SWINBURNE UNIVERSITY OF TECHNOLOGY

COS30008 Data Structures and Patterns
Recursion, Linked Lists

Recursion

- If a procedure contains within its body calls to itself, then this procedure is said to be recursively defined.
- This approach of program specification is called recursion and is found not only in programming.
- If we the define a procedure recursively, then there must exist at least one sub-problem that can be solved directly, that is without calling the procedure again.
- A recursively defined procedure must always contain a directly solvable sub-problem. Otherwise, this procedure does not terminate. This condition that is used to stop the recursion is called the **base case**.

Problem-Solving with Recursion

- Recursion is an important problem-solving technique in which a given problem is reduced to smaller instances of the same problem.
- The general structure of a recursive definition is

$$\underbrace{X}_{\text{left hand side}} = \ldots \underbrace{X}_{\text{right hand side}} \ldots$$

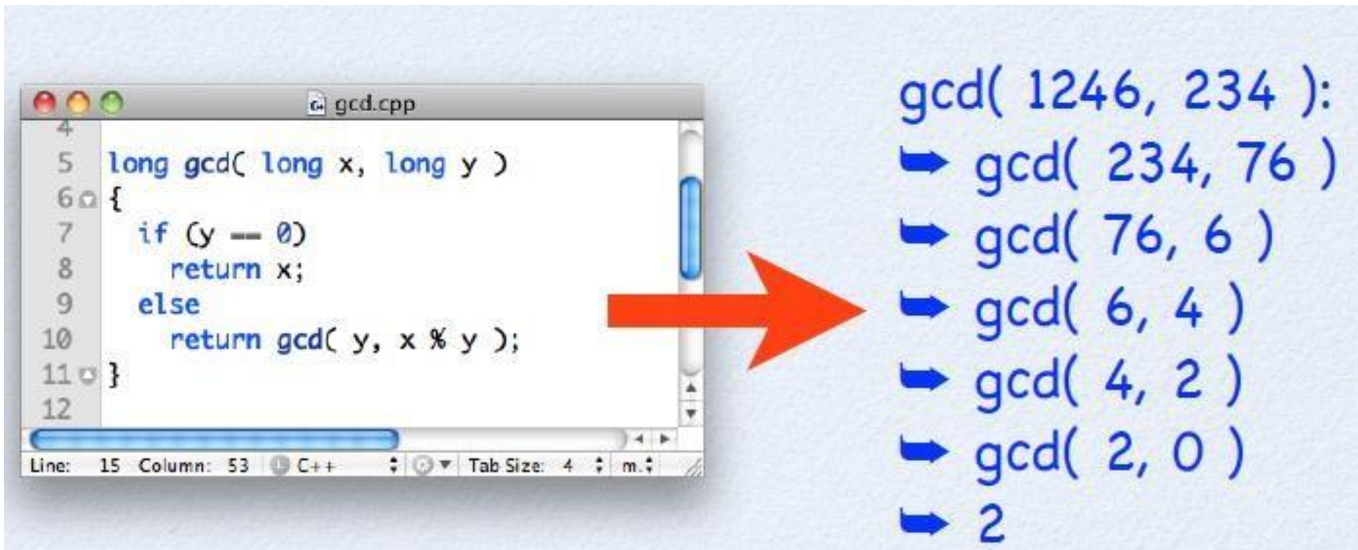
Fibonacci

- The Fibonacci numbers are the following sequence of
- numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
- In mathematical terms, the sequence F(n) of Fibonacci numbers is defined recursively as follows:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$

```
long fib( unsigned long n )
{
  if (n <= 1)
    return n;
  else
    return fib( n-1 ) + fib( n-2 );
}
```

Tail-Recursive

- Tail recursion is a form of linear recursion. In tail recursion, the recursive call is the last operation the function does.




```
int pow(int a, int b)
{
        if (b==1)
        return a;
        else
        return a * pow(a, b-1);
}
```

Other type of Recursion

- Binary recursion
  - Some recursive functions don't just have one call to themselves, they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.
  - E.g. **Fibonacci sequence:** By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two

```
#include <iostream>

using namespace std;
int recursiveFib (int n)
{
    // base case
    if (n <= 1)
        return n;
    // binary recursive call
    return recursiveFib(n-1) + recursiveFib (n - 2);
}

int main ()
{
    int n = 6;
    cout<<n<<"th fibonacci number is "<<recursiveFib(n)<<endl;
    return 0;
}
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Towers of Hanoi

- Problem:
  - Move disks from a start peg to a target peg using a middle peg.
- Challenge:
  - All disks have a unique size and at no time must a bigger disk be placed on top of a smaller one.

The Recursive Procedure

```cpp
#include <iostream>

using namespace std;

void move( int n, string start, string target, string middle )
{
    if ( n > 0 )
    {
        move( n-1, start, middle, target );
        cout << "Move disk "  << n << " from " << start
             << " to "<< target << "." << endl;
        move( n-1, middle, target, start );
    }
}

int main()
{
    move( 3, "Start", "Target", "Middle" );

    return 0;
}
```

Line: 22 Column: 1 C++ Tab Size: 4 main

Terminal

```
Sela:HIT3303 Markus$ ./hanoi
Move disk 1 from Start to Target.
Move disk 2 from Start to Middle.
Move disk 1 from Target to Middle.
Move disk 3 from Start to Target.
Move disk 1 from Middle to Start.
Move disk 2 from Middle to Target.
Move disk 1 from Start to Target.
Sela:HIT3303 Markus$
```

Recursive Problem-Solving: Factorials

- The factorial for positive integers is

$$n! = n * (n - 1) * \ldots * 1$$

- The recursive definition:

$$n! = \begin{cases} 1 \textit{ if } n = 0 \\ n * \textit{(n-1)! if } n > 0 \end{cases}$$

Recursion in C++

URL: https://www.youtube.com/watch?v=4agL-MQq05E

Why?

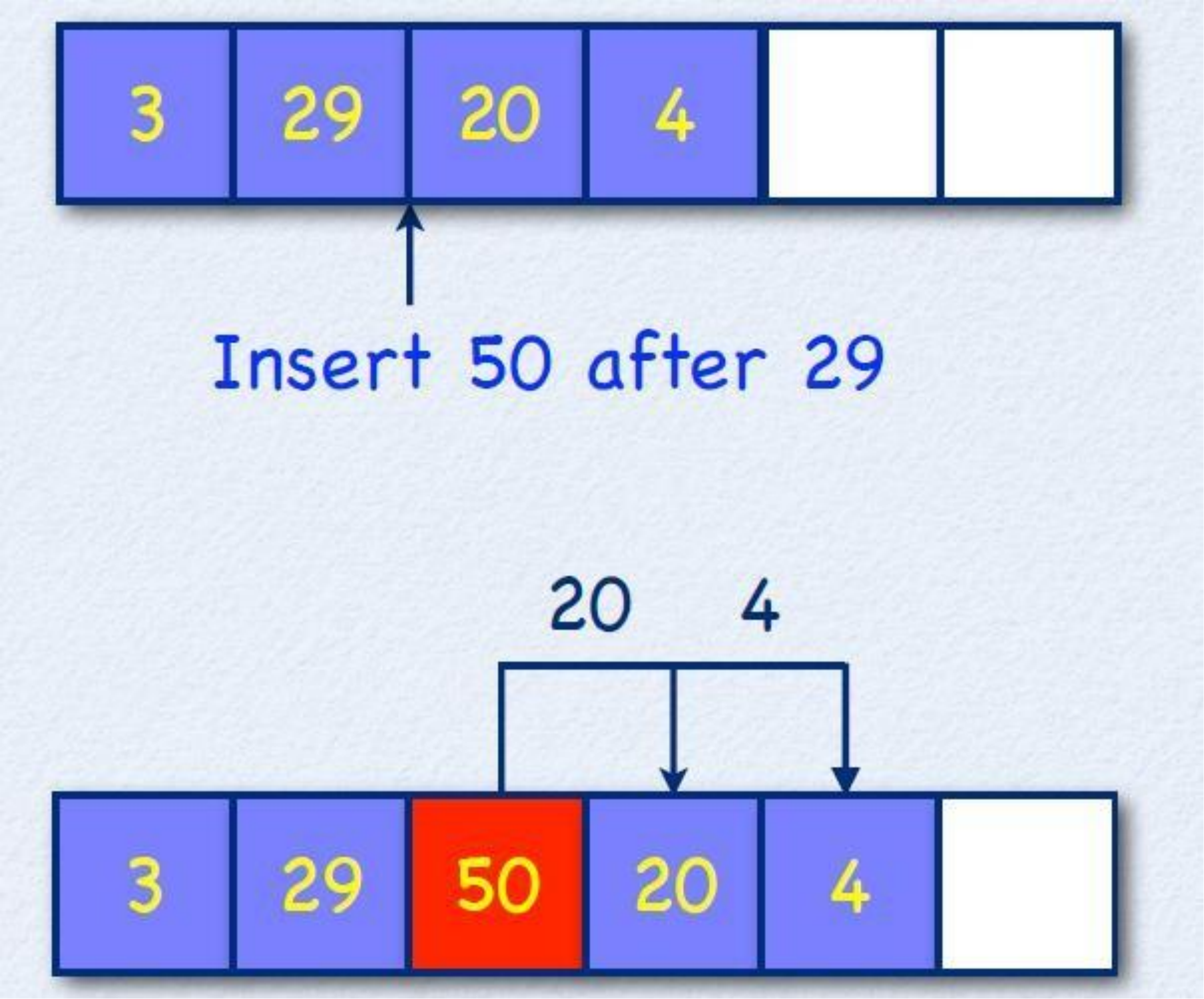
Recursion is an important concept that can be used for linked lists!

Problems with Arrays

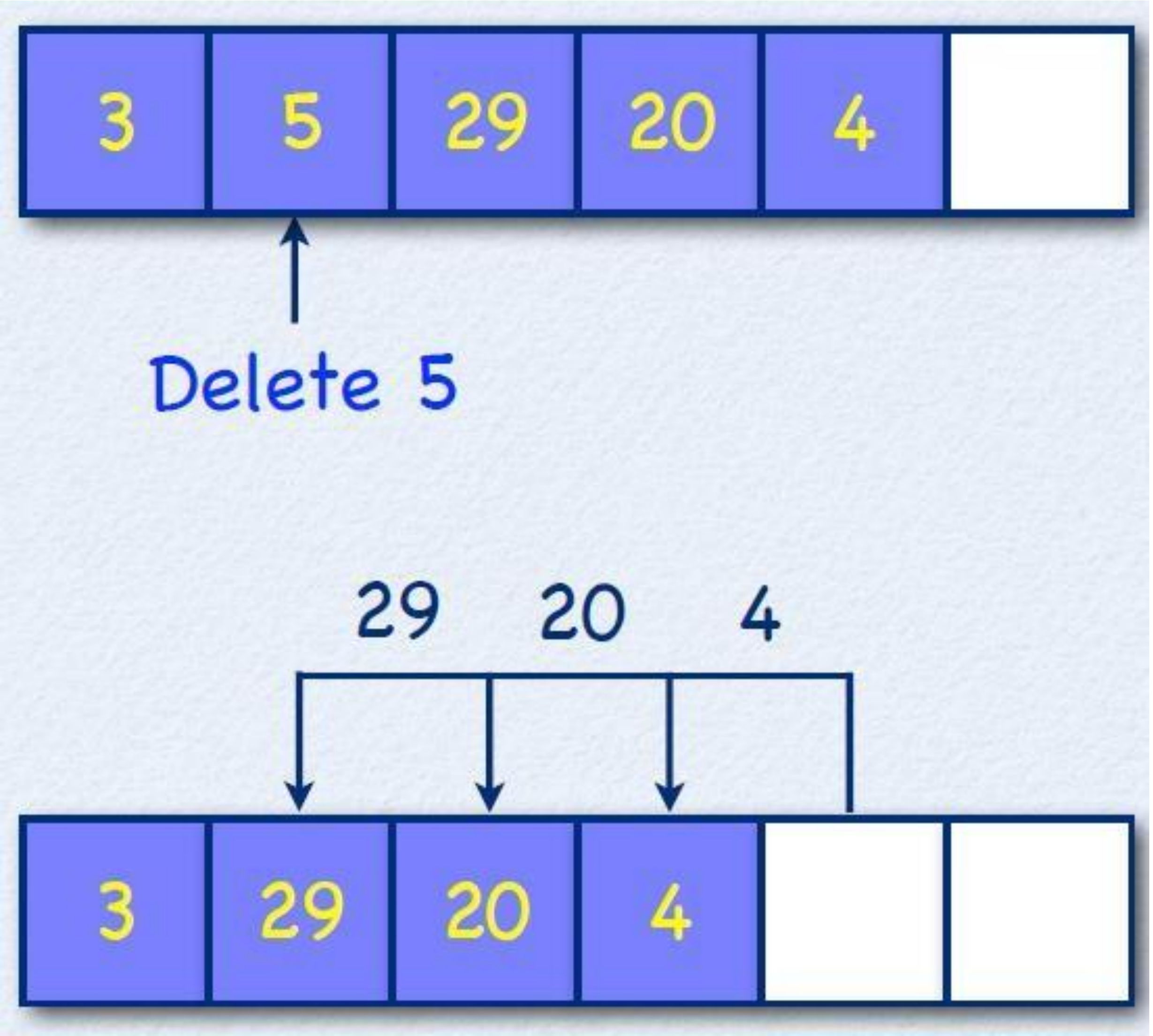
- An array is a contiguous storage that provides insufficient abstractions for handling insertion and deletion of elements.
  - Insertion and deletion require n/2 shifts on average.
  - The computation time is O(n).
  - Resizing affects performance.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Insertion Requires Relocation

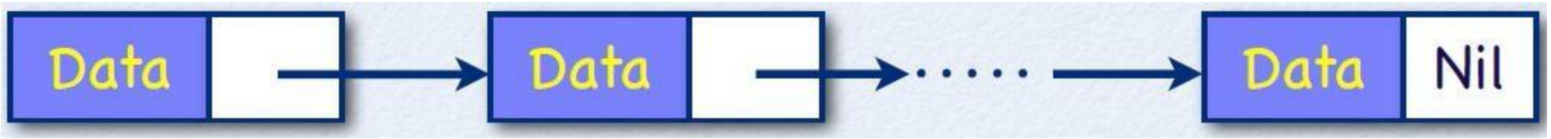


Deletion Requires Relocation



Singly-Linked Lists

- A singly-linked list is a sequence of data items, each connected to the next by a pointer called next.



- A data item may be a primitive type, a composite type, or even another pointer.
- A singly-linked list is a recursive data structure whose nodes refers to nodes of the same type.

List Nodes

- A list manages a collection of elements.
- The class ListNode defines a value-based sequence container for values of type int.

```cpp
class ListNode{

public:
    int item;
    ListNode* next;


    ListNode(){
        item =0;
        next = NULL;
    }

    ListNode(int n){
        item =n;
        next = NULL;

    }

    ListNode(int n, ListNode *p){
        item = n;
        next = p;
    }
};
```

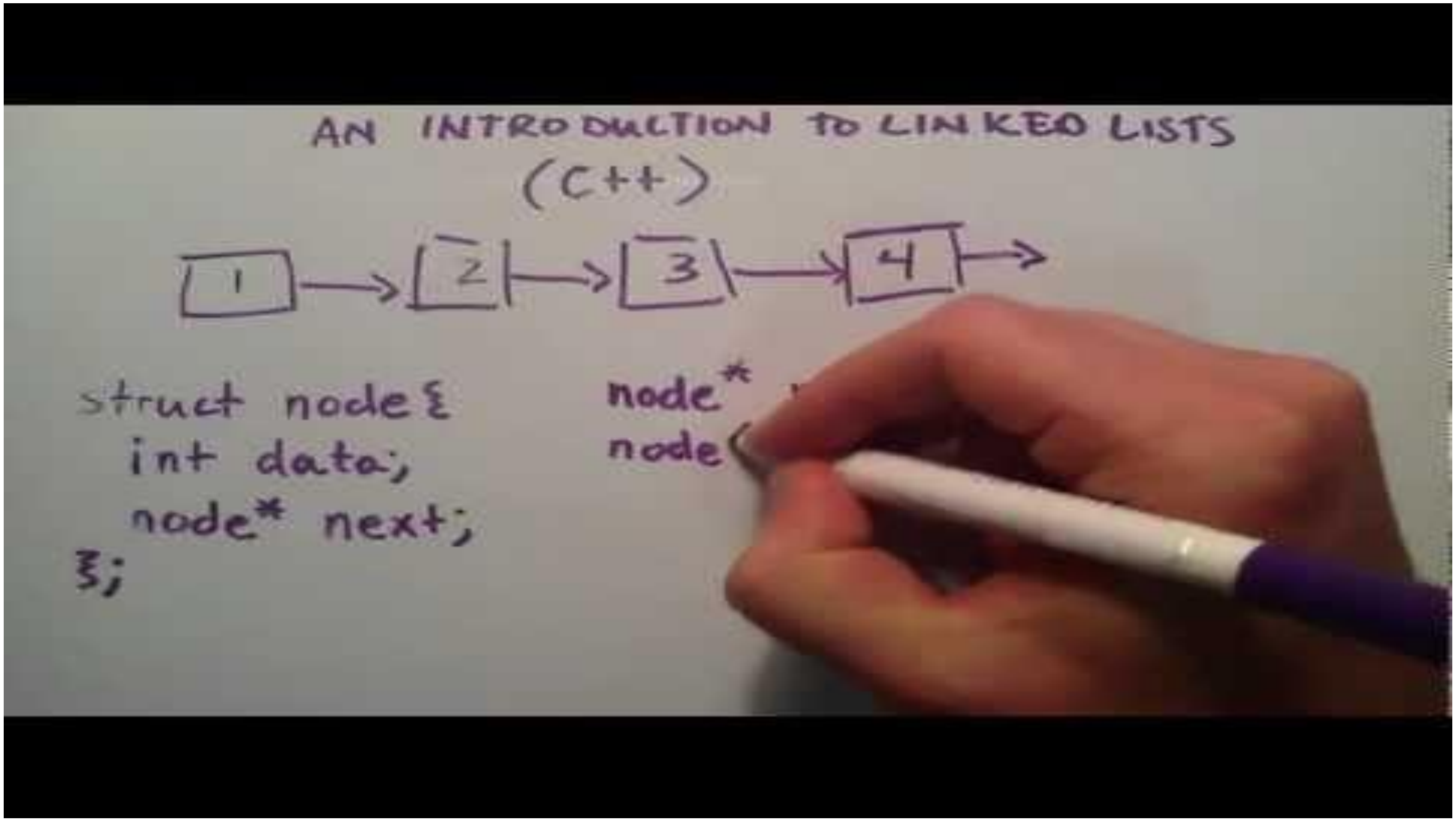
A Simple List of Integers

```cpp
#include<iostream>
#include"llist.h"

using namespace std;

int main(){

    ListNode one(1);
    ListNode two(2, &one);
    ListNode three(3, &two);


    ListNode* lTop=&three;

    while(lTop !=NULL){

        cout<<"value "<<lTop->item<<endl;
        lTop = lTop->next;
    }
    system("pause");

    return 0;
}
```

C:\Users\Pan\Desktop\helpdesk1\llist\Debug\llist.exe

```
value 3
value 2
value 1
Press any key to continue . . .
```

Introduction to Linked Lists



URL: https://www.youtube.com/watch?v=o5wJkJJpKtM

Aliasing

- In programming, two variables are aliased, if the refer to the same entity (aka object).
- If two variables are aliased, then any changes to one variable also effects the other.
- Aliasing can be avoided by using the value-based object model in C++: Every object is unique.
- In languages that rely solely on the reference-based object model (e.g., Java) are prone to aliasing. To avoid it, one needs to clone objects.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Reference

- C++ supports three forms of member variables:
  - values, references, and pointers.
- The use of references and pointers prevents copying the underlying object. But
  - References are aliases to objects that have to reside in a fixed and unique location as long the reference is active (i.e., in use). Creating references to parameters passed as actual values to a member function can violate the uniqueness criterion for locations of references.
  - Objects referenced by pointers can denote both values located on the heap and values located on the stack. Without additional information, we cannot distinguish these cases, which can lead to memory leaks.

Difference between Stack and Heap

- Both Stack and Heap are stored in the Computer's RAM (Random Access Memory)
- **Stack – Is for static memory allocation**
  - When an object is created inside a function without using the "new" operator, the object will be created and stored on the Stack. When the function ends, the object is destroyed as it goes out of scope and will be deleted from the memory.
- **Heap – Is for dynamic memory allocation**
  - When an object is created inside a function using the "new" keyword, the object will be created on the Heap. Data on the heap will remain there until it is manually deleted by the programmer which may result in a memory leak.

Reference URL: https://www.programmerinterview.com/index.php/data-structures/difference-between-stack-and-heap/

Templates

- Templates are blueprints from which classes and/or functions automatically generated by the compiler based on a set of parameters.
- Each time a template is used with different parameters is used, a new version of the class or function is generated.
- A new version of a class or function is called specialization of the template.

Function overloading

- **Function overloading** is a programming concept that allows programmers to define two or more **functions** with the same name. Each **function** has a unique signature.. Note that they have different return values and parameters

```
int max( int a, int b)
{return a<b?b:a;}
```

```
double max(double a, double b)
{ return a<b?b:a}
```

SWINBURNE UNIVERSITY OF TECHNOLOGY

Function template

- “Type substitutes”

```
template <class T>
T max(T a, T b)
{ return a<b ? b:a;}
```

- The symbol T is called a type parameter. It is simply a place holder that is replaced by an actual type or class when the function is invoked.

Template header

- A function template is declared in the same way as an ordinary function, except that it is preceded by the specification
  - template <class T>
- The type parameter T may be used in place of ordinary types within the function definition
- The word class is used to mean a class or primitive type.
  - A template may have several type parameters, specified like this:
    - template<class T, class U, class V>

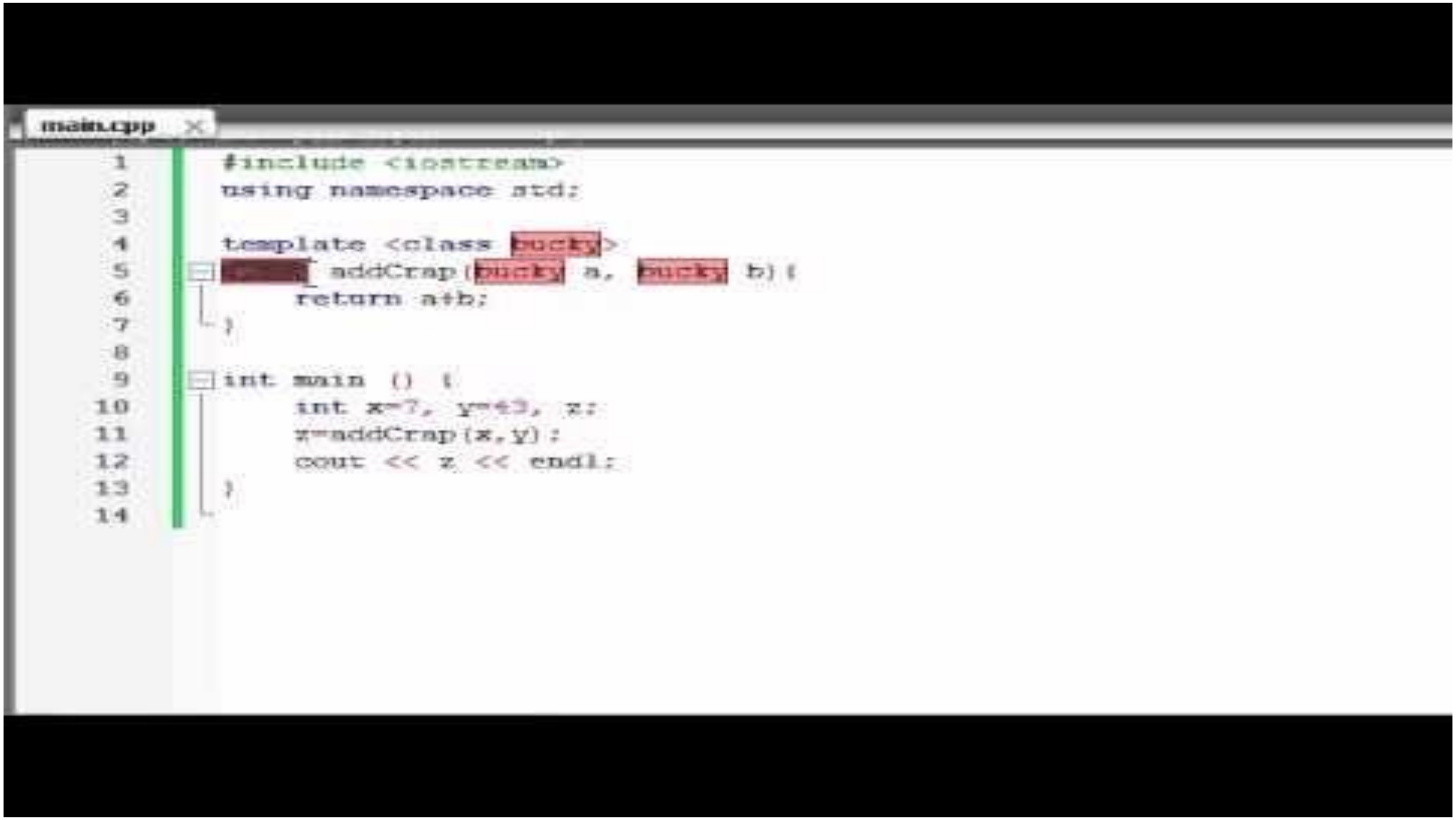
SWINBURNE UNIVERSITY OF TECHNOLOGY

Call a template function

- Template functions are called the same way ordinary function are called.

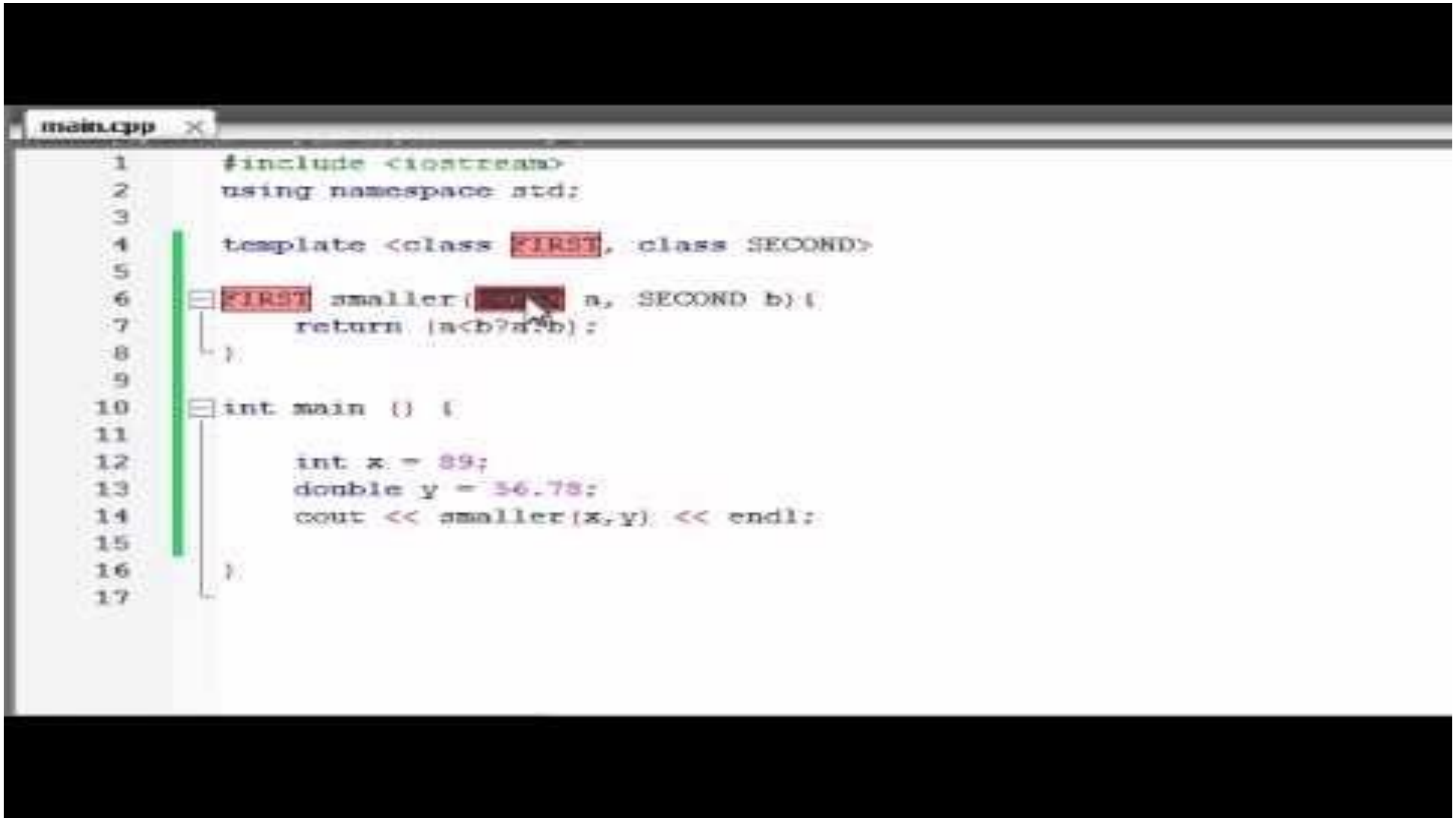
```
int m = 22
int n=66
int max_value;
max_value=max(m, n);
```

Function Templates



URL: https://www.youtube.com/watch?v=W0aoAm6eYSk

Function Templates with Multiple Parameters



URL: https://www.youtube.com/watch?v=SeleR7PDs5Q

What will happen?

```
template <class T>
T max(T a, T b)
{return a < b ? b : a;}
int main(){
  long m =2;
  double n= 4.12;
  cout << max(m,n)
       << endl;
}
```

Would this compile?

It will not. Because the template function expects arguments of the same data type. ‘m’ is long while ‘n’ is double.

What will happen?

```
template <class T, class U>
T max(T a, U b)
{return (a < b ? b : a);}
int main(){
  long m =2;
  double n= 4.12;
  cout << max(m,n)
       << endl;
}
```

Would this compile?

Yes it will. The template here is able to handle arguments of 2 different data types.

Output

Output will be 4. As the output will follow the data type of the first parameter, m.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Class template

- Class template work the same way as a function template except that it generates classes instead of function. The general syntax is

```
template<class T, ...> class X{...}
```

- As with function templates, a class template may have several template parameters. Some of them can be primitive types parameters

```
template<class T, int n, class U> class X{...}
```

Primitive types must be constant

- Template are instantiated at compile time, values passed to the primitive types must be constant

```
template<class T, int n>
class X{ .. }
int main()
{
  X<float, 22> a1;
  const int n =44;
  X<char, n> a2
  int m=66;
  X<short, m>a3   //error (int m is not const)
}
```

Class templates are also known as **parameterised types**.

SWINBURNE UNIVERSITY OF TECHNOLOGY

More on class template

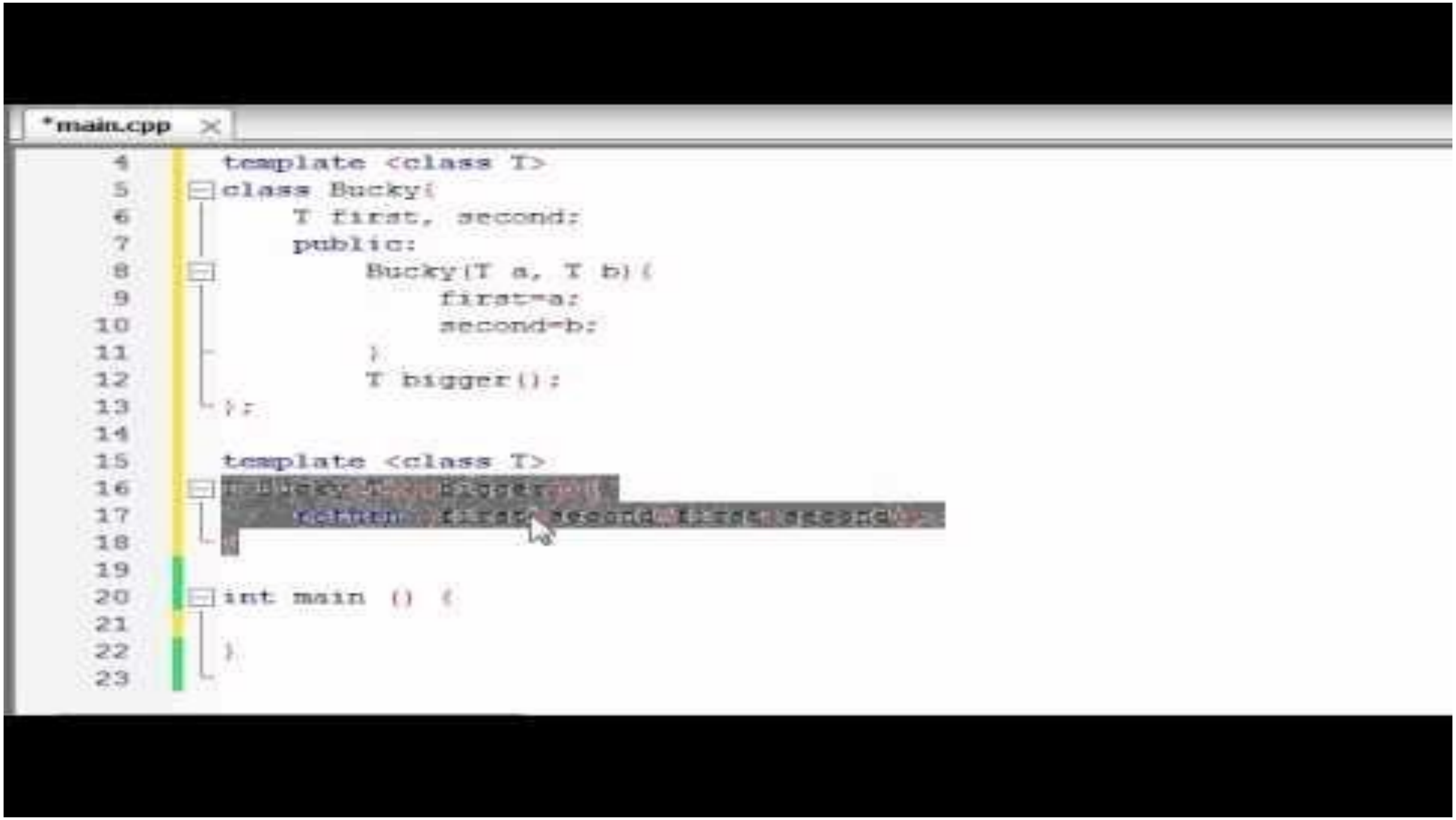
- The member function of a class template are themselves function templates with the same template header as their class

```
template<class T>
class mathFunc{
    static T square(T t){return t*t}
    };
```

is handled in the same way that following template function would be handled

```
template<class T>
T square(T t){return t*t}
```

Class Templates



URL: https://www.youtube.com/watch?v=U2QvTsMvWmM

Node Class template

```
template <class DataType>
class ListNode
{
public:
    DataType fData;
    ListNode* fNext;

public:
    ListNode( const DataType& aData, ListNode* aNext = (ListNode*)0 )
    {
        fData = aData;
        fNext = aNext;
    }
};
```

The New Main

nodes2.cpp

```cpp
#include <iostream>
#include "ListNodeTemplate.h"

using namespace std;

int main()
{
    ListNode<int> One( 1 );
    ListNode<int> Two( 2, &One );
    ListNode<int> Three( 3, &Two );

    ListNode<int>* lTop = &Three;

    while ( lTop != (ListNode<int>*)0 )
    {
        cout << "value " << lTop->fData << endl;
        lTop = lTop->fNext;
    }

    return 0;
}
```

Node Iterator

ListNodeIterator.h

```cpp
#include "ListNodeTemplate.h"

template<class T>
class ListNodeIterator
{
private:
    ListNode<T>* fNode;

public:
    typedef ListNodeIterator<T> Iterator;   // Iterator type definition

    ListNodeIterator( ListNode<T>* aNode );

    const T& operator*() const;
    Iterator& operator++();                 // prefix
    Iterator operator++( int );             // postix (extra unused argument)
    bool operator==( const Iterator& aOther ) const;
    bool operator!=( const Iterator& aOther) const;

    Iterator end();
};
```

Node Iterator Test

ListNodeIteratorTest.cpp

```cpp
#include <iostream>

#include "ListNodeIterator.h"

using namespace std;

int main()
{
    typedef ListNode<int> IntegerNode;

    IntegerNode One( 1 );
    IntegerNode Two( 2, &One );
    IntegerNode Three( 3, &Two );

    for ( ListNodeIterator<int> iter( &Three ); iter != iter.end(); ++iter )
    {
        cout << "value " << *iter << endl;
    }

    return 0;
}
```

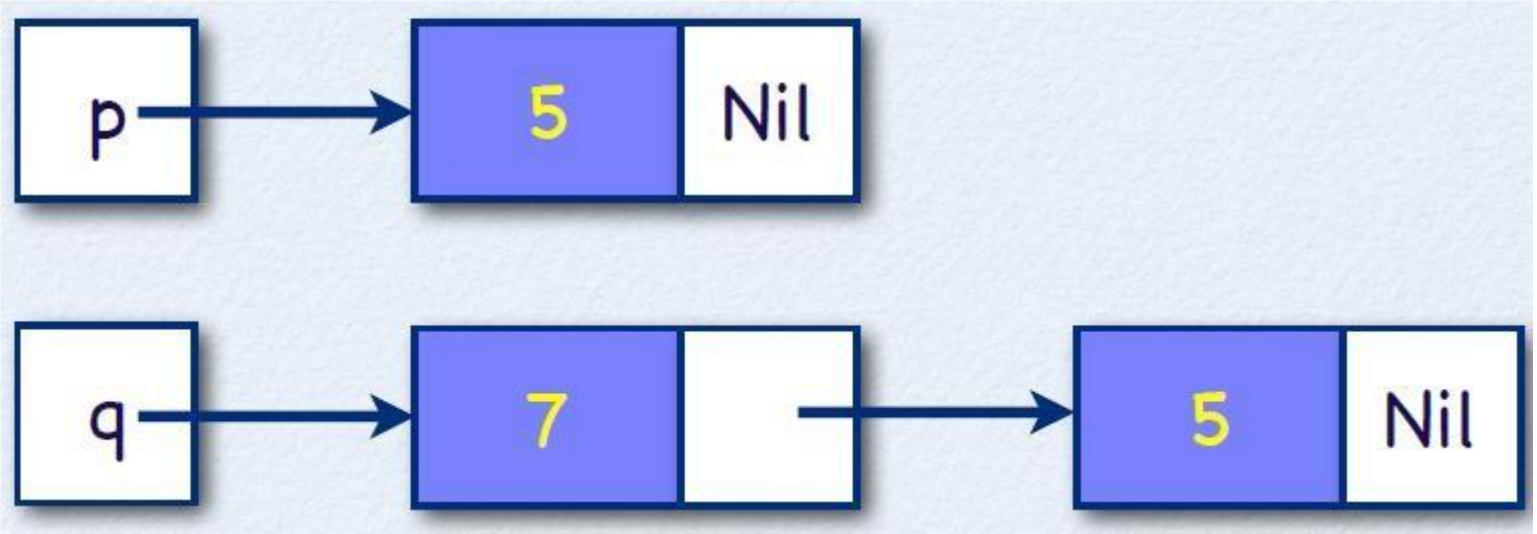
Line: 24 Column: 1 C++ Tab Size: 4 main

The Need for Pointers

- A linked-list is a dynamic data structure with a varying number of nodes.
- Access to a linked-list is through a pointer variable in which the base type is the same as the node
- type:
  - ListNode<int>* pListOfInteger = (ListNode<int>*)0;
  - Here (ListNode<int>*)0 stands for Nil.

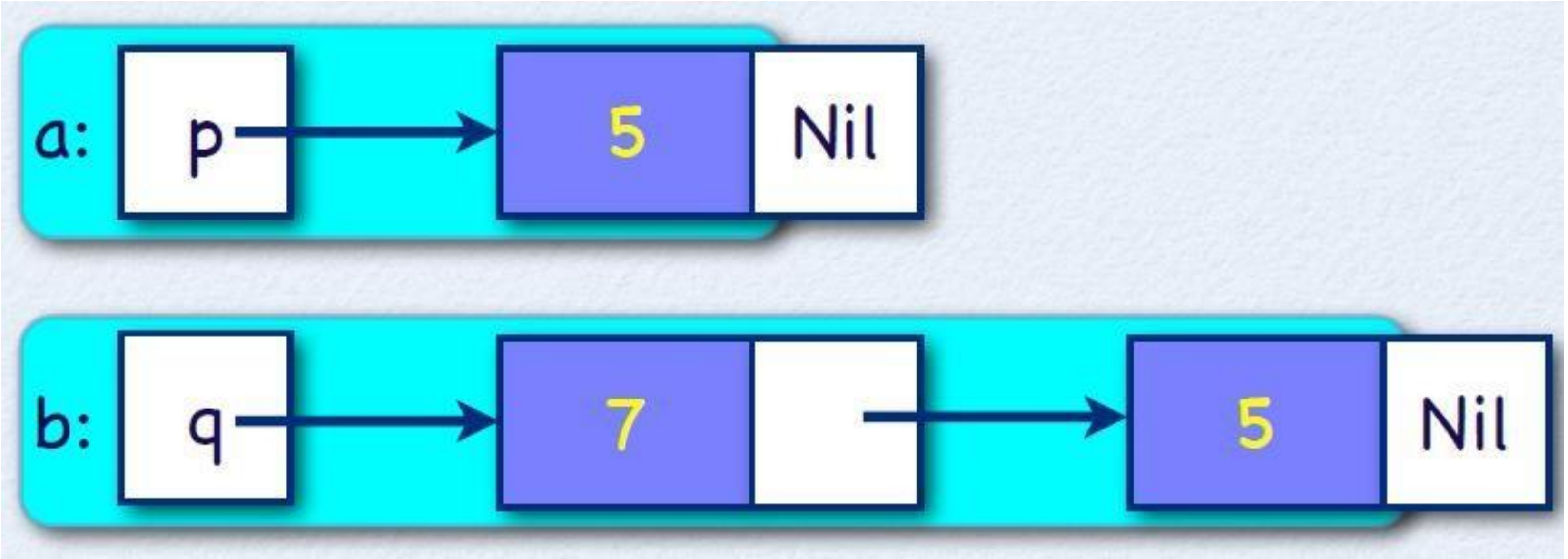
Node Construction

- `IntegerNode *p, *q;`
- `p = new IntegerNode( 5 );`
- `q = new IntegerNode( 7, p );`

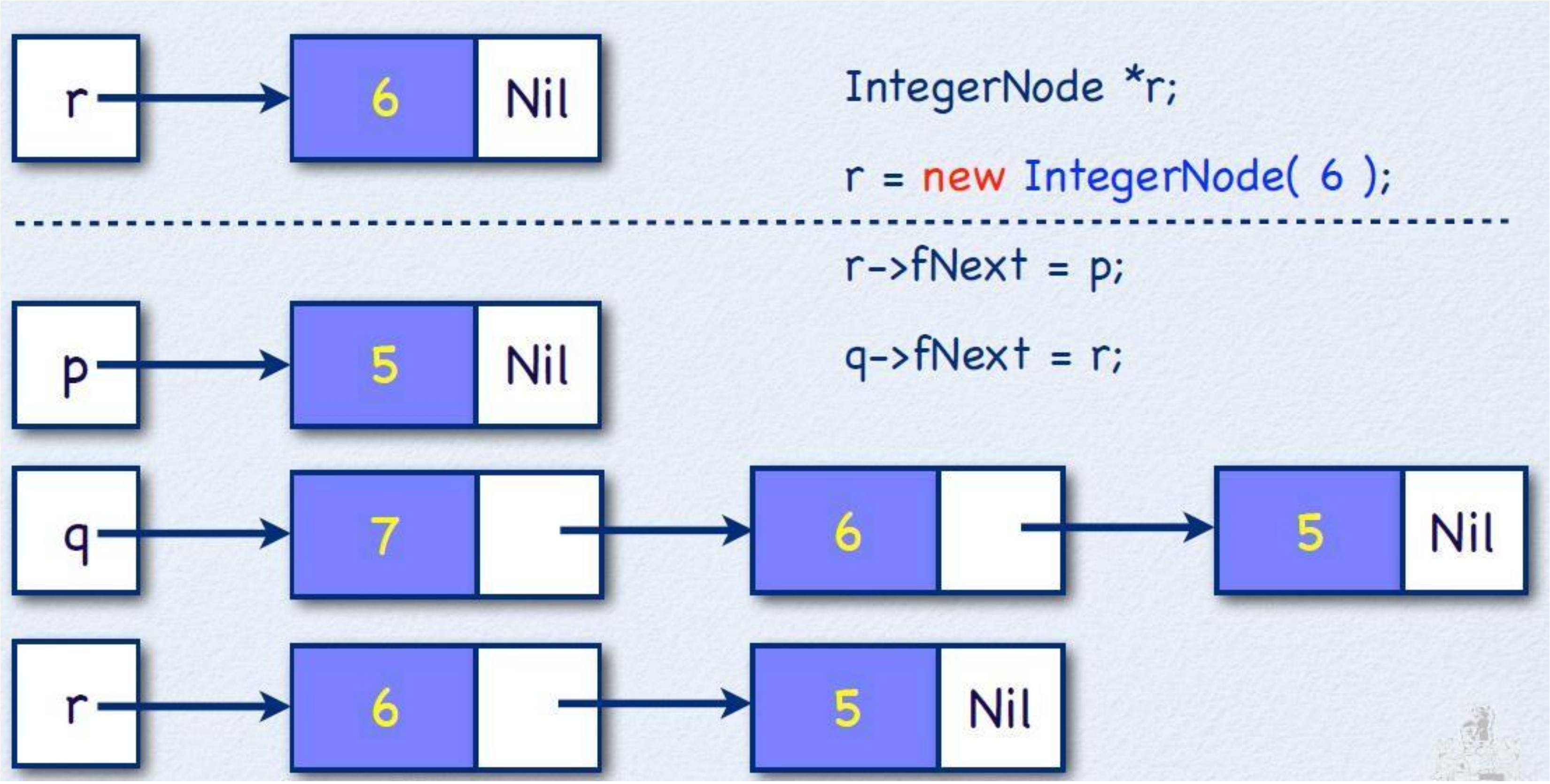


Node Content Access

- `int a = p->fData;`
- `int b = q->fNext->fData;`

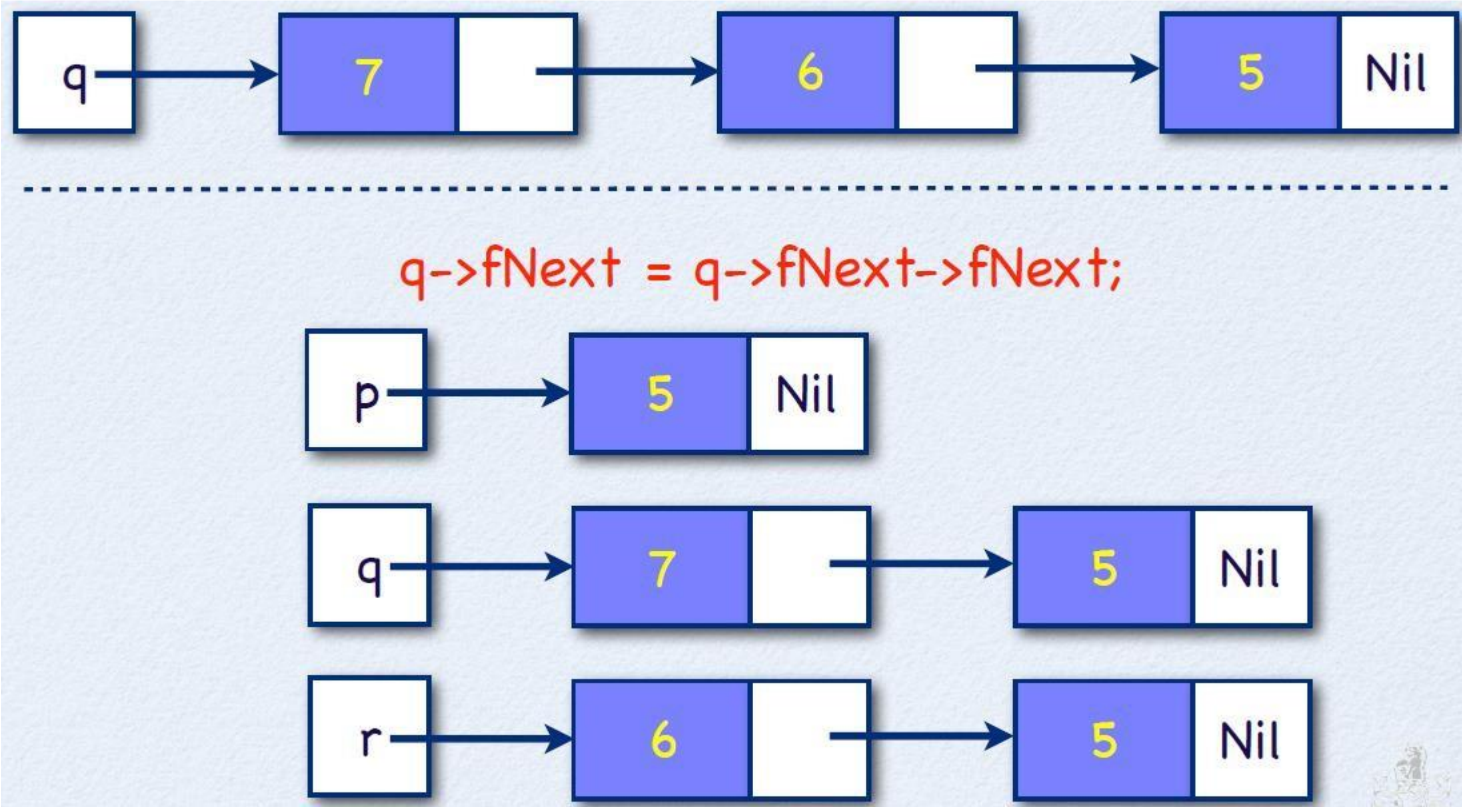


Inserting a Node



SWINBURNE UNIVERSITY OF TECHNOLOGY

Deleting a Node

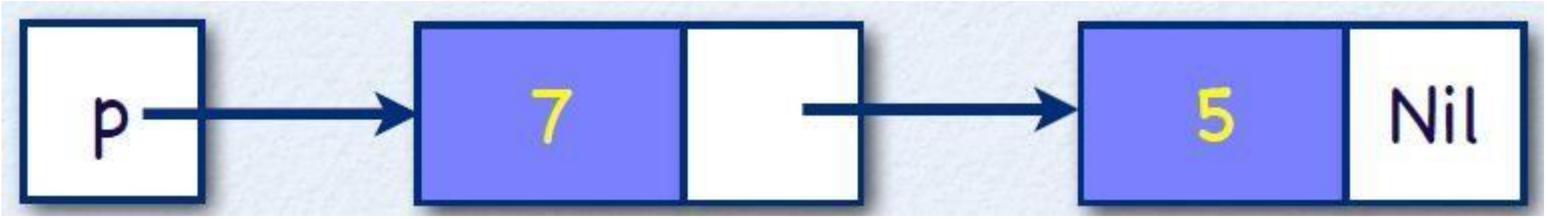


SWINBURNE UNIVERSITY OF TECHNOLOGY

Insert at the Top(at the end)

- IntegerNode *p = (IntegerNode*)0;
- p = new IntegerNode(5, p);

- p = new IntegerNode(7, p);

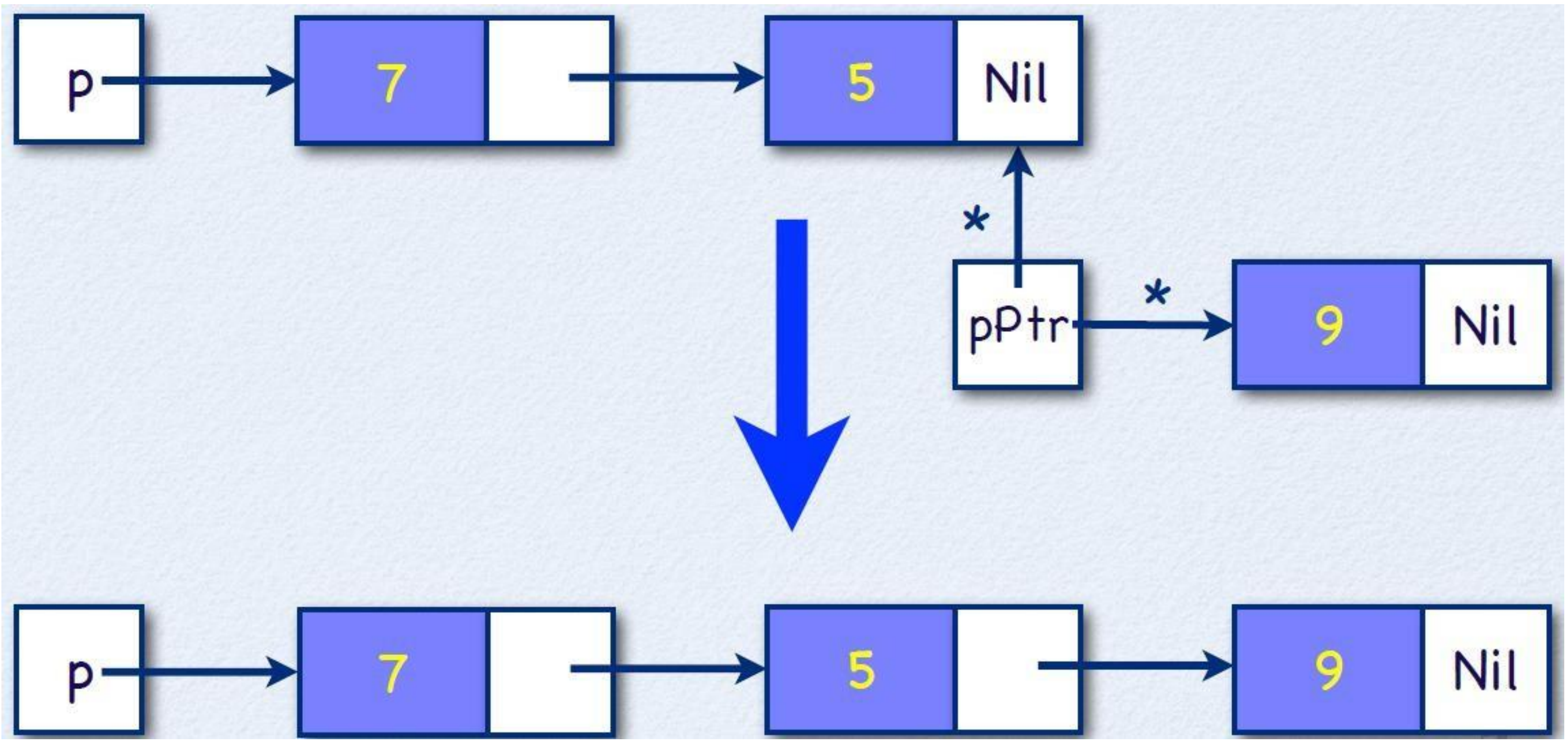


Insert at the Beginning

- To insert a new node at the beginning of a linked list we need to search for the beginning:

```
IntegerNode *p = (IntegerNode*)0;
IntegerNode **pPtr = &p;

while ( *pPtr != (IntegerNode*)0 )
    pPtr = &((*pPtr)->fNext);    // pointer to next

*pPtr = new IntegerNode( 9 );
```

Insert at the Beginning



End of Recursions and Linked List