These notes relate to chapter 17 of PPP using C++

Written by Robert J Baldwin 16/05/2023

**Part 1:**

**STEP 1:**

**CODE: Link\* norse\_gods = new Link("Thor");**

1.1: Create a Link pointer called norse\_gods.

1.2: Create new Link object on the heap with data value “Thor”. “Thor’s” prev and succ pointers are set to nullptr by default.

1.3: Link pointer norse\_gods points to address of “Thor” object. Let’s say the memory address is 67.

**STEP 2:**

**CODE: norse\_gods = insert(norse\_gods, new Link{ "Odin" });**

2.1: Call the insert() function: Link\* insert(Link\* p, Link\* n) // insert n before p

Insert() takes two arguments; Link\* p and Link\* n. Argument 1 points to the address that norse\_gods currently points to. We called this 67, or “Thor”.

2.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Odin”. Its prev and succ are set to nullptr by default. Lets say the memory address is 78.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 78, or “Odin”.

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to norse\_gods pointer which is pointing to 67, or “Thor”.

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed – norse\_gods, which is 67 “Thor” ) to n’s succ (the second argument passed – 78 “Odin”). Now, “Odin” 78’s succ pointer points to p (norse\_gods 67 “Thor”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (norse\_gods – “Thor” 67), point p’s prev’s succ to the address of n. At the moment, “Thor’s” prev is null so it does not point to a succ. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“odin” 78) previous points to p’s (norse\_gods – “Thor” 67’s) prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev(“Thor”67’s p) now points to n (“Odin” 78)

**CODE:** return n;

Address of n (78 “Odin”) retuned. Now, norse\_gods points to the address 78 or “Odin”

**STEP 3:**

**CODE:** norse\_gods = insert(norse\_gods, new Link{ "Zeus" });

3.1: The insert() function is called and again: Link\* insert(Link\* p, Link\* n) // insert n before p

Two arguments are passed. Norse\_gods is passed to the first argument (p). norse\_gods currently points to 78 “odin”. insert(norse\_gods == 78 “odin”, new Link{ "Zeus" });

3.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Zeus”. Its prev and succ are set to nullptr by default. Lets say the memory address is 12.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 12, or “Zeus”

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to norse\_gods pointer which is pointing to 78, or “Odin”.

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed – norse\_gods, which is 78 “Odin” ) to n’s succ (the second argument passed – 12 “Zeus”). Now, 12 “Zeus” succ pointer points to address p (norse\_gods, which is pointing to 78 “Odin”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (norse\_gods – 78 “odin”) point p’s prev’s succ to the address of n. At the moment, “Odin” prev is null so it does not point to a succ. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“Zeus” 12) previous points to p’s (norse\_gods – “Odin” 78) prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev (“Odin” 78) now points to n (“Zeus” 12)

**CODE:** return n;

Address of n (12 “Zeus”) retuned. Now, norse\_gods points to the address 12 or “Zeus”

**STEP 4:**

**CODE:** norse\_gods = insert(norse\_gods, new Link{ "Freia" });

4.1: The insert() function is called and again: Link\* insert(Link\* p, Link\* n) // insert n before p

Two arguments are passed. Norse\_gods is passed to the first argument (p). norse\_gods currently points to 12 “Zeus”. insert(norse\_gods == 12 “Zeus”, new Link{ "Freia" });

4.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Freia”. Its prev and succ are set to nullptr by default. Lets say the memory address is 100.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 100, or “Freia”

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to norse\_gods pointer which is pointing to 12, or “Zeus”.

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed – norse\_gods, which is 12 “Zeus” ) to n’s succ (the second argument passed – 100 “Freia”). Now, 100 “Freia” succ pointer points to address p (norse\_gods, which is pointing to 12 “Zeus”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (norse\_gods – 12 “Zeus”) point p’s prev’s succ to the address of n. At the moment, “Zeus” prev is null so it does not point to a succ. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“Freia” 100) previous points to p’s (norse\_gods – “Zeus” 12) prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev (“Zeus” 12) now points to n (“Freia” 100)

**CODE:** return n;

Address of n (100 “Freia”) retuned. Now, norse\_gods points to the address 100 or “Freia”

**Part 2:**

**STEP 5:**

**CODE:** Link\* greek\_gods = new Link("Hera");

5.1: Create a Link pointer called Greek\_gods

5.2: Create new Link object on the heap with data value “Hera” “Hera’s” prev and succ pointers are set to nullptr by default.

5.3: Link pointer greek\_gods points to address of “Hera” object. Let’s say the memory address is 1050.

**STEP 6:**

**CODE:** greek\_gods = insert(greek\_gods, new Link{ "Athena" });

6.1: Call the insert() function: Link\* insert(Link\* p, Link\* n) // insert n before p

Insert() takes two arguments; Link\* p and Link\* n. Argument 1 points to the memory address that greek\_gods currently points to. We called this 1050, or “Hera”

6.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Athena”. Its prev and succ are set to nullptr by default. Lets say the memory address is 1176.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 1176, or “Athena”.

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to greek\_gods pointer which is pointing to 1050, or “Hera”

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed - greek\_gods which is 1050 “Hera” ) to n’s succ (the second argument passed – 1176 “Athena”). Now, “Athena” 1176’s succ pointer points to p (greek\_gods 1050 “Hera”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (greek\_gods – “Hera” 1050), point p’s prev’s succ to the address of n. At the moment, “Hera” prev is null so it does not point to a succ. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“Athena” 1176) previous now points to p’s (greek\_gods – “Hera” 1050), prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev(“Hera” 1050’s p) now points to n (“Athena” 1176)

**CODE:** return n;

Address of n (1176 “Athena”) retuned. Now, greek\_gods points to the address 1176 or “Athena”.

**STEP 7:**

**CODE:** greek\_gods = insert(greek\_gods, new Link{ "Mars" });

7.1: Call the insert() function: Link\* insert(Link\* p, Link\* n) // insert n before p

Insert() takes two arguments; Link\* p and Link\* n. Argument 1 points to the memory address that greek\_gods currently points to. We called this 1176, or “Athena”

7.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Mars”. Its prev and succ are set to nullptr by default. Lets say the memory address is 66.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 66 or “Mars”.

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to greek\_gods pointer which is pointing to 1176, or “Athena”

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed - greek\_gods which is 1176 “Athena” ) to n’s succ (the second argument passed – 66 “Mars”). Now, “Mars” 66’s succ pointer points to p (greek\_gods 1176 “Athena”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (greek\_gods – “Athena” 1176’s p), point p’s prev’s to the address of n. At the moment, “Athena’s” P is null so condition not met. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“Mars” 66) previous now points to p’s (greek\_gods – “Athena” 1176), prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev(“Athena” 1176’s p) now points to n (“Mars” 66)

**CODE:** return n;

Address of n (66 “Mars”) retuned. Now, greek\_gods points to the address 66 or “Mars”.

**Step 8:**

**CODE:** greek\_gods = insert(greek\_gods, new Link{ "Poseidon" });

8.1: Call the insert() function: Link\* insert(Link\* p, Link\* n) // insert n before p

Insert() takes two arguments; Link\* p and Link\* n. Argument 1 points to the memory address that greek\_gods currently points to. We called this 66, or “Mars”

8.2: The second argument points to a new Link object created on the heap. This link object has been assigned the data value “Poseidon”. Its prev and succ are set to nullptr by default. Lets say the memory address is 21.

What happens inside the function with these two arguments passed?

**CODE: if (n == nullptr) return p;**

The first instruction tests if n (the second argument passed) is == to nullptr. It is not, because it’s pointing to 21 or “Poseidon”.

**CODE: if (p == nullptr) return n;**

The second instruction tests if p (the first argument passed) is == to nullptr. It is not, because it’s pointing to greek\_gods pointer which is pointing to 66, or “Mars”

**CODE: n->succ = p;** // p comes after n

Assign p (the first argument passed - greek\_gods which is 66 “Mars” ) to n’s succ (the second argument passed – 21 “Poseidon”). Now, “Poseidon” 21’s succ pointer points to p (greek\_gods 66 “Mars”)

**CODE: if (p->prev)p->prev->succ = n;**

If p has a prev (greek\_gods – “Mars” 66’s p), point p’s prev’s to the address of n. At the moment, “Mars’s” P is null so condition not met. Skips this step.

**CODE:** n->prev = p->prev; // p's prev becomes n's prev

n’s (“Poseidon” 21) prev now points to p’s (greek\_gods – “Mars” 66), prev (nullptr)

**CODE:** p->prev = n; // n becomes p's prev

Ps prev(“Mars” 66’s p) now points to n (“Poseidon” 21)

**CODE:** return n;

Address of n (21 “Poseidon”) retuned. Now, greek\_gods points to the address 21 or “Poseidon”.

**Part 3:**

The next part of the programme solves the problem of Zeus being linked into the norse\_gods list.

**Step 9:**

**CODE:** Link\* p = find(norse\_gods, "Zeus");

9.1: A new Link pointer called P is called.

9.2: The find function is called taking 2 arguments.

What happens inside the function when these two arguments are passed?

**Function Code:**

Link\* find(Link\* p, const string& s) // find s in list;

// return nullptr for “not found”

{

while (p) {

if (p -> value == s) return p;

p = p -> succ;

}

return nullptr;

}

**Breakdown:**

**CODE:** find(Link\* p, const string& s)

9.3: The address that norse\_gods list is currently pointing to is passed in and given the variable p.

9.4: The string value “Zeus” is passed in as the second argument as s.

**CODE:** while (p)

9.5: While p (norse\_gods) is present/true.

**CODE:** if (p -> value == s) return p;

Search for p’s value and check for equality against string s, which has the value “Zeus”. If there’s a match, return the address of p where s is associated.

**CODE:** p = p -> succ;

9.6: This code is skipped because the if statement was passed, so a return value of “zeus” 12 is returned from the function.

9.7: Link\* p is now pointing to “Zeus” 12.

if (p) {

if (p == norse\_gods) norse\_gods = p -> succ;

erase(p);

greek\_gods = insert(greek\_gods, p);

}

**Checking if a node was found**: The code checks if the value of p is not nullptr, which means the node with the value "Zeus" was found in the "norse\_gods" list.

**Moving the node between lists**: If the node was found (p is not nullptr), the following steps are executed:

* Check if p is the first node in the "norse\_gods" list by comparing it with norse\_gods.
* If p is indeed the first node (p == norse\_gods), update the "norse\_gods" pointer to point to the next node (p->succ), effectively removing p from the list.
* Call the erase function to remove the node p from its current list. The erase function adjusts the connections of the neighbouring nodes accordingly and returns the successor of p in the list.
* Call the insert function to insert the node p into the "greek\_gods" list. The insert function adjusts the connections of the neighbouring nodes and returns the new beginning of the list.
* Assign the return value of insert back to the "greek\_gods" pointer to update the head of the list.

Erase()

**CODE:** Link\* erase(Link\* p) // remove \*p from list; return p’s successor

{

if (p == nullptr) return nullptr;

if (p -> succ) p -> succ -> prev = p -> prev;

if (p -> prev) p -> prev -> succ = p -> succ;

return p -> succ;

}