

EECS 280: Programming and Introductory Data Structure

Course Material Summary Part 1

University of Michigan at Ann Arbor

Last Edit Date: 03/15/2023

Disclaimer and Term of Use:

1. We do not guarantee the accuracy and completeness of the summary content. Some of the course material may not be included, and some of the content in the summary may not be correct. You should use this file properly and legally. We are not responsible for any results from using this file.
2. This personal note is adapted from *Professor Amir Kamil, Andrew DeOrio, James Juett, Sofia Saleem, and Saquib Razak*. Please [contact us](#) to delete this file if you think your rights have been violated.
3. This work is licensed under a [Creative Commons Attribution 4.0 International License](#).

EECS 280 Summary Part 1

Pointers

Key Concepts:

- Pointers hold addresses
- Must be declared to same type as address (this includes const)
- Pointers that point to pointers are double pointers and use **.
- * is used for declaring and dereferencing pointers.
- & is used for creating references and getting the address of an object.

1. Address-of operator (&)

Use & to get the address of a declared object.

```
int x = 3;
int &r = x;
cout << x;      // prints 3
cout << r;      // prints 3
cout << &x;     // prints address of x
cout << &r;     // prints address of x
```

In `int x;`, `x` is a variable whose type is `int`.

In `int &r = x;`, `r` is a variable whose type is `reference-to-int`.

2. Dereference operator (*)

Addresses can be stored in pointers, which is declared by placing a * symbol to the left of the name declaration.

```
int x = 7;
int *ptr = &x;
cout << *ptr;    // prints 7
cout << ptr;     // prints address of x
```

In `int x = 7;`, `x` is a variable whose type is `int`.

In `int *ptr = &x;`, `ptr` is variable whose type is `pointer-to-int`.

In `cout << *ptr;`, the dereference operator “follows the pointer” to the object it points to.

3. Pointers and References

The * and & symbols mean different things when they are used as part of a type and when they are used in an expression:

- When used in a **type**, * means that the type is a pointer type, while & means the type is a reference type.
- When used as a **unary prefix operator** in an expression, * is used to dereference a pointer, while & is used to obtain the address of an object.
- When used as a **binary infix operator**, * is multiplication, while & is a bitwise operation.

Arrays

Key Concepts:

- Arrays act as pointers to the first element.
- We can use pointer arithmetic to traverse the array with pointers.

1. Basics

- An array has a fixed size, and its size cannot be changed.
- An array holds elements that are of the same type.

- The elements of an array are stored in a specific order, with the index of the first element being 0.
- The elements are stored contiguously in memory, one after another.

- Accessing any element of an array takes constant time, regardless of where the element is.

- Correct array declarations:

```
int arr[4] = {1, 2, 3, 4}; // 1, 2, 3, 4
int arr[4] = {1, 2};      // 1, 2, 0, 0
int arr[4] = {};          // 0, 0, 0, 0
int arr[] = {1, 2, 3, 4}; // 1, 2, 3, 4
```

2. Arrays and Pointers

- Directly assign an array to another array is forbidden
- The compiler converts the array into a pointer to the first element in the array.

```
int arr[4] = {1, 2, 3, 4};
cout << &arr[0]; // 0x1000
cout << arr;     // 0x1000
*arr = -3;
cout << arr;     // 0x1000
cout << arr[0];  // -3
```

array:	0x1000	1	array[0]
	0x1004	2	array[1]
	0x1008	3	array[2]
	0x100c	4	array[3]

- An integer can be added to or subtracted from a pointer, resulting in a pointer that is offset from the original one.
- Two pointers can be subtracted, resulting in an integer that is the distance between the pointers

```
int *ptr = arr;
cout << ptr << ", " << *ptr; // 0x1000,1
cout << ++ptr << ", " << *ptr; // 0x1004,2
cout << ptr + 2 << ", " << *ptr; // 0x100c,2
int *ptr1 = ptr + 2;
cout << ptr1 << ", " << *ptr1; // 0x100c,4
cout << ptr1 - ptr << ", " << ptr - ptr1; // 2,-2
cout << arr[ptr1 - ptr]; // 3
```

- Array indexing is implemented using pointer arithmetic.

```
cout << *(arr + 2); // prints 3
cout << arr[2];     // same as *(arr + 2)
cout << 2[arr];     // same as *(arr + 2)
```

- We can compare pointers' addresses and their values using operators such as <, >, <=, >=, ==, and !=.

3. Array Traversal by Pointers

```
int arr[5] = {5, 4, 3, 2, 1};
for (int *ptr = arr; ptr < arr + 5; ptr++) {
    cout << *ptr << endl; // prints all elements
}
```

Strings

1. Comparison between C strings and C++ strings

	C Strings	C++ Strings
Library Header	<cstring>	<string>
Declaration	char cstr[]; char *cstr;	string str;
Length	strlen(cstr);	str.length();
Copy Value	strcpy(cstr1, cstr2);	str1 = str2;
Indexing	cout << cstr[1];	cout << str[1];
Concatenate	strcat(str1, str2);	str1 += str2;
Compare	strcmp(str1, str2);	str1 == str2;

2. C-Style Strings

In C language, a string is represented as an array of characters, which looks like this: `char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'}`;

`'l', 'o', '\0'}`; Note that `\0` is a sentinel value, which denotes the end of a character array.

We can also initialize it with a string literal `char str[6] = "hello";` or `char str[] = "hello";`

Note: We can also declare a C-style string as a pointer.

3. Processing C-style String

We can set up a traversal by pointer loop that iterates until it happens upon the null character `\0`.

```
// Stops when '\0'
while (*cstr) {
    ...
}
```

4. Command Line Input

`$ g++ -Wall -Werror -pedantic -O1 test.cpp -o test`

- `g++` is a program you run.

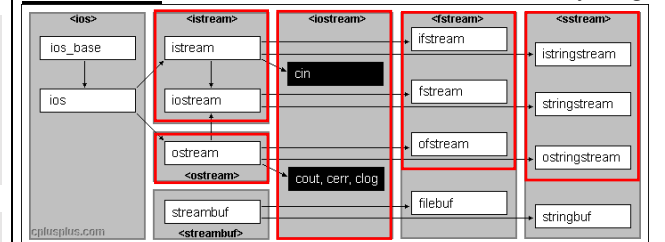
- `argc` - number of arguments getting passed in (8 in above example).

- `argv` - array of C-Strings containing your arguments.

- `atoi()` transforms the C-string input into an int.

- `int main(int argc, char *argv[]);` or `int main(int argc, char **argv);` works the same since array is a pointer

5. I/O Stream



Some useful stream operations:

<code>atoi()</code> convert C-style string to integer	<code>stoi()</code> convert C++ string to integer	<code>is_open()</code> check if a file is open	<code>fail()</code> check failbit or badbit is set or not
<code>good()</code> check state of stream is good or not	<code>eof()</code> check error state flag is set or not	<code>open()</code> open file <fstream>	<code>close()</code> close file <fstream>

`ifstream` reads (>>) from files; `ofstream` writes (<<) to files.

`istream` reads (>>) generally; `ostream` writes (<<) generally.

The const Key Word

const pointer: The pointer value (an address) itself cannot be changed	Pointer-to-const: the object which pointed by the pointer cannot be changed
<code>int x = 3;</code> <code>int * const ptr = &x;</code>	<code>int x = 3;</code> <code>int const * ptr = &x;</code> <code>const int * ptr = &x;</code>

Reference-to-const: A reference creates alia for an object, but the alia cannot be used to change the object.

```
int x = 3;
int const &y = x;
x = 10; //y not change
y = 10; //error
```

Note: `* const int ptr = &x;` and `* int const ptr = &x;` are not correct because `*` must have a type before it.

- Only a function that contains `const` in its signature is allowed to be called on `const` objects. Non-`const` objects can be used here, but `const` objects **cannot** be used in non-`const` functions.
- Use **pass-by-pointer** or **pass-by-reference** if modification of the original object is **needed**.
- Use **pass-by-pointer-to-const** or **pass-by-reference-to-const** if modification of the original object is **not needed**.
- Only use **pass-by-value** for fundamental and small objects.

Abstract Data Type (ADT) in C

1. Basic

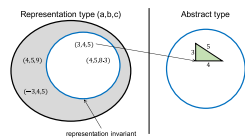
The C language only has support for structs with data members (i.e. member variables). The functions that operate on the ADT **must be defined separately** from the struct.

```
struct Triangle {
    double a;
    double b;
    double c;
};

void three(Triangle *tri){
    tri->a = 3;
    tri->b = 3;
    tri->c = 3;
}
```

Note: Not all structs are ADTs. An ADT must have function(s).

2. Representation Invariant & Implementation and Interface



For the example above, the invariants are as following:

- $a > 0 \ \&\& \ b > 0 \ \&\& \ c > 0$
- $a + b > c \ \&\& \ a + c > b \ \&\& \ b + c > a$
- Use `assert()` to check these.
- We only interact with an ADT through its interface. POD, a data representation, is such.

Abstract Data Type (ADT) in C++

1. Basic: We use `class` instead of `struct` to define C++ ADTs. A C++ class includes **both member variables and member functions**.

```
class Triangle {
    double a;
    double b;
    double c;
public:
    Triangle(double a_in,
             double b_in,
             double c_in);
    void scale(double s){
        this->a *= s;
        this->b *= s;
        this->c *= s;
    };
};

//Initializer list for ctor
Triangle(double a_in,
         double b_in,
         double c_in):
    a(a_in), b(b_in), c(c_in){}

//Normal init. for ctor
Triangle(double a_in,
         double b_in,
         double c_in){
    this->a = a_in;
    this->b = b_in;
    this->c = c_in;
}
```

2. Member Accessibility and Constructors

- In C++ class, we give a set of members a particular access by placing `private:` or `public:` before members (`private:` can be ignored when we have the `class` key word).
- Private members can only be accessed **within** the class.
- Ways to **call** a C++ constructor to initialize:

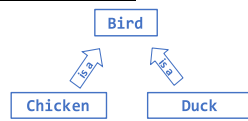
Normal Initialization	Uniform Initialization
Triangle t1; // default ctor	Triangle t4{3, 4, 5};
Triangle t2(3, 4, 5);	Triangle t5 = (3, 4, 5);
Triangle t3 = Triangle(3, 4, 5);	Triangle t6 = Triangle(3, 4, 5);

Note: If a class has any **non-default constructors** declared, then **no implicit default constructor** is provided. Instead, we have to **write a non-default constructor to make it work**.

Inheritance

Inheritance is the ability for a class to reuse the interface or functionality of another class.

1. Derived class



```
class Bird {
public:
    Bird(const string &name_in)
        : age(0), name(name_in){}
    ~Bird() {} // Destructor
private:
    int age;
    string name;
};
```

- Derived class must **also initialize base-class sub-objects**.
- Derived class **does not initialize inherited members**.

2. Ordering of Constructors and Destructors

```
class Duck : public Bird {
public:
    Duck(const string &name_in)
        : Bird(name_in), cross(0){}
    ~Duck() {} // Destructor
private:
    int cross;
};
```

- A **constructor** is invoked when a class-type object is **created**.
- A **destructor** is invoked when a class-type object **dies**.

- If there are multiple objects that are constructed and destructed, C++ follows a “socks-and-shoes” ordering:

```
int main() {
    cout << "ctor" << endl;
    Duck big("blue");
    cout << "dctor" << endl;
}

"ctor" -> Bird ctor -> Duck
dctor -> "dctor" -> Duck dctor
-> Bird dctor
```

Note: The base class `ctor` or `dctor` has to be called **every time** when a **derived class object is constructed or destructed**.

Polymorphism

1. Function Overloading (ad hoc polymorphism)

- When we invoke an overloaded function, the compiler resolves the function call by **comparing the types of the arguments** to the **parameters of the candidate functions** and finding the best match.
- Functions are needed to be in the **same scope** for overloading.
- Function signatures **must be different** for overloading.

2. Subtype Polymorphism

- Subtype polymorphism allows a **derived-class object to be used where a base-class one is expected**. **Indirection** avoids slicing.
- We can do it with pointers or reference to avoid copy:

```
Duck bigduck("haha");
Bird &bird_ref = bigduck; //bird_ref is an alias of bigduck
Bird *bird_ptr = &bigduck; //bird_ptr is a ptr of bigduck
```

- Implicit downcast is **forbidden**. Use **explicit downcast** instead:
`Duck &duck_ref = static_cast<Duck &>(bird_ref);`
`Duck *duck_ptr = static_cast<Duck *>(bird_ptr);`
- The inheritance relation **must be accessible** (`: public Bird`).

3. Static and Dynamic Binding

- The **static type** of a pointer/reference is the type it is **declared** with and is known at **compile time**.
- The **dynamic type** of a pointer/reference is the type of the object it is **actually pointing to at run time**.

4. Dynamic cast

It can let us use the function in the class object we pointing to. If the pointed-to object is not the requested type, the result is null.

```
Duck d("MyDuck");
Bird *bPtr = &d;
cout << bPtr->talk() << endl; // tweet
Duck *dPtr = dynamic_cast<Duck *>(bPtr);
cout << dPtr->talk() << endl; // quack
```

5. Virtual Function & Override Key Word

- Only virtual functions can be overridden, at least base class needs it. Note that only functions with same signature can override.
- We may not need the `virtual` key word in derived class, but `override` key word is needed. That's says in a derived class, `virtual int func() const{...}` is the same as `int func() const override{...}`.

6. Member Lookup

- The compiler starts by looking for a member with that name in the **compile-time** or **static type** of the object. It is an error if no member of the given name is found in the static type or its base types (compiler goes to base class if not found in derived class).
- If the member found is an **overloaded function**, then the arguments of the function call are used to determine which overload is called.
- If the member is a **variable or non-virtual function** (including static member functions), the access is **statically bound** at compile time.
- If the member is a **virtual function**, the access uses **dynamic binding**. At runtime, the program will look for a function of the **same signature**, starting at the dynamic type of the receiver, then proceeding to its base type if necessary.

7. Polymorphism and Design

- When a **base class** defines a function as **virtual**, it means the derived classes may provide an overridden version if they want.
- When a base class defines a function as **pure virtual**, it is an abstract class and cannot be instantiated. Derived classes **must** provide an overridden version (or else remain themselves abstract).
- Always use `override` keyword in derived class in these cases.

8. Overload vs. Override

- **Function overloading** (achieved at **compile time**) provides multiple definitions of the function by **changing signature**. (**Base and derived classes**)
- **Function overriding** (achieved at **run time**) redefines the base class function in its derived class with same signature. (**Derived class only**). Need to use `virtual` or `override` keyword.

Other Stuff

- Compile time (machine code); runtime (running, after compile)
- variable (compile time); object (runtime) not all obj. are var.(C string)
- The **interface** of an abstraction specifies **what the abstraction does**, while the **implementation** specifies **how the abstraction does it**.
- Binary $101 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 = 6$ decimal
- Hex $B5 = 1011 \ 0101 = 5 \times 16^0 + 11 \times 16^1 = 181$ decimal