 7.0f);   //works!
}
```

Template Functions

- We can also explicitly tell the compiler which data-type to replace ‘T’ with when it generates the new ‘normal’ function.

```
void main()
{
    int   i =  Add<int>(5, 7);          //works!
    float f =  Add<float>(5.0f, 7.0f); //works!
}
```

Template Functions

- If you feed multiple data-types into a template parameter, you must explicitly declare which data-type the compiler should use.

```
template<typename T>
T Add(T a, T b)
{
    return  a + b;
}

void main(){
    int   r1 = Add(5, 7.0f);          //error!
    int   r2 = Add<int>(5, 7.0f);     //works!
    float r3 = Add<float>(5, 7.0f);   //works!
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Template Functions

- The rules for data-type conversion still apply when passing values to generic data-types.

```
template<typename T>
T Add(T a, T b){
    return  a + b;
}
```

If called as Add<int>
The compiler generates

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

  - So then what is the value of ‘result’ in the following?

```
void main(){
    int result = Add<int>(5, 7.5f);
}
```

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Template Functions

- Template functions can have multiple generic data-types; they don’t have to be called ‘T’.

```
template<typename A, typename B, typename C>
C Add(A a, B b)
{
    return (C)(a + b);
}

void main()
{
    int c = Add<int, float, int>(5, 7.0f);
}
```

Template Functions

- For specific data-types that should be handled differently, we can create a specialised template function (think function overloading).

```
template<class T>
bool LessThan(T a, T b)
{
    return a < b;
}

template<>
bool LessThan(const char* a, const char* b)
{
    return strlen(a) < strlen(b);
}
```

Template Functions

- We can make ‘non-type’ template parameters that behave a lot like normal function parameters.
  - Must be integral types (int, bool, enum, ptr*, ref&, etc).

```
template<typename A, int B> //B is a non-type
int Add(A a)
{
    return  a + B; //use B like normal variable
}

void main(){
    int result = Add<int, 20>(10); //works!
}
```

Template Functions

- Non-type parameters are generated by the pre-processor just like regular template parameters.
  - A new function is generated for each unique value.

```
void main(){
    int r1 = Add<int, 1>(9);  //generate function for int + ‘1’
    int r2 = Add<int, 2>(8);  //generate function for int + ‘2’
    int r3 = Add<int, 1>(6);  //reuse existing func for int + ‘1’
    int r4 = Add<int, 2>(5);  //reuse existing func for int + ‘2’
    int r5 = Add<char,1>(9);  //generate function for float + ‘1’
    int r6 = Add<char,2>(8);  //generate function for float + ‘2’
}
```

Template Functions

- Because non-type parameters are generated by the pre-processor, they are constant values.

```
template<typename T, int N>
void ZeroLast(T a[N]) //N can be used like a constant
{
    a[N-1] = 0;
}

void main()
{
    int array[5] = {1,2,3,4,5};
    ZeroLast<int, 5>(array);
}
```

Template Functions

- Because template functions are generated by the pre-processor at compile time before the linker, the function declaration and definition must be in the same location for the pre-processor to read from.
  - This means that template functions should exist entirely:
    - Within a *.h* file (for inclusion and use by other files).
    - Or within a *.cpp* file (for use within the same file).

Interlude

- You may now do exercise 1 from the Templates Lab.

Canberra Institute of Technology

aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX

Template Classes

- Template classes are much like template functions.
  - Template classes are not classes.
  - Template classes are blueprints for normal classes.
  - Template classes use generic/place-holder data-types.
  - When a previously unused data-type is given to a template class, the compiler generates a new ‘normal’ class that handles it.

Template Classes

- Template classes are also defined in a very similar way to template functions.
  - The generic types can be used anywhere within the class.

```
template<typename T, int N>
class MyContainer
{
public:
    void Set(int index, T item){
        items[index] = item;
    }

private:
    T items[N];
};
```

```
void main()
{
    MyContainer<int, 10> container;

    for(int i = 0; i < 10; ++i)
        container.Set(i, i);
}
```

Template Classes

- Like template functions, template classes must have their declaration and definition in the same location.
  - Either a *.cpp* for internal use or a *.h* for external use.
  - A definition may be separated within a file however:

```
template<typename T, int N>
class MyContainer
{
public:
    void Set(int index, T item);

private:
    T items[N];
};
```

```
//somewhere under class definition...
template<typename T, int N>
void MyContainer<T, N>::Set(int index, T item)
{
    items[index] = item;
}
```

Template Dangers

- Debuggers can get confused about which instance of a class/function to step into/inspect.
- Nested templates or template hierarchies can quickly become difficult to read and write:

```
template <class Q>
template <class N>
QuadTree<Q>::Node<N>::Node(float* apTopLeft, float* apBottomRight, int aLevel){
    ...
}
```

Summary

- Templates are incredibly useful when correctly used.
- Many libraries make use of templates, such as STL.
- You will need to understand them to make games.

- Time to do exercise 2 from the Templates Lab…