

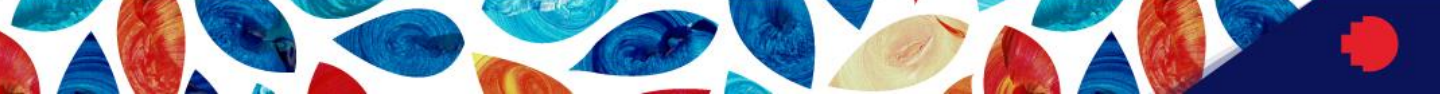


C++ Programming Bootcamp 2

COSC2802

Topic 9 Pointers II





Day 10 Workshop Overview

1. Program Memory Management
2. Dynamic Memory Management
3. Objects | Memory Management
4. Abstract Data Types and Linked Lists (first look – revisit in PS2)



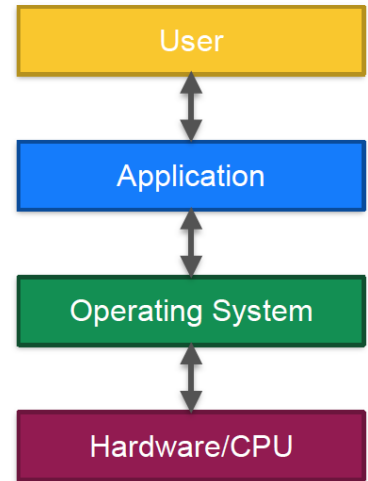
Program Memory Management



How do programs manage memory?

- ▶ We have informally discussed things like:
 - The program “setting aside memory”
 - Declaring a variable, “creates memory”
 - Seen array overflow, that is, “reading beyond the end of an array”
 - Mentioned that the “operating system does stuff”

- ▶ The question, is what is happening?



Application Structure

- ▶ There are two general components to an application/program that we are concerned with:
 - 1 Program code loaded into computer memory
 - 2 Allocated memory for storing program data, such as
 - Variables
 - Function parameters
 - Program control information


(1) Program Code

- Each line of code is converted into assembly instructions
- An assembly instruction is a single operation that the CPU can execute
- To run a program, the assembly instructions are loaded into memory. Thus:
 - every instruction has an associated memory address
- Operating system uses an *instruction pointer* (memory address):
 - to track the instruction in a program that it is up to

(2) Memory Structure

- ▶ In a typical program, memory is managed in two forms
 - Automatic memory allocation
 - Dynamic, programmer controlled, memory allocation
- ▶ Automatic memory allocation is managed through the programming language compiler (C/C++) or interpreter (Java)
 - The typical method for this is the **Program (or Call) Stack** **a**
- ▶ Dynamic memory allocation, is maintained by the programmer through the programming language
 - This is typically done on the **Heap** (or **Free Store**) **b**
- ▶ Operating Systems and CPUs provide

Instruction Pointer

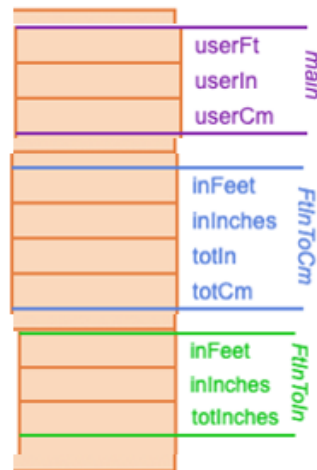


0x004a	ADD x y
0x004b	SUB x b
0x004c	JUMP 0x004a
...	...

(a) Program/call stack

- The C++ compiler automatically handles allocating and de-allocating of memory for variables and function calls.
- The compiler generates CPU instructions for memory management
 - A data structure called a **stack**

```
int FtInToIn(int inFeet, int inInches) {  
    int totInches;  
    ...  
    return totInches;  
}  
  
double FtInToCm(int inFeet, int inInches) {  
    int totIn;  
    double totCm;  
    ...  
    totIn = FtInToIn(inFeet, inInches);  
    ...  
    return totCm;  
}  
  
int main() {  
    int userFt;  
    int userIn;  
    int userCm;  
    ...  
    userCm = FtInToCm(userFt, userIn);  
    ...  
    return 0;  
}
```



Stack Frame



(a) Program/call stack

- ▶ The C++ compiler automatically handles allocating and de-allocating of memory for variables and function calls.
 - The compiler generates CPU instructions for memory management
 - A data structure called a *stack*
- ▶ In the stack
 - As memory is required, a block (of the correct size) is allocated by being *pushed* onto the stack
 - Once memory is no-longer required, blocks are de-allocated by being *popped* off the stack
 - This forms an ordered structure, using FILO (first-in, last-out)
 - That is, the first allocated block is the last to be de-allocated



(a) Program/call stack

- ▶ The operating system determines and manages the location of the program stack in physical memory. The OS may
 - Limit the total size of the stack
 - Randomly position the stack
 - Clear (set to zero) all stack memory locations



(b) Heap (Free Store)

- ▶ The programmer manages the allocation and de-allocation of memory on the *heap*
- ▶ In Java, objects are allocated on the heap
 - `Object obj = new Object();`
 - The “new” keyword creates the object
 - The object is stored on the heap



Heap | Allocation and De-allocation

- ▶ Any memory that is allocated, should be **de-allocated**
 - Also called **“freeing”** or **“deleting”** memory
 - By de-allocating memory, a program can re-use it for another purpose
 - If memory is not “cleaned-up”, the Operating System is not aware that memory is no longer needed
 - Any new memory that a program requires, must be allocated elsewhere
- ▶ Java has automated garbage collection, so the programmer does not have to be concerned with de-allocating the memory
 - In C++, **You (the programmer) MUST** de-allocate all memory
- ▶ Only when a program is terminated, will the operating system reclaim all memory that was not de-allocated

Example

See: zybooks

```
#include <iostream>
using namespace std;

// Program is stored in code memory

int myGlobal = 33;    // In static memory

void MyFct() {
    int myLocal;      // On stack
    myLocal = 999;
    cout << " " << myLocal;
}

int main() {
    int myInt;         // On stack
    int* myPtr = nullptr; // On stack
    myInt = 555;

    myPtr = new int;    // In heap
    *myPtr = 222;
    cout << *myPtr << " " << myInt;
    delete myPtr; // Deallocated from heap

    MyFct(); // Stack grows, then shrinks

    return 0;
}
```

Code memory

1	Add R1, #1, R2
2	Sub R3, #1, R4
3	Add R1, R3, R5
4	Jmp 40

Static memory

3000	33	myGlobal
3001		

Stack

3200	555	myInt	<div><div></div><div>main()</div><div></div><div></div></div>	
3201	9400	myPtr		
3202	999	myLocal		<div><div></div><div>MyFct()</div><div></div></div>
3203				

Heap

9400	222
9401	
9402	



Call Stack vs Heap

Generally, the heap is significantly larger than the call stack

- Call stack should be used to store small, short-lived, scoped data
- Heap is good for storing data that is not bounded within the scope of a single function or method

In Java:

- Objects are placed on the Heap
- Local variables are placed on the stack

In C++ objects can be allocated memory

- either on Stack or on Heap (depending on the scope and ownership of the object).

Key point: the stack is statically allocated (calculated at compile time); you cannot load dynamic amounts of data on the stack.

So, if you're reading from file and you don't know how large the data is, you will need to use the heap somehow!



Examples when to allocate on Heap

In General:

- Creating objects with lifetime extending beyond the function/block in which it is created:
 - duration of the object is between new and delete
- Creating an object that is very large (avoid stack overflow)
- Creating object whose size isn't known until run-time (not known at compile time)
- ...



Dynamic Memory Management





Dynamic Memory Management

- Can control *allocation* and *deallocation* of memory on the heap for:
 - objects and for arrays of any built-in or user-defined type
 - Use the operators *new* and *delete*
- Dynamic Memory Management is about managing memory on the heap
- The key principle: ***Everything you create, you must delete***
- If you don't: MEMORY LEAKS!

Allocating Memory on the Heap

► In C/C++ memory is allocated on the heap using the 'new' keyword

```
int* a = new int;
```

Returns a
pointer

"new" memory

Type of memory
to generate

- The "new" returns a pointer to the allocated memory
- The allocated memory is not initialised
 - This can be done separately
- The compiler/OS will set aside enough memory based on the type

▪ new keyword (zybooks 23.1)

Allocating Memory on the Heap

► The allocated memory may also be initialised inline, using “bracket” notation

```
int* a = new int(7);
```



Initialise

- Use of this is dependent on the type
 - Some types do not support inline allocation and initialisation.
- As with all other memory, the allocated memory must be initialised!

Deleting Memory on the Heap

- Memory is de-allocated using the “delete” keyword

```
delete a;
```

“delete” memory

delete on pointer

Delete is called on the pointer (not the value, so don't dereference it).

Once memory has been deallocated **it should not be used.**

To ensure this once memory has been deleted you must set all pointers to that memory location to *nullptr*, **if you don't you may get segmentation faults on accident!**

‘delete’ keyword (zybooks 23.1)



Primitive Types

- ▶ All primitive types can be allocated using the type keyword.

```
int* a = new int;  
double* d = new double;  
char* c = new char;
```

- ▶ Primitive types can be initialised inline.

```
int* a = new int(7);  
double* d = new double(7.5);  
char* c = new char('a');
```

- ▶ Delete operates as already shown

```
delete a;
```



References and Memory Management

This is typically not done, but it is also possible to use references to manage memory:

```
int& mref = *(new int(4)) ;  
std::cout << mref << std::endl ;  
delete &mref ;
```



Arrays

- ▶ Arrays can be allocated using square brackets

```
int* array = new int[LENGTH];
```

- ▶ Arrays cannot be initialised inline
- ▶ A special delete operator is required

```
delete[] array;
```

- This does not need to be given the length
 - This is because of how “`new[]`” is implemented in C++
 - Strictly this is an operator



Multidimensional Arrays

Allocating multidimensional arrays are a pain!

You cannot declare them in one simple line, they require a loop to allocate each internal array separately.

```
int** arr = new int*[ROWS];  
for(int i = 0; i < ROWS; ++i) {  
    arr[i] = new int[COLS];  
}
```

Deleting them is also similarly painful.

```
for(int i = 0; i < ROWS; ++i) {  
    delete [] arr[i];  
}  
delete [] arr;
```

NOTE: `std::vectors` handle all of this for you! Use them!



Objects | Memory Management





Creating objects in C++

There are different ways to create objects in C++, each with its own purpose and implications:

1. **Automatic storage duration:** create objects on the stack *without using new*. For example: Seen this earlier

```
MyClass obj; // Object created on the stack
```

2. **Dynamic memory allocation:** create objects on heap and control manually using *new*. For example:

```
MyClass* ptr = new MyClass();           // Object created on the heap
// ...
delete ptr;                             // Explicitly deallocate the memory
```

Note: everything you use *new* to allocate memory you must explicitly deallocate to avoid memory leaks.

3. **Smart pointers:** provided in modern C++ and which handle memory deallocation for you

Programming Studio 2



Objects | Allocating

- Allocating memory for a Class *creates an object* of that class

```
Example* ex = new Example(10);
```

- Creating an object calls a constructor of the class

- A constructor must always be called
- Even if that constructor is empty (takes no parameters)

```
Example* ex = new Example();
```

- If a class defines no constructors, C++ will generate a default empty constructor

Objects | deallocating (zybooks)

When an object is deleted (or moved out of scope), a special method is called:

- The destructor! Which is denoted by a tilde (~)
- The destructor cleans up all memory, objects, or entities that are used by the object.
- Meaningfully, this means deleting any memory that the object allocated to the heap

```
class Example {  
public:  
    Example(int value);  
    ~Example();  
};  
  
Example::~~Example() {  
    // cleanup  
}
```

`Example* ex(10);`
`delete ex;`

← Destructor

← Destructor Implementations

Calls destructor

Note: if no destructor defined, compiler automatically generates 'default'

- **No** deallocation of dynamically allocated memory

Example Destructor

```
class Example
{
public:
    // Constructor
    Example()
    {
        std::cout << "Constructor called" << std::endl;
        data = new int[10]; // Dynamically allocate memory
    }

    // Custom destructor
    // WITHOUT THIS results in memory leak as NO deallocation of the int array
    ~Example()
    {
        std::cout << "Destructor called" << std::endl;
        delete[] data; // Deallocate the dynamically allocated memory
    }

private:
    int *data;
};

int main()
{
    Example *example_object = new Example(); // Create a dynamically allocated object
    // Do something with the dynamically allocated object
    delete example_object; // Deallocate the dynamically allocated object
}
```

Also see: [zyBooks example \(Linked List\)](#)



Copy Constructor





Copy Constructor

A copy constructor is a constructor that takes the member values from one object and copies them into a new one.

Examples when copy constructor required:

- Creating a copy of an object explicitly
- Returning an object from function by value
- **Pass-by-value parameter to function** (what happens? See: Topic 3 Functions)
 - Uses **copy constructor** – if not defined then compiler implicitly defines a constructor that creates **shallow** copy e.g.

```
newObj.member1 = origObj.member1
```
 - In some cases **deep copy** required e.g. objects with dynamically allocated memory
 - Make **new** copy of all data members

Important: avoid leaving both objects pointing to the same (dynamically located) memory

Example

```
class Car{
public:
    Car() : make("Toyota") {this->year = new int(2024)};
    Car(const Car &);
    std::string make;
    int * year;
    void print();
};

Car::Car(const Car& rhs) {
    this->make = rhs.make;
    this->year = rhs.year;
}

void Car::print() {
    std::cout << this->make << " " << *(this->year) << std::endl;
}
```

```
Car car1;
std::cout << "Car 1:" << std::endl; car1.print();
Car car2(car1);

*car2.year = 2019;

std::cout << "Car 1:" << std::endl; car1.print();
std::cout << "Car 2:" << std::endl; car2.print();
```

Example

```
class Car{
public:
    Car() : make("Toyota") {this->year = new int(2024)};
    Car(const Car &);
    std::string make;
    int * year;
    void print();
};

Car::Car(const Car& rhs) {
    this->make = rhs.make;
    *(this->year) = new int(*(rhs.year));
}

void Car::print() {
    std::cout << this->make << " " << *(this->year) << std::endl;
}
```

```
Car car1;
std::cout << "Car 1:" << std::endl; car1.print();
Car car2(car1);

*car2.year = 2019;

std::cout << "Car 1:" << std::endl; car1.print();
std::cout << "Car 2:" << std::endl; car2.print();
```



Issues: Memory Management



Issues with Memory Management

Issues include:

- Memory Leaks: when dynamically allocated memory not properly deallocated
 - the leak leads to gradual loss of available memory
 - See zbyooks
- Dangling Pointers: when a pointer has address of deallocated memory
 - Accessing/dereferencing = undefined behavior e.g. crashes or corrupted data
- Double Delete:
 - deallocating memory more than once, or memory that was not dynamically allocated
 - can lead to undefined behavior e.g. crashes or corrupted data
- ...



Good Memory Management Practices:

Good memory management practices include:

- Properly allocate and deallocate memory e.g. 'new' and 'delete'
- Take care with pointers e.g. make sure they are valid/properly initialized before use.
- **Avoid raw pointers as much as possible!**
 - Use safer alternatives instead e.g. smart pointers (more in [Programming Studio 2](#))
- Use **standard library containers** and algorithms to minimize manual memory management.
- ...



Abstract Data Types and Linked Lists

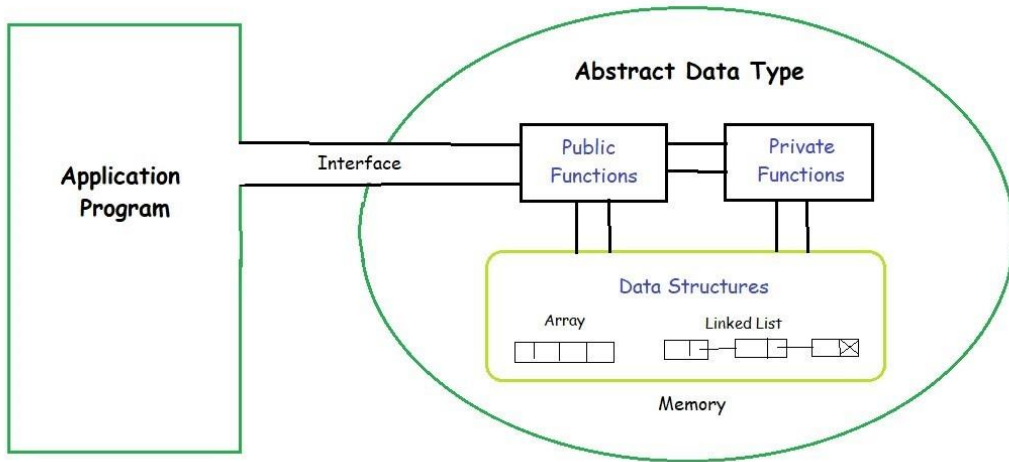
A first look (revisit in PS2)



Abstract Data Types

An **Abstract Data Type (ADT)** is a way of organizing code that allows stronger encapsulation and more readable program structure.

i.e., It is a set of data and a collection of operations that can be performed on that data.





Abstract Data Types

An **Abstract Data Type (ADT)** is a way of organizing code that allows stronger encapsulation and more readable program structure.

i.e., It is a set of data and a collection of operations that can be performed on that data.

Consists of:

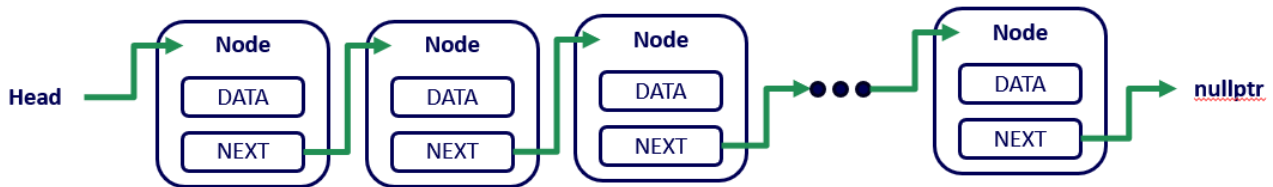
1. An interface: a set of operations that can be performed.
2. Allowable behaviours: a contract on how the ADT should behave and respond to the operations.

The implementation of an ADT requires:

1. An internal representation of the data (variables, objects, etc).
2. A set of methods implementing the interface.
3. Representation Invariants: a set of rules that all instances of the ADT must follow.
Determines what can and cannot be represented by the ADT.

Linked List ADT

A linked list is a sequence of “nodes”, where each node contains some data and a pointer to the next node.



Linked List ADT

For a basic linked list, we need to be able to add elements and remove them.

We will use Push and Pop (like a stack) to do this.

Push: add an element to the front of the list.

Pop: remove the element from the front of the list.

```
class Node{
public:
    Node(int data) : next(nullptr), data(data) {};
    Node* next;
    int data;
};
```

```
class LinkedList{
public:
    LinkedList() : head(nullptr) {};
    ~LinkedList();
    void push(int);
    int pop();
private:
    Node* head;
};
```

```
void LinkedList::push(int data) {
    Node * new_node = new Node(data);
    new_node->next = this->head;
    this->head = new_node;
}

int LinkedList::pop() {
    int data = this->head->data;
    Node * old_node = this->head;
    this->head = this->head->next;
    delete old_node;
    return data;
}

LinkedList::~~LinkedList() {
    while(this->head != nullptr) {
        this->pop();
    }
}
```

