# C++ & Software Develpment

# Table of Contents

# C++

# General

## Definition, Declaration, Initialization, instantiation

#### Decleration

- Exist somewhere
- Introduction of a new name

Example:

```
void xyz();
```

#### Definition

- Exists here, allocate memory for it
- Definition of prefously declared name or both

Example:

```
void xyz(){...};
```

#### Instantiation

- Allocating memory for object

#### Initialization

- Assignement of a value at construction-time

Example:

```
int a = 5;
```

## Implicit vs. Explicit

```
int int1 = int(0,100); // explicit
int int2(0,100); // implicit
```

### auto Keyword

### AnyIndex

### std::pair

### Aligned Queue

### IndexSet

### BucketQueue

### Run Time Polymorphism

### AlignedVector

# Pointer, Reference and Dynamic Memory Allocation

Pointers, References and Dynamic Memory Allocation are the most powerful features in C/Cpp language, which allows programmers to directly manipulate memory to efficiently manage the memory - the most critical and scarce resource in computer - for best performance. However, "pointer" is also the most complex and difficult feature in C/C++ language.

Pointers are extremely powerful because they allows you to access addresses and manipulate their contents. But they are also extremely complex to handle. Using them correctly, they could greatly improve the efficiency and performance. On the other hand, using them incorrectly could lead to many problems, from un-readable and un-maintainable codes, to infamous bugs such as memory leaks and buffer overflow, which may expose your system to hacking. Many new languages (such as Java and C#) remove pointer from their syntax to avoid the pitfalls of pointers, by providing automatic memory management.

## Pointer

A pointer variable (or pointer in short) is basically the same as the other variables, which can store a piece of data. Unlike normal variable which stores a value (such as an int, a double, a char), a pointer stores a memory address.

#### Uninitialized Pointer

When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location. This is dangerous! You need to initialize a pointer by assigning it a valid address. This is normally done via the address-of operator (&).

When you only declare a pointer, the memory for that pointer is allocated but the contents of that memory are not touched. As such, the value of the pointer, which is just a number representing some other place in memory, is whatever value happens by chance to be in that memory.

It could be pointing anywhere, and the compiler will happily let you use it without initialising it. Some compilers will warn you about it.

When you actually run the program, you will be accessing some random piece of memory; if you’re lucky, you'll get a segfault. If you're unlucky, you'll trash data without even knowing it.

Uninitialized pointers can cause the following problems: * It might be pointing at memory you don't have access to, in which case it causes a segmentation fault and crashes your program * It might be pointing at real data, and if you don't know what it's pointing to, you're causing unpredictable (and very hard to debug) changes to your data. * You have no way of knowing if it's been initialized or not - because how do you tell the difference between a valid address and the address that happened to be there when you declared the pointer?

Example: The pointer **p** is uninitialized and points to a random location in memory when you declare it. It could be pointing into the system stack, or the global variables, or into the program's code space, or into the operating system. When you say ***p**=**12**;, the program will simply try to write a 12 to whatever random location p points to. The program may explode immediately, or may wait half an hour and then explode, or it may subtly corrupt data in another part of your program and you may never realize it. This can make this error very hard to track down. Make sure you initialize all pointers to a valid address before dereferencing them.

```
int main()
{
    int* p;
    *p = 56;
}
```

**Initializing Pointers via the Address-Of Operator (&)**

The address-of operator (&) operates on a variable, and returns the address of the variable. For example, if number is an int variable, &number returns the address of the variable number.

You can use the address-of operator to get the address of a variable, and assign the address to a pointer variable.

Example:

```
int number = 88;     // An int variable with a value
int * pNumber;       // Declare a pointer variable called pNumber pointing to an int
(or int pointer)
pNumber = &number;   // Assign the address of the variable number to pointer pNumber

int * pAnother = &number; // Declare another int pointer and init to address of the
variable number
```

**NULL Pointer**

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

Sometimes people recommend that pointers be assigned to NULL, the universal value meaning "this pointer doesn't point at anything", because a lot of code already checks for NULL pointers. For example, if you call free(NULL), it's guaranteed to do nothing. If you passed an uninitialized pointer in who knows what would happen.

Using a NULL pointer has the following advantages: * If I try and dereference it, it will still segfault, but at least I can test if it's NULL and act accordingly * I know that it WILL segfault, and do something else. If it's a random value, I don't know anything until it crashes. * If you initialize it to NULL, I can't make it point to data unless I explicitly tell it to. So I only modify what I meant to. * As implied above, I can tell when I've initialized it and when I haven't, and make a decision.

Example:

```
int *p = 0;
int *p = NULL;
```

Usecase:

```
int doSomething(int* x) {
    if(x == NULL) {
        return ERROR_CODE;
    }
    ...
    // do something with x
    ...
    return SUCCESS;
}
```

**Dangling Pointer**

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where Pointer acts as dangling pointer

Example 01: De-allocation of memory

```
// Deallocating a memory pointed by ptr causes dangling pointer
int main()
{
    int *ptr = (int *)malloc(sizeof(int));

    // After below free call, ptr becomes a dangling pointer
    free(ptr);

    // No more a dangling pointer
    ptr = NULL;
}
```

Example 01: Function Call

```c
// The pointer pointing to local variable becomes
// dangling when local variable is static.
int* fun()
{
    // x is local variable and goes out of
    //scope after an execution of fun() is over.
    int x = 5;
    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
```

The above problem doesn’t appear (or p doesn’t become dangling) if x is a static variable.

```c
int *fun()
{
    // x now has scope throughout the program
    static int x = 5;

    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // Not a dangling pointer as it points
    // to static variable.
    printf("%d",*p);
}
```

Example 03: Variable goes out of scope

```
void main()
{
  int *ptr;
  .....
  .....
  {
      int ch;
      ptr = &ch;
  }
  .....
  // Here ptr is dangling pointer
}
```

**void* Pointer**

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value. Important Points

Example:

```
int main()
{
    int x = 4;
    float y = 5.5;

    //A void pointer
    void *ptr;
    ptr = &x;

    // (int*)ptr - does type casting of void
    // *((int*)ptr) dereferences the typecasted
    // void pointer variable.
    printf("Integer variable is = %d", *( (int*) ptr) );

    // void pointer is now float
    ptr = &y;
    printf("\nFloat variable is= %f", *( (float*) ptr) );

    return 0;
}
```

- void pointers **cannot** be dereferenced. It can however be done using **typecasting** the void pointer

Example: ERROR dereferencing

```
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Example: CORRECT dereferencing

```
int main()
{
    int a = 10;
    void *ptr = &a;
    printf("%d", *(int *)ptr);
    return 0;
}
```

- Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

Example: ERROR pointer arithmetic

```
int main()
{
    int a[2] = {1, 2};
    void *ptr = &a;
    ptr = ptr + sizeof(int);
    printf("%d", *(int *)ptr);
    return 0;
}
```

**Uninitilaized Pointer vs. NULL Pointer**

- An uninitialized pointer stores an undefined value.
- A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

**Void Pointer vs. NULL Pointer**

- Null pointer is a value, while void pointer is a type

**Pointer Arithmetic**

**this Pointer**

To understand '**this**' pointer, it is important to know how objects look at functions and data members of a class.

- Each object gets its own copy of the data member.
- All access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share single copy of member functions. Now the question is, what if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated? Compiler supplies an implicit pointer along with the functions names as '**this**'. The '**this**' pointer is passed as a hidden argument to **all nonstatic member function calls** and is available as a local variable within the body of all nonstatic functions. The '**this**' pointer is a constant pointer that holds the memory address of the current object. The '**this**' pointer is not available in static member functions as static member functions can be called without any object (with class name). For a class X, the type of the '**this**' pointer is 'X* const'. Also, if a member function of X is declared as const, then the type of the '**this**' pointer is 'const X *const' (see this GFact)

Following are the situations where '**this**' pointer is used:

1. When a local variable's name is same as a member's name

Example:

```
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
      // The 'this' pointer is used to retrieve the object's x
      // hidden by the local variable 'x'
      this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

Output:

```
x = 20
```

For constructors, initializer list can also be used when parameter name is same as member's name.

1. To return reference to the calling object

Example:

```
Test& Test::func ()
{
  // Some processing
  return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.

```
class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

The type of this depends upon function declaration. If the member function of a class X is declared const, the type of this is const X* (see example 01 below), if the member function is declared volatile, the type of this is volatile X* (see example0 2 below), and if the member function is declared const volatile, the type of this is const volatile X* (see example 03 below).

Example 01: Type const

```
class X {
  void fun() const {
    // this is passed as hidden argument to fun().
    // Type of this is const X*
    }
};
```

Example 02: Type volatile

```
class X {
  void fun() volatile {
    // this is passed as hidden argument to fun().
    // Type of this is volatile X*
    }
};
```

Example 03: Type const volatile

```
class X {
  void fun() const volatile {
    // this is passed as hidden argument to fun().
    // Type of this is const volatile X*
    }
};
```

Ideally delete operator should not be used for this pointer. However, if used, then following points must be considered.

1. **delete** operator works only for objects allocated using operator new (See https://www.geeksforgeeks.org/?p=8539). If the object is created using new, then we can do delete this, otherwise behavior is undefined.

Example:

```
class A
{
  public:
    void fun()
    {
        delete this;
    }
};

int main()
{
  // Following is Valid
  A *ptr = new A;
  ptr->fun();
  // make ptr NULL to make sure that things are not
  // accessed using ptr.
  ptr = NULL;

  // And following is Invalid: Undefined Behavior
  A a;
  a.fun();

  getchar();
  return 0;
}
```

Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:15:7: error: 'NULL' was not declared in this scope
ptr = NULL; // make ptr NULL to make sure that things are not
// accessed using ptr.
prog.cpp:22:9: error: 'getchar' was not declared in this scope
getchar();
```

1. Once delete this is done, any member of the deleted object should not be accessed after deletion.

```
class A
{
  int x;
  public:
    A() { x = 0;}
    void fun() {
      delete this;

      /* Invalid: Undefined Behavior */
      cout<<x;
    }
};
```

The best thing is to not do delete this at all.

#### Dereferencing

#### Array Decay

#### const Pointer

#### const char *

Defines a variable pointer to a constant char, not a constant pointer to a variable char. This is the correct way to declare a pointer to a string literal, because the way you're doing it now, the compiler will let you do this:

```
char * terry = "hello";
terry[0]='H'; // no error
```

But the program will crash when you try to run it. However, if you try to do this:

```
const char * terry = "hello";
terry[0]='H'; // error
```

The compiler won't let you. The correct way to use a variable pointer with a variable char is:

```
// Allocates an array of size 6 on the stack and copies to it the contents of "hello".
char modifiable_string[]="hello";
char *variable_pointer_to_variable_char=modifiable_string;
```

## Smart Pointers

C++ is about memory ownership "ownership semantics". The owner of a of a chunk of dynamically allocated memory is responsible to realse that memory.

Consider the following simple C++ code with normal pointers.

```
MyClass *ptr = new MyClass();
ptr->doSomething();
// We must do delete(ptr) to avoid memory leak
```

Using smart pointers, we can make pointers work in way so that we don't need to explicitly call delete. Smart pointer is a wrapper class over a pointer with operator like * and → overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can't like automatic destruction (yes, we don't have to explicitly use delete), reference counting and more. The idea is to make a class with a pointer, destructor and overloaded operators like * and →. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically deleted (or reference count can be decremented).

Example:

```
// A generic smart pointer class
template <class T>
class SmartPtr
{
  T *ptr;  // Actual pointer
public:
  // Constructor
  explicit SmartPtr(T *p = NULL) { ptr = p; }

  // Destructor
  ~SmartPtr() { delete(ptr); }

  // Overloading derefercing operator
  T & operator * () {  return *ptr; }

  // Overloding arrow operator so that members of T can be
  // accessed like a pointer (useful if T represents a class
  // or struct or union type)
  T * operator -> () { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}
```

Output:

```
20
```

**Unique Pointer**

Can’t be copied

**Shared Pointer**

Keeps track of all references. Memory gets freed as soon as the reference count is 0.

**Weak Pointer**

Shared pointer that does not take onwership of the entity i.e. does not increase reference count.

**std::make_shared**

- Used for creating shared pointers for exception
- Do not use explicit initialization for a shared pointer with **new** type, because an additioal control block is needed. I

## Dynamic Memory Allocation

**new Keyword**

**malloc**

Memory allocation

```
int *tag = (int *) malloc(sizeof(int));
```

malloc() and calloc() return void* type and this allows these functions to be used to allocate memory of any data type (just because of void*) . In Cpp, we must explicitly typecast return value of malloc to (int *). Note that the below line without the typecast compiles in C, but doesn’t compile in Cpp.

```
int *tag = malloc(sizeof(int) * n);
```

**new vs. malloc()**

**delete() and free()**

## References

In C++, Reference variables are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized. But there are exceptions where we can have invalid references.

Example: Reference to value at uninitalized pointer

```
int *ptr;
// Reference to value at some random memory location
int &ref = *ptr;
```

Example: Reference to a local variable is returned

```
int& fun()
{
  int a = 10;
  return a;
}
```

Once fun() returns, the space allocated to it on stack frame will be taken back. So the reference to a local variable will not be valid.

## Pass Arguments by Reference or Pointer?

In C++, variables are passed by reference due to following reasons:

1. To modify local variables of the caller function: A reference (or pointer) allows called function to modify a local variable of the caller function.

Example:

```
void fun(int &x) {
    x = 20;
}

int main() {
    int x = 10;
    fun(x);
    cout<<"New value of x is "<<x;
    return 0;
}
```

Output:

```
New value of x is 20
```

1. For passing large sized arguments: If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object. For example, let us consider the following Employee class and a function printEmpDetails() that prints Employee details.

Example:

```
class Employee {
private:
    string name;
    string desig;
    // More attributes and operations
};

void printEmpDetails(Employee emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();
    // Print more attributes
}
```

The problem with above code is: every time printEmpDetails() is called, a new Employee abject is constructed that involves creating a copy of all data members. So a better implementation would be to pass Employee as a reference.

```
void printEmpDetails(const Employee &emp) {
    cout<<emp.getName();
    cout<<emp.getDesig();
    // Print more attributes
}
```

This point is **valid only** for struct and class variables as we don't get any efficiency advantage for basic types like int, char.. etc.

1. To avoid Object Slicing: If we pass an object of a subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value. For example, consider the following program, it prints "This is Pet Class".

Example :

```
class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

void describe(Pet p) { // Slices the derived class object
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}
```

Output:

```
This is Pet Class
```

If we use pass by reference in the above program then it correctly prints “This is Dog Class”. See the following modified program.

```
class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

// Doesn't slice the derived class object.
void describe(const Pet &p) {
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}
```

Output:

```
This is Dog Class
```

This point is also not valid for basic data types like int, char, .. etc.

1. To achieve Run Time Polymorphism in a function: We can make a function polymorphic by passing objects as reference (or pointer) to it. For example, in the following program, print() receives a reference to the base class object. print() calls the base class function show() if base class object is passed, and derived class function show() if derived class object is passed.

Example:

```
class base {
public:
    // Note the virtual keyword here
    virtual void show() {
        cout<<"In base \n";
    }
};


class derived: public base {
public:
    void show() {
        cout<<"In derived \n";
    }
};

// Since we pass b as reference, we achieve run time
// polymorphism here.
void print(base &b) {
    b.show();
}

int main(void) {
    base b;
    derived d;
    print(b);
    print(d);
    return 0;
}
```

Output:

```
In base
In derived
```

As a side note, it is a recommended practice to make reference arguments **const** if they are being passed by reference only due to reason no. **2** or **3** mentioned above. This is recommended to **avoid unexpected modifications** to the objects. === Other

# Software Development

## Undefined Behaviour

In computer programming, undefined behavior (UB) is the result of executing computer code whose behavior is not prescribed by the language specification to which the code adheres, for the current state of the program. This happens when the translator of the source code makes certain assumptions, but these assumptions are not satisfied during execution.

The behavior of some programming languages—most famously C and Cpp—is undefined in some cases. In the standards for these languages the semantics of certain operations is described as undefined. These cases typically represent unambiguous bugs in the code, for example indexing an array outside of its bounds. An implementation is allowed to assume that such operations never occur in correct standard-conforming program code. In the case of C/C++, the compiler is allowed to give a compile-time diagnostic in these cases, but is not required to: the implementation will be considered correct whatever it does in such cases, analogous to don't-care terms in digital logic. It is the responsibility of the programmer to write code that never invokes undefined behavior, although compiler implementations are allowed to issue diagnostics when this happens. This assumption can make various program transformations valid or simplify their proof of correctness, giving flexibility to the implementation. As a result, the compiler can often make more optimizations. It also allows more compile-time checks by both compilers and static program analysis.

## Buffer Overflow

## Memory Leaks