C++ compares to C

# [ POINTERS ]

There are some differences in pointers between the two languages. First, let us review what pointers are. In C and C++, there are times when we do not wish to use up memory in the stack. Instead we wish to dynamically allocate memory so that it will be more useful to is in various ways, such as creating linked lists. This is done by creating a small place in the stack that holds nothing but an **address**. This address then **points** to a place in memory called the **heap***.* In this heap is where the actual data is stored.

Here is an example of how all this works. Think of each row as a space in memory, each with a distinct address. The memory itself is split up into multiple areas. The Static Area holds all global variables. The stack holds the local functions/variables. The heap is allocated memory. The following code will produce the structure given in the example below:

OnlinePurchaseStructure \*s;

int main(void){  
 (\*s).amount = 18146;

s->name = "Alienware Purchase";

(\*s).CreditCardNumber = 1086xxxxxxxxxxx15;

s->modelNum = m14x;

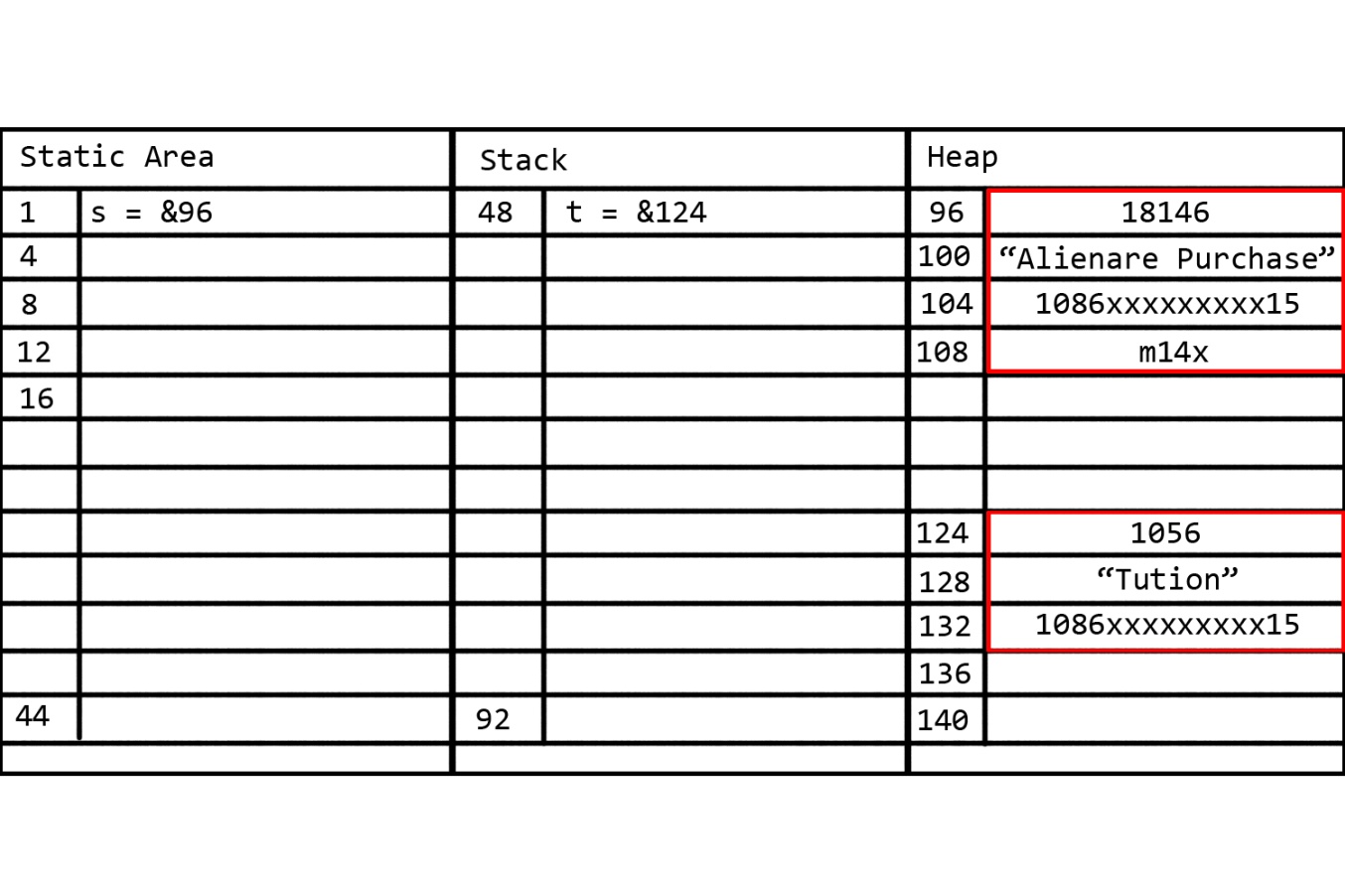
TuitionStructure \*t;

t->amount = 1056;

(\*t).name = "tuition"

t->CreditCardNumber = 1086xxxxxxxxxxxxx15;

}



The two pointers, s and t, do not want to hold the WHOLE structure in the static or stack area, so they simply POINT to an area in the heap that holds the real data.

Note, however, that REAL memory addresses aren't written with simple numbers. They are written as hexadecimal, like so: 0x108A24F8

Syntax

Size of

When creating these pointers, they must be created in the exact size of the variable we are trying to create. A pointer to an int must be the EXACT size of an int. A pointer to a struct that holds 4 ints must be the EXACT size of 4 ints. We can obtain the size of things by using these commands:

//In both C and C++

sizeof(int);

Creation

To create a pointer, put an asterisk before the name.

//In both C and C++

int\* savings;

Modification

To access the data, we need to know of two things: How to get the ADDRESS of the data, and how to DEREFERNCE the data. The following example demonstrates this:

//In both C and C++

int main(void){

int intVar;

int\* pointVar;

pointVar = &intVar;

\*pointVar = 10;

cout << intVar;

}

In this example, the outputn will be 10. The & operator gives you *the address it is currently at.* The \* variable when being called *gives the value it is pointing to.* If something is a pointer and you do not put an asterisk \* before calling it, it will output the hex address location.

Within Structures

If the pointer is a pointer to a structure, as with our first example, then there are two way you can access its data:

student s;

\*s.gpa = 4.0; // <-- THIS DOES NOT WORK. Becuase the . operater goes BEFORE the \* // operator, causing a LOT of headaches.

(\*s).gpa = 4.0; // This works.

s->gpa = 4.0; // This also works.

The two above statements are completely equivalent, it's just a matter of it being a WHOLE LOT EASIER to understand with -> instead of (\*s).

Allocating Memory in the heap

This is where some differences between c and c++ comes in, mostly with naming. Let's say you wanted to create a variable in one function that DOESN'T DIE when the function ends, so that it can be used in other function. One answer is a global variable. The second answer is to create memory in the heap and store it there so that you can access it at any time. This happens because when you create memory in the heap, it is NOT going to go away until you FREE IT. This is important to note. When you allocate memory from the heap, it MUST eventually be freed, otherwise it is a MEMORY LEAK. You can allocate memory and delete like so:

//In c

int\* intVar = malloc(sizeof(int));

free(intVar);

//In c++

int\* intVar = new int;

delete intVar;

Pointers and arrays

Arrays are cool. Everyone likes arrays. Did you know what they REALLY are though? Turns out an array is a bunch of space in the heap that is reserved ONE RIGHT AFTER THE OTHER. In this way, we can access ANYTHING in the array, at ANY time, INSTANTLY. How is this done, one would ask? Like so:

//In c

int main(void){

int \*v = malloc(3\*(sizeof(int));

\*v = 42;

\*(v+1) = 43;

\*(v+2) = 44;

printf("%d\n%d\n%d\n",\*v, \*(v+1), \*(v+2));

free v;

}

//In c++

int main(void){

int \*v = new int [3];

\*v = 42;

\*(v+1) = 43;

\*(v+2) = 44;

cout << \*v << "\n" << \*(v+1) << "\n" << \*(v+2) << "\n";

delete v;

}

these will both output the exact same code in the terminal:

42

43

44

Cool, right? This EXACT SAME IDEA is encapsulated in arrays. Arrays ARE pointers. C++ will interpret them like this:

int a[5]; ---> int\* a = new int [5];

int b = a[2]; ---> int b = \*(a + 2);

If you want a more detailed description of arrays, check out the array section.

# [ CLASS ] - And object Oriented Programming

Classes are structs, through and through. Another word for a struct is a *class,* and this does nothing but allow you to define it as *private, public, or protected.* Classes allow you to define variables in them as the most basic level, but you can *also* define functions in them, which are called *member functions* or *methods.*

NOW, here's the biggest thing to take from this: Let's say you had a savings account class with two functions: deposit() and withdraw(). You COULD make these outside the function, BUT, this BREAKS the idea of object-oriented programming. The idea is to have a CLASS, a **self-contained** structure that can do everything it needs to do all within it, ready to go. Like a GUN class, or CHARACTER class, if this was a game. The class gives you variables, functions, and everything you need to do something/store data/etc. This is the idea of OOP.

Now, after defining a function in a class, you want to call it in a different function. How do you do that? The same way you NORMALLY handle situations like this:

class account{  
 public:

int savings;

int checking;

void deposit(int amount) {  
  checking = (checking + amount)

}  
 };

In this sutuation, it is handled exactly as you anticipate:

int main(void){  
 account s;

s.savings = 2000;

s.checking = 4500;

s.deposit(2500);  
 }

This accesses the data that you need in s and modifies it.

Now, what is this :: nonsense that seems to appear everywhere in c++? This is the scope resolution. It basically tells the compiler to use the GLOBAL function/variable, and NOT the local one. For an example, see the [ :: ] section.

When you wish to put a class in a .h file, and it has a structure/class definition, you must put it as:

class account{  
 public:

int savings;

int checking;  
 void deposit(int amount);  
 };

This tells the compiler that the function deposit is there, but not to redifine it, as it is already in a different .c pp file.

# [ :: ]

The double colon is called the **Scope-Resolution Operator.** It is used when you have a global function/variable named the same as a local function/variable, and you wish to choose which one to access.

int count = 0;  
  
int main(void) {  
  int count = 0;  
  ::count = 1;  // set global count to 1  
  count = 2;    // set local count to 2  
  return 0;  
}

# [ INHERITANCE ]

Inheritance is an interesting little addition to c++. Essentially, it means that once class can 'copy paste' everything from another class into it and be able to use it as if it was actually defined in it.

class student{};

class graduateStudent : public student{};

In this situation, the class graduateStudent inherits any and all functions that were in student.

# [ PROTECTED : PUBLIC : FRIEND ]

When one creates a class, he can separate it into sections: public and private. When it is private, you are allowed access to those variables outside the class. When it is protected, you are not.

class student{

friend void initialize(int);

protected:  
  int gpa = 3;

public:

int WhatIsGPA(void){

return gpa;}  
}

In this examples, you can clearly see two sections, the public and the private. The actual gpa is in the protected section, so one outside this class cannot simply call s.gap = 4.0, and increase the gpa, since it is protected. The function WhatIsGPA, however, will let you at least see what the gpa is. But that is all.

The friend subclass lets you obtain access to protected values outside the class, and is declared by calling the function input, name, and output. In the above examples, a function called initialize, declared outside class, has access to change gpa directly.

# [ CONSTRUCTORS AND DESTRUCTORS ]

A constructor is an idea unique to classes, and makes up another reason why c++ is an object oriented programming language, while c is not. In c and c++, if you create a structure, you cannot initialize the variables within the structure. The values could, and are most likely, to be pointing to garbage space in memory. Accessing these variables without initializing them is a big no-no. In c, you initialize them where you create them. For instance,

void fn(){

student s1;

s1.gpa = 0

s1.hours = 0

}

But we don't want to have to do that in c++. This breaks the idea of OOP since we are relying on someone to make our function work properly. This is where constructors come in.

A constructor is something they added to c++ that is incredibly useful. Think of it as a function that is called by the compiler every that an object of that class is created. Take our student example. If we do this:

class student{

//CONSTRUCTER STARTS HERE

student(){

cout << "I am currently constructing. HI! :D"

gpa = 0;

}

//CONSTRUCTER ENDS HERE

protected:  
  int gpa = 3;

public:

int WhatIsGPA(void){

return gpa;}  
}

int main(void){

student s;

// RIGHT NOW

cout << s.WhatIsGPA()

}

Take a look at this code. The constructor is that thing that looks like a function with no output value. This is the case because it doesn't need one. Same with the input. YOU never call it. The compiler does. Since I put a COUT line in there, that means ever time you create a student object, that message will appear in the console. In this example, it will appear at the place marked //RIGHT NOW. The s.WhatIsGPA will output 0, since that is what gpa gets initialized as according to our constructor.

See how this is all this is slowly coming together to make an awesome class, object, system? This is the power of OOP. Let us move on to destructors.

Destructors are very similar in appearance to constructors, except that there is a small ~ sign in front of it. This is a cute Easter egg that they put into the code, since the ~ sign is the operational NOT symbol. A, ~A. (A, NOT A).

class student{

public:

//CONSTRUCTER STARTS HERE

student(){

cout << "I am currently constructing. HI! :D\n";

gpa = new int;

}

//CONSTRUCTER ENDS HERE

//DESTRUCTOR STARTS HERE

~student(){

cout << "I am currently deconstructing. BYE! D:\n";

delete gpa;

}

//DESTRUCTOR ENDS HERE

int WhatIsGPA(void){

return \*gpa;}

int \* gpa;

};

int fn(void){

int v = 4;

student s;

s.gpa = &v;

cout << s.WhatIsGPA() << endl;

return 0;

}

int main(void){

fn();

system("PAUSE");

return 0;

}

Running this code will output:

I am currently constructing. HI! :D

4

I am currently deconstructing. BYE! D:

In this way, when the LOCAL SCOPE of the object you have created disappears, ANY CALLS to the heap will automatically get cleared up with the de-constructor, so that YOU don't have to worry about it. These are all ways in which OOP makes our lives a hell of a lot easier.