ADVANCED CPP 3

POINTERS

Char\* buffer = new char[8]; // allocate 8 bytes on the heap

Memset(buffer, 0, 8);

Delete[] buffer // remove from heap

*Memset(void\* ptr, int x, size\_t n);*

*Void\* ptr -* starting address of memory to be filled

*Inx x -* value to be filled

*Size\_t n* - number of bytes to be filled

REFERENCES

*Example 1*

Int a = 10;

Int& ref = a;

ref = 2;

cout << a << endl; // 2

*Example 2*

Void increment(int& a) {

a++;

}

Int main() {

Int a = 10;

Increment(a);

Cout << a << endl; // 11

}

// Use the example above to pass references to functions

When declaring a reference, you have to assign a value:

Int &ref; // **not allowed!**

Int a = 10;

Int& ref = a; // good!

You also cannot reassign:

Int& ref = a;

Ref = b; // this will change the value of a to b

If you want to reassign, you will have to use pointers:

Int\* ref = &a; // point to memory address of a

\*ref = 2; // change value of a

Ref = &b; // point to memory address of b

\*ref = 1; // change value of b

STATIC

The *static* keyword is used differently inside of classes than it is outside of classes.

Outside of class

Outside of classes, marking a variable as static means that that variable is only available in that translation unit, i.e. not global.

|  |  |  |
| --- | --- | --- |
| ***Main.cpp*** | ***Other.cpp*** | ***Description*** |
| Int n = 20; | Static int n = 10; | This is fine, as n in *other.cpp* refers to just that translation unit |
| Int n = 20; | Int n = 30; | Error! Linking error as both global variables. The linker doesn’t know which to use. |
| Int n = 30; | Extern int n; | Fine, *other.cpp* knows the n var is coming from another file |
| Static int n = 10; | Extern int n; | Error! n marked static in *main.cpp* so cannot be accessed by *other.cpp* |

Think of static in the context of marking variables as private in their translation units (lexical environments/files).

It is good practice to mark variables and functions as static, unless you need them to be used across multiple translation units.

Inside of class

Marking as *static* means there is only one version of that variable/method among multiple instances.

When using *static* in a class, you have to define them outside a class, otherwise you will get a linking error (unresolved external symbol).

Class Test {

Public:

Static int n;

}

Int Test::n; // define – you can also assign value

Int main() {

Test test1;

Test1.n = 100;

Test test2;

Cout << test2.n << endl; // you’d never write a static var like this

Cout << Test::n << endl; // instead call like this

}

// output **2**, even though test1 instance was the one that set the value.

Static methods cannot access non-static members! You must make members also static if you wish to access them within a static method.

STRINGS (under the hood)

Const char\* name = “Ali Issaee”; // array of char

// even though this is marked as a pointer, we do not need to delete it after use.

Char name[3] = {‘A’, ‘l’, ’i’};

Cout << name < endl; // this will print ali followed by a load of random chars, as not null terminated

If we set a debug after the name was initialized and inspect the memory (in VS – debug > windows > memory > memory 1) and type the variable *name*, you will see the location in memory.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **bytes** | 1 | 2 | 3 | 4 | 5 | 6 |
| **Memory** |  |  |  |  |  |  |  |
| 0x00F0FC20 |  | 61 | 6c | 69 | cc | cc | cc |
| 0x00F0FC20 |  | 61 | 6c | 69 | 00 | cc | cc |

Note in the first row, the ASCII characters are displayed in the first 3 bytes, (‘A’, ‘l’ and ‘i’), the 4th byte has cc.

Instead, define name as so:

Char name[4] = {‘A’, ‘’l’, ‘i’, ‘\0’};

The second row in the table above, you can see the 4th byte is 00, which is the null terminator. This defines the end of the string. Now if you output *name* you will get the expected response.

In c++ we should be using the <string> library.

Note that if you try to write a char pointer without *const*, your compiler might give an error. This is because a char pointer is a read only variable and is not modifiable.

CONST

Int num = 20;

Const int\* a = new int;

\*a = 30; // Illegal! Cannot change value as *const* is at start of assignment

a = &num; // this is fine as we change the reference

int\* const a= new int;

\*a = 30; // this is fine as *const* refers to reference being constant

a = &num; // illegal!

Const int\* const a = new int(10); // cannot change reference or value!

// note where the asterisk \* is, is it after *int* or *const*? *int const\* a;* is the same as *const int\* a;*

Const in class

Class Entity {

Private:

Int m\_x;

Int\* m\_y;

Public:

Int getX() const {return m\_x;} // we cannot modify class members here, *m\_x = 10* here would be illegal

Const int\* const getY() const {

Return m\_y;

}

// 1st const – returns a pointer that cannot be modified

// 2nd const – the contents of the pointer cannot be modified

// 3rd const – we cannot modify class members

Void print(const Entity& e) {

Cout << e.getX() << endl;

}

If getX() wasn’t const, then this function wouldn’t compile as we are passing a const reference as an argument which means we cannot modify the variable *e.*

If getX() wasn’t const, then how does this function know that getX() doesn’t modify any of its contents? We therefore have to mark methods as constants in order to pass const references to functions.

This is just one of the ways as to why it is useful.

Always mark class methods as *const* if they don’t/not supposed to modify any contents of the class.

Mutable

The *mutable* keyword allows variables to be mutable inside of const methods.

Class Test{

Private:

Mutable int n = 10;

Public:

Void modify() const {n = 30;}

}

MACRO

If you right click ‘solution’ in VS and go to C/C++ > preprocessor, you can define preprocessor definitions for certain configurations. E.g. put ‘\_DEBUG’ in the *debug* config.

#ifdef \_DEBUG

#define LOG(x) std::cout << x << std::endl

#else

#define LOG(x)

#endif

Int main() {

LOG(“Hello”); // removed in release mode, thus incr speed

}

Macros are mostly used for debugging purposes.

MULTIDIMENSIONAL ARRAYS

Int\* array = new int[50]; // normal array, allocate 200 bytes

Int\*\* a2d = new int\*[50]; // still 200 bytes. Points to 50 pointers to int, i.e. an array of ints

A 3 dimensional array would look like this:

Int\*\*\* a3d = new int\*\*[50]; // a pointer to a pointer to a pointer

//using the a2d example above

A2d[0][0] = 0;

A2d[0][1] = 1;

A2d[0][2] = 2;

As the array was heap allocated, you need to iterate through the array and delete when you have finished with it. If you just use: *delete[] a2d*, there will be a memory leak as that will only release the 50\*4 bytes, and not the bytes within the array itself.

For (int I = 0; I < 50; i++) {

Delete[] a2d[i];

}

Delete[] a2d;

Single dimensional arrays are less expensive, so when possible use these and avoid multidimensional arrays.

UNION

Similar to a struct, but can only occupy the memory of one member at a time. Unions are useful when you want to give 2 different names to the same variable. The purpose of a union is to save memory by using the same memory region for storing different objects at the same time.

// MORE INFO NEEDED ON THESE

VIRTUAL DESTRUCTOR

Virtual destructors are important when dealing with polymorphism and to prevent memory leaks. As destructors are used to clean up classes and regain any memory that was used by the class, it is important to mark any destructors as virtual so the right destructors are called on object deletion.

Class Parent {

Public:

Virtual ~Parent() { // cleanup }

};

Class Child : public Parent {

Public:

~Child() { // cleanup }

};

Int main() {

Parent\* polymorphic = new Child;

Delete polymorphic; // both destructors called

}

If the parent class wasn’t marked as virtual, then only the Parent destructor would be called, and any clean up by the Child class will have been missed.

PRECOMPILED HEADER

A precompiled header is a header file that is compiled into an intermediate form that is faster to process for the compiler. It is useful, as when a change in your code happens, the compiler doesn’t have to go through header files (and header files within header files…), and compile them.

Header files are compiled for each translation unit, e.g. if 2 files are including <vector>, the compiler compiles it twice. Do not put frequently changing files into precompiled headers. Precompiled headers are vital for files which aren’t yours, i.e. external libraries.

*Pch.h*

#include <iostream>

#include <vector>

// include all other external libraries

*Pch.cpp*

#include “pch.h”

1. In VS, right click pch.cpp > properties > c/c++ > precompiled headers:
   1. Precompiled header: create
2. Now go to properties of entire project > c/c++ > precompiled headers
   1. Precompiled header: use
   2. Precompiled header file: pch.h

NAMESPACES

Use namespaces to avoid naming conflicts. You can then access the variables in 3 ways:

1. Import all symbols from a namespace in to a scope

Void fn() {

Using namespace ns;

}

1. Import individual symbols from a namespace in to a scope

Void fn() {

Using ns::CONST\_NAME;

}

1. Explicitly qualifying a symbol for one-time use

Void fn() {

Cout << ns::CONST\_NAME;

}

DO’s AND DON’Ts

1. Prefer *const* and *inline* to #define (macros)
   1. i.e. prefer the compiler to the preprocessor
2. prefer <iostream> to <stdio.h>
3. prefer *new* and *delete* to *malloc()* and *free()*
4. use *delete* on pointer members in destructors
5. strive for class interfaces that are complete and minimal
6. avoid data members in the public interface (prevent read/write access). Use getters and setters in the public interface instead.
7. Use *const* whenever possible
8. Prefer pass by reference to pass by value
9. Use namespaces to avoid naming conflicts

QUIZ