# CSCI851 Spring-2021
# Advanced Programming
# Lecture 03

## C++ Foundations III:
## Pointers, classical arrays, and …

# A reminder on style

It’s helpful to have guidelines to follow.

Google’s C++ style may well be worth looking at an example.

https://google.github.io/styleguide/cppguide.html

That isn’t teaching you how to code in C++, it’s a reference guide on how to be consistent.

# Outline

Pointing to memory.

- Addresses.
- Pointers:
  - To variables.
  - Null.

Arrays and String things…

Arrays and pointers.

The operator `sizeof`.

Function pointers.

A shortcut to mushrooms.

Void.

# Pointing to memory

The primitive types map directly on to memory entities like bytes and words, entities that most processors are designed to work with.

This allows C++ to efficiently use the hardware, without there being an abstraction in between.

Memory is effectively seen as a sequence of bytes, each typed object is given a location in memory, and values are placed in such objects.

When we declare a variable we access it using the variable name.

But we can also access the address the variable references, and operate on that address.

- To access the address we use the address-of operator `&`.

So if we have an integer variable …

```
int value = 41;
```

… we can output the address of value as follows:

```
cout << &value << endl;
```

The address is probably mostly not useful to output, more on this soon...

But `&` can be used in another way …

```
int value = 41;
int &referenceValue = value;
```

This `referenceValue` is an alias, not an object, and operations on `referenceValue` are carried out on the variable/object to which the reference is bound.

So …

```
referenceValue = 100;
```

… changes the value of `value` to 100 and …

**Warning**: References must be initialised.

...

```
int otherValue = referenceValue;
```

... declares a new variable `otherValue` with an initial value equal to the value in `value`.

This might seem a little limited in use but we use this all the time in functions where we want the values being passed to change and don't want to have a complicated return object.

- As in Java, we can pass by reference or pass by value.

# Passing variables to functions

C++ has 2 ways to pass variables to functions:

- Pass by value, to be used when the function doesn't need to change the value of the arguments given to it.

```
return_type function_Name (type var1, type var2, …);
int get_larger (int A , int B);
```

- Pass by reference, to be used when the function may change the arguments.

```
return_type function_Name (type &var1, type &var2, …);
int sort (int &A , int &B);
```

But we can mix these …

```
return_type function_Name (type var1, type &var2, …);
int add_rate(int rate, int &value);
```

# Functions: Default Arguments

When calling functions, tailing arguments can be omitted if default values are declared in the function's parameters.

For example, a function declaration with default arguments:

```
void DrawString(char Text[], int Style = 0, int Size = 12, int  HSet = 0, int VSet = 0);
...
```

Valid calls to the above function declaration include:

```
DrawString("Enter your amount");
DrawString("You won", 3, 24);
DrawString("Increase your bid? ", 3);
```

# Pointers

Remember we can access the address the variable references, using `&`, and operate on that address.

- As part of this we have types that store addresses, pointers to type, or pointers.
  - Pointers are not aliases, they are actual objects.
- The following three forms mean the same thing…

```
int*  ptr;
int * ptr;
int *ptr;
```

- … that `ptr` is a pointer to an `int`, so it stores the location of an `int`.

The `*`, called the dereferencing operator, is tied to the variable name, not to the type name, so you have to be careful if you are declaring multiple pointers.

Use

```
int *ptr1, *ptr2;
```

Not

```
int *ptr1, ptr2;
```

Best not to mix declarations of pointers and non-pointers.

To set the value of a pointer we use the address-of operator `&`, as follows:

```
int value;
int *ptr;
ptr = &value;
```

And now we can make modifications to `value` through `ptr`, for example, …

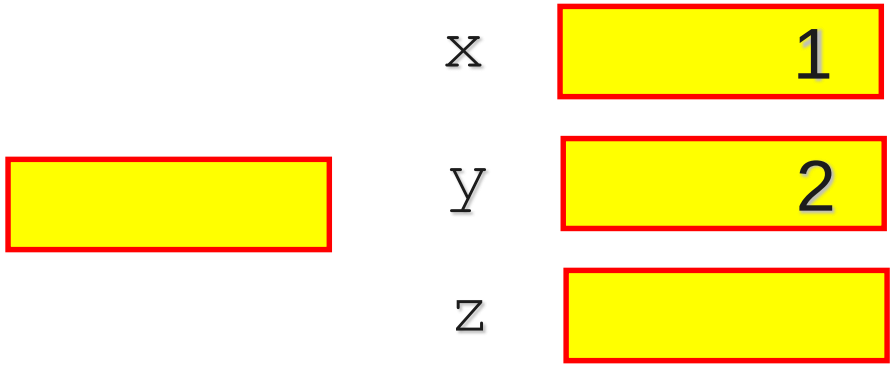
```
*ptr = 5;
```

… sets `value` to 5.

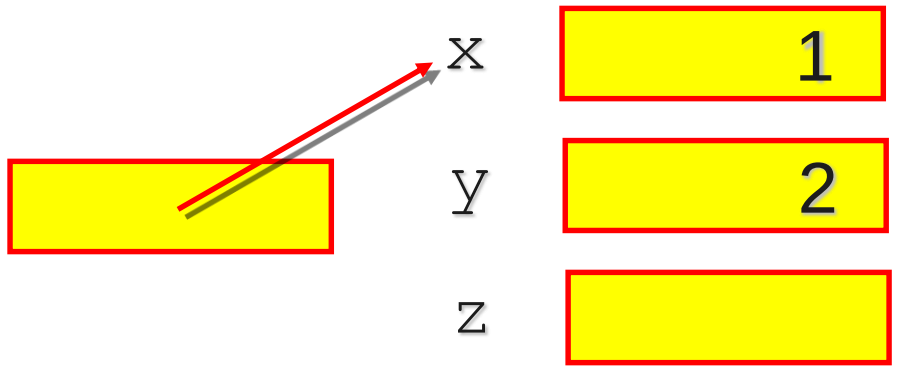
It's important to initialise pointers before you use them, otherwise you may get runtime errors based on using whatever happens to be the value in the location the pointer points to.

`*ptr` is the value stored at the address stored in `ptr`. So …

```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```

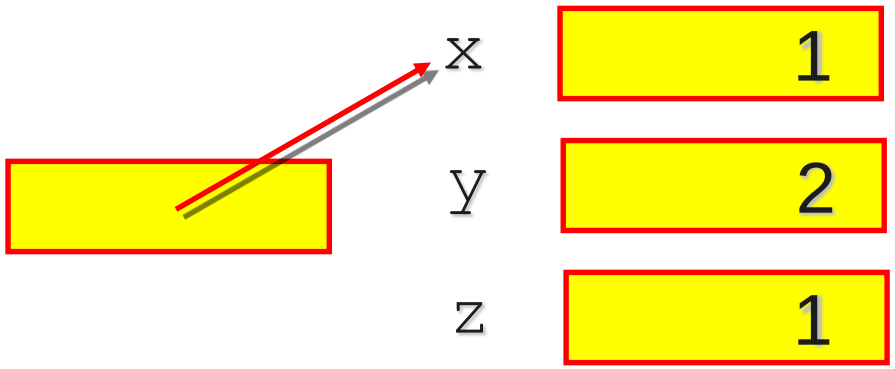


```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```



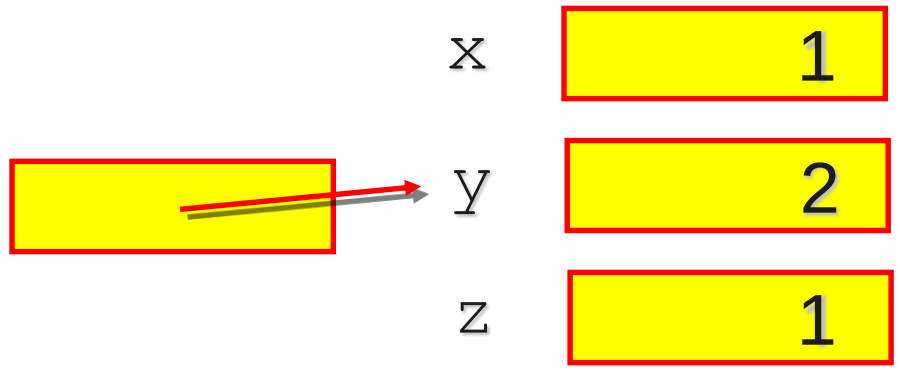

`p` is set to point at `x`.

```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```



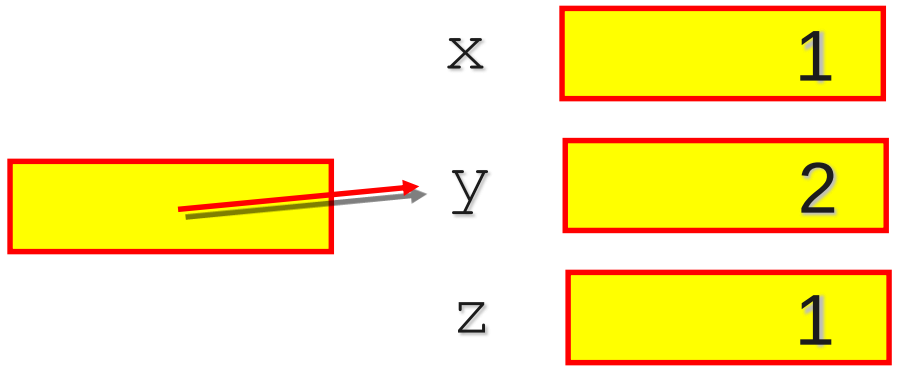

`z` is set to the value where `p` points at.

```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```




p is set to point at y

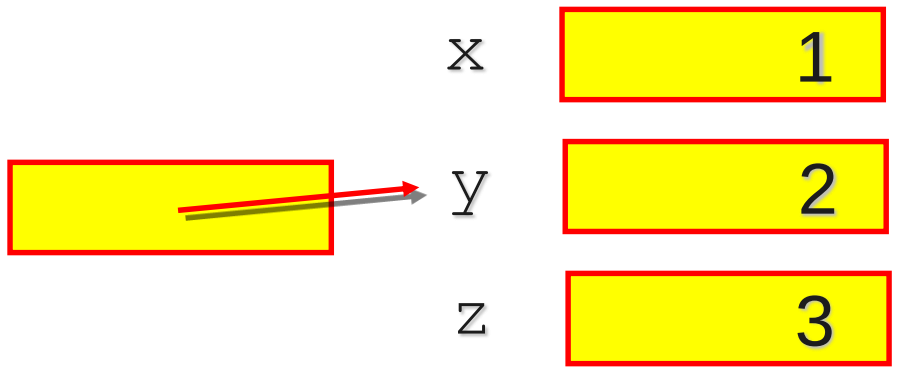
```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```




increment `z` by the value pointed at by `p`.

The += operator means increase left by the right.

```
int x=1, y=2, z;
int* p;

p = &x;
z = *p;
p = &y;
z += *p;
cout << z;
```




The output is the value 3.

# Null pointers and `nullptr`

It is possible to specify that a pointer doesn't point anywhere, by setting the pointer to the literal `0`, or the alias `NULL` defined as `0` in the `cstdlib` header.

```
int *ptr = 0;
```

```
int *ptr = nullptr;
```

The `nullptr` is a C++11 literal that can be converted to any other pointer type.

`nullptr` is always a pointer type, `NULL` is not.

Where we have overloaded operators, so the same function with different arguments, we don't have the problem of giving the function a null pointer and having it treated as a integer 0 provided we are using `nullptr`.

null-pointer.cpp

```
…
        int p1=0;
        int *p2=0;
        int p3=NULL;
        int *p4=nullptr;

        cout << p1 << " " << &p1 << endl;
        cout << p2 << " " << &p2 << endl;
        cout << p3 << " " << &p3 << endl;
        cout << p4 << " " << &p4 << endl;
…
```

```
$ CC null-pointer.cpp
"null-pointer.cpp", line 10: Error: nullptr is not defined.
1 Error(s) detected.
$ CC -std=c++11 null-pointer.cpp
```

This is CC compilation with the C++11 library included … this now works!

# Why use pointers? An example …

There will be more on pointers later in the subject, including topics such as smart pointers, which are used for handling dynamic objects.

For now, consider how they may be useful in sorting.

Consider that have 1000 large records that we are wanting to order.

Rather than swapping the records, we can swap the relatively small pointers instead.

# Passing by pointer vs passing by reference

Generally passing by reference is safer.

- Pointers can be null and can be reassigned, references cannot be either.

Unless it actually makes sense to allow for the possibility of the parameter being null, or you want to change where something points, **it's better to use constant or non-constant references to pass arguments.**

Note that references are generally going to be implemented using pointers.

# Arrays in C++

Speaking of sorting, that suggests we have multiple elements of the same, or at least comparable, type; which leads to arrays.

Arrays, references, and pointers, are all examples of **compound types**, types defined in terms of another type.

Arrays are collections of variables of the same type, roughly anyway, and of fixed size, usually, that we access by position.

– This is a pretty qualified statement.

It is possible to have dynamic arrays, but generally …

… if we aren't sure of the number of elements to be stored …

… we are better off using a `vector`, more on these later.

Why not use the dynamic vectors all the time?

- Because we may be able to optimise operations for the fixed number of elements that we have.
  - But we probably better using array containers rather than classical arrays for a fixed number of elements anyway.

# Setting up arrays …

Array declaration uses

```
type array_Name[dimension];
int class_Marks[10];
```

The dimension has to be known at compile time, so dimension needs to be a constant.

The `[]` is referred to as subscripting.

It's a good time to introduce the qualifier used to make sure something is constant, `const`.

# The `const` and `constexpr` qualifers

The keyword `const` is similar, but not the same as `final` in Java.

- C++11 has `final` too, more on this later because it's tied up with classes.

Operators cannot change an object with the const qualify. So …

```
int i = 10;
const int ci = 7;
int j = ci;
ci = 2;
```

… this last one isn't okay, and neither would leaving `ci` uninitialized.

The keyword `constexpr` is used for constant expressions, with values that cannot change but could be evaluated at compile time.

- It's an instruction to try to evaluate the expression at compile time.

Any `const` object initialized from a constant expression is a constant expression.

```
constexpr int ci = 7;
constexpr int sz = size();
```

These will compile with the `-std=c++11` flag on CC…

```
$ CC -std=c++11 file.cpp
```

More on these later.

# `const` and magic numbers

It's not unusual to want to have values that are used in several places throughout a file, or are going to be fixed.

It may be that you aren't quite sure what the value should be.

- Sizes, for arrays for example, are a typical example.

Or it's a recognised constant, like e or pi.

In both cases, it's better to have a constant variable that holds the value rather than using a magic number.

If I set something equal to 3.14, did I really mean that's pi and I just couldn't be bothered putting more digits?

Or is that value exactly 3.14?

If I'm using multiple sizes it's clearer if I set them in a single place and use them multiple times.

```
const int SIZE = 10;
```

# Initialising arrays

When we declare an array like …

```
const int postCodeLength = 4;
int postCode[postCodeLength];
```

… the memory location is set up but there is no initialisation.

To initialise all four locations to 0 …

```
int postCode[4]={0};
```

The size of an array is constant.

- The array name/identifier represents a memory address

To access an element of an array use its index

```
postCode[2] = postCode[1] + 1;
```

Note, the first element is `postCode[0]`.

# More initialising

There are a few different ways to initialise.

For an array of three `int`s with values 0, 1, and 2.

```
const unsigned sz =3;
int ia1[sz] = {0, 1, 2};
```

The size can be inferred from the initialiser …

```
int a2[] = {0, 1, 2};
```

But we might not have all the initial values so …

```
int a3[5] ={0, 1, 2};
string a4[3] = {"hi", "bye"};
```

Careful with the size …

```
int a5[2] = {0, 1, 2};
```

… interesting difference between CC and g++...

The unitialized parts are value-initialized, `int` to 0, `string` to an empty string.

# Character arrays are special

Character arrays can be initialised using a string literal, and they end with a null character `\0`.

- These are referred to as C-strings.

This can be explicit in element by element declarations.

```
char a1[] = {'C', '+', '+'};
char a2[] = {'C', '+', '+', '\0');
char a3[] = "C++";
const char a4[6]="123456";
```

The last one will complain because there is no space for the null to be added ☹

# Arrays and pointers

Arrays and pointers are quite closely related.

Mostly when we use an object of array type we are actually using a pointer to the first element of the array.

Note that arrays are, by default, passed by reference.

- Therefore arrays passed to functions can be changed by the function unless the keyword `const` is used.

# Example: Passing Arrays to Functions

```
void AddArray (
     int Size,              // size of the arrays
     const int A[ ],        // array passed as input
     const int B[ ],        // array passed as input
     int C[ ] )             // array passed for output
{
     for (int i=0; i<Size; i++)
            C[i] = A[i] + B[i];
}

int main(){
     const int ArySize = 5;
     int Ary1[ArySize]={1,2,3,4,5};
     int Ary2[ArySize]={6,7,8,9,10};
     int Ary3[ArySize];

     AddArray(ArySize, Ary1, Ary2, Ary3);
     ...
}
```

# Example: Passing Multidimensional Arrays to Functions

"Multidimensional arrays" must have their dimensions specified within the function's parameters, although the 1st dimension may be omitted, for example:

```
void print3DMatrix (const float A[][3][3]);

int main() {
     float Matrix[3][3][3]={{1,2,3},{4,5,6},{7,8,9}};
     print3DMatrix(Matrix);
     ...
}

void print3DMatrix (const float A[][3][3])
{
    for(int i=0;i<3;i++)
       for(int j=0;j<3;j++)
          for(int k=0;k<3;k++)
             cout << i << j << k << " = " << A[i][j][k] << endl;
}
```

Back to pointers : Consider the following function …

```
int SumArray(int arr[], int n)
{
    int i, sum=0;

    for (i=0;i<n;i++)
            sum += arr[i];
    return sum;
}
```

For the array

```
int A[10] = {1,2,3,4,5,6,7,8,9,10};
```

we can sum the entire array as

```
SumArray(A,10);
```

or

```
SumArray(&A[0],10);
```

But the same function can also sum the last nine elements using

```
SumArray(&A[1],9);
```

So an array name and an address seem to be equivalent.

And indeed, a pointer type can be referenced like an array.

```
int A[10];
int* B=A;
```

… meaning the pointer `B` gets the address of the array `A`, its name, so `B` can be used just like `A`.

If we dereference `B`, so `*B`, we get the value of `A[0]`.

But `B[0]` is the same as referring to `A[0]`.

Similarly `B[5]` is the same variable as `A[5]`, and `B[10]` is still off the end of the array.

But it gets worse, in that we can reference the address of other variables and do the same kind of position addition, subscripting, even though it's not an array.

So with

```
int A;
int* B=&A;
```

… we can use variables such as `B[7]` and `B[50]`.

When a C++ program references array elements, the compiler has to do some **pointer arithmetic**.

For example, `A[1]` refers to the memory location one after the address `A`.

In pointer arithmetic this is `*(A+1)`.

One what?

- One memory location.

What's that?

- Depends on type of `A`.
- The operator `sizeof` can help here.

# The `sizeof` operator

If you are doing pointer arithmetic the compiler will figure out how far to jump, but it is still sometimes useful to know how much space is taken by a variable.

C++ provides an operator called `sizeof` to give the programmer this information.

The operator usually appears looking like a function as in

```
sizeof(type)
sizeof(int)
```

```
sizeof(int)
```

.. returns the number of bytes that the `int` type occupies – in this particular implementation of C++.

The parentheses are not needed, but are usually used.

So ...

```
sizeof(type)
sizeof type
```

… both tell us the number of bytes for that type.

# `sizeof` a pointer …

What do you get if you apply `sizeof` to a pointer?

You can do something like …

```
cout << sizeof(int*) << endl;
```

Note that `sizeof` can act of a **type**, **variable** or **pointer** to a variable type …

So this is fine …

```
double value;
cout << sizeof(value) << sizeof(double) << sizeof(double*);
```

The sizeof a string is different because it’s a class and there is dynamically allocated memory in there.

# Function pointers

Sometimes we use pointers to refer to functions.

- That is, a pointer that points to the address of the executable code of the function.

The pointers can be used to:

- Call functions.
- Pass functions as arguments to other functions.

You cannot perform pointer arithmetic on pointers to functions.

Consider the following illustrations …

```
int  *f(int);
char (*g)(int);
char (*h)(int, int);
```

The first is not a pointer to a function, since the `()` operator has higher precedence than `*`. Rather this is a function `f` which takes an `int` and returns type `int*`.

The precedence means we need to bracket the pointer name, as in the second and third examples.

So:

- g is a pointer to a function taking an `int` and returning a `char`.
- h is a pointer to a function taking two `int`'s and returning a `char`.

Pointers to functions have types associated with both the return type and the parameter types of function.

Pointers to functions are particularly useful to describe how, in some function which takes them as an argument, we are to interpret some relationship.

- For example, the function describes a comparison rule.

```
int (*Compare)(const char*, const char*);
```

- This defines a function pointer `Compare` which can hold the address of any function that takes two constant character pointers as arguments and returns an integer.

A function pointer can also be defined and initialised in one line.

```
int (*Compare)(const char*, const char*) = strcmp;
```

When a function address is assigned to a function pointer, the two types must match.

The above definition is valid because `strcmp()` from `<stdlib>` has matching parameters and return type:

```
int strcmp(const char*, const char*);
```

Now `strcmp` can be either called directly, or indirectly via `Compare`.

The following three calls are equivalent:

```
strcmp("Cat", "Bat");      // direct
(*Compare)("Cat", "Bat"); // indirect
Compare("Cat", "Bat");     // indirect
```

A common use of a function pointer is to pass it as an argument to another function.

- This is because the receiving function requires different versions of the passed function in different circumstances.

# A shortcut to … mushrooms? Actually to vectors

We've talked about arrays, how to set them up and use them.

We've also mentioned that you are often better using vectors, so we are going to introduce vectors now.

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

int main()
{
   size_t size;
   cout << "Enter the size of the container: ";
   cin >> size;

   // get space for size integers and initialize them to 0
   vector<int> intArray( size );

   for(int i=0; i<size; ++i)
      intArray[i] = i;
}
```

intArray:vector<int>

The variable size is taken care of.

No need to use dynamic memory allocation.

To reference elements of the vector we can use `[]` again, like we did with arrays.

Later we will come across a more generic way of accessing containers, iterators.

# A special type: `void`?

We typically find `void` as the return type of functions that don't return values.

We don't define variables of type `void`.

There are no operations on `void`, and it doesn't have an associated value.

But, we can have void pointers…

# Void pointers : `void*`

A void pointer is used to hold the address of any type, but without the type being held being known.

- And you don't access content through the void pointer, dereferencing won't work.

This is usually used when we want to deal with memory as memory, without accessing the content.

- So in comparing locations for example…

Note: `sizeof(void*)` … still 4.

If we are access to access the content of the memory a void Pointer addresses, we need to type cast it first.

The cast

```
(type *)vptr
```

... will convert the `void` pointer `vptr` to a `type` pointer.

So we can have collections of void pointers to be used to store data of a range of types.

If we are to access the content of the memory a void Pointer addresses, we need to type cast it first.

```
int i = 5;
int *ip;
void *vp;
ip = &i;
vp = ip;
cout << *vp << endl;
cout << *((int*)vp )<< endl;
```

C++ cannot print the `void` but can print the `int`.

# Type conversion to a string …

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

string itos(int i) // convert int to string
{
        stringstream s;
        s << i;
        return s.str();
}

int main()
{
        int i = 127;
        string ss = itos(i);
        const char* p = ss.c_str();
        cout << ss << " " << p << "\n";
}
```

This code is from http://www.stroustrup.com/bs_faq2.html Changing `int` to something else will work as long as the something else has << overloaded for it!

# C++ Foundations IV: Dynamics

# Dynamics

If you sensibly can, you should be relying on the standard structures supported within the standard libraries.

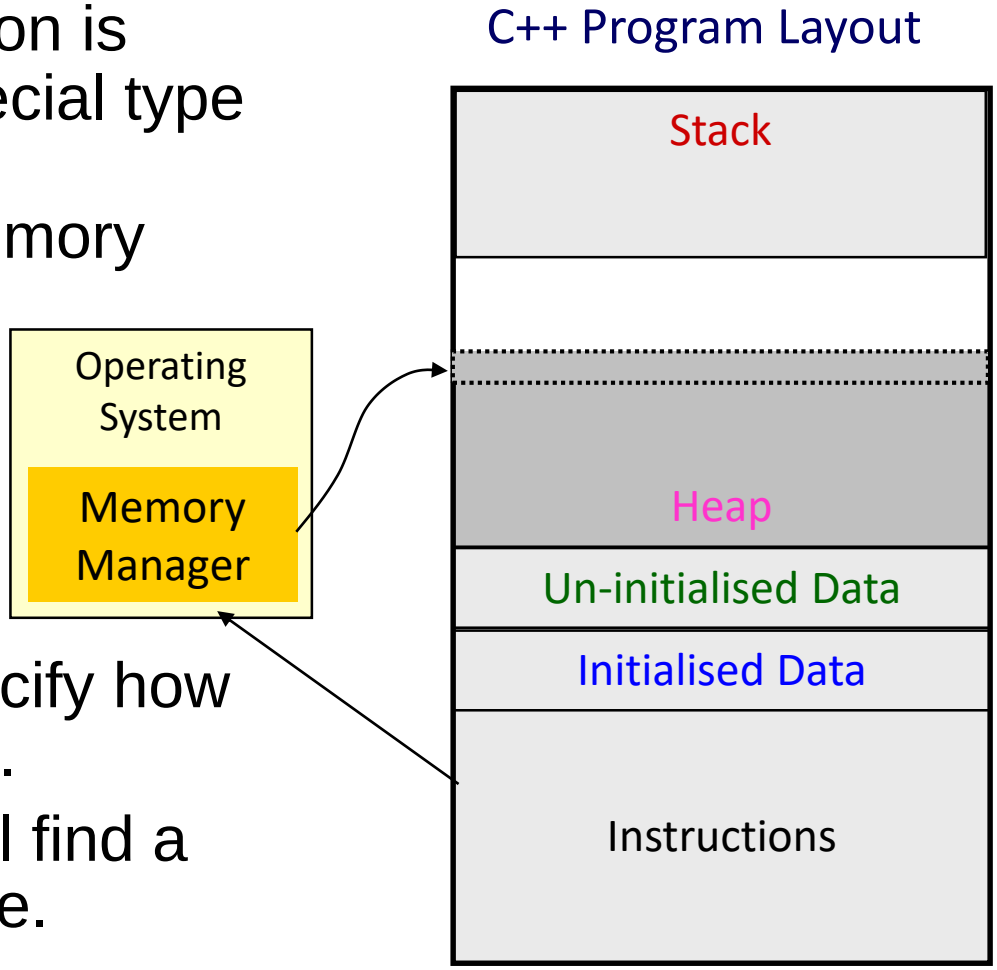
But, the standard tools are not appropriate in every situation and you may be able to do something more efficiently in a specific context.

In C++ you may want to dynamically manipulate memory.

- It’s kind of dangerous but can improve performance.

# Dynamic Memory Allocation

Dynamic memory allocation is carried out by using a special type of operator that directly communicate with the Memory Manager.




A programmer has to specify how much memory is required.

The memory manager will find a location currently available.

# Stack or Heap

The typical variables we have seen so far have been local.
They exist only while we are within scope, and the compiler deals with creating and destroying them.
These types of entities are stored in memory on the stack.

But there is other memory available, on the free store or heap.
Allocation of this memory is at run-time, and use of this memory is persistent so we need to explicitly say when we want to stop using it.

# `new` and `delete`

We firstly set up a pointer …

```
int *intpointer;
```

… then dynamically allocate memory with `new`.

```
intpointer = new int;
```

`new` is a type safe operation, it returns a pointer to the type given, `int` in this case.

- That pointer points to an object of the specified type, here an `int`, with the amount of memory required being automatically determined on the basis of the operand type.

If we use `new` we need to use `delete` to release the memory.

```
delete intpointer;
```

If we don't we get a **memory leak**.

Variables can be default initialised, with the default value type and sometimes location dependent.

- Built-in types defined outside function bodies are initialised to zero, those within are uninitialized, effectively having an undefined values.

We can also initialise the variables ourselves when we set up the memory,

```
int *p = new int(5);
```

The type specifier `auto` can come in useful again.

```
auto p1 = new auto(obj);
```

# Memory leaks …

Use `bcheck` to find them …

```
int *p;
p = new int(5);
cout << p << endl;
//  delete p;
```

Run `bcheck` ...

- See `a.out.errs`.
- The errs file give you details on where leaks occurred.

# new[] ...

To create a dynamic array we can use the `new[]` operator.

```
int *intVar;

intVar = new int[100]; // dynamic array

for(int i = 0; i < 100; ++i)
   intVar[i] = 25-i; // initialize the array

delete [] intVar; // frees the allocated array
```

# A final note on `delete[]` …

You have to be careful if you have something like a pointer to an array of pointers, …

```
Person **p = new Person* [2];
p[0] = new Person("Peter");
p[1] = new Person("Alex");
```

Using `delete[] p;` just causes the `p` pointer to be released, not the actual objects themselves.

You could step through the different index values and use `delete p[index]` on each.

Or, as will probably be discussed later, you could use a wrapper class.

# Example:

```
float **fVar;
fVar = new float* [10]; // allocate pointer
                        array, 10 float pointers

for(int i = 0; i < 10; ++i)
        fVar[i] = new float[10]; // allocate
                                    memory to each
.   .   .   .
.   .   .   .
for(int i = 0; i < 10; ++i)
        delete [] fVar[i];

delete [] fVar;
```

# Dynamic memory management: Problems

The textbook describes three common problems:

1. Forgetting to delete memory.
2. Using an object after it was deleted.
3. Deleting the same memory twice.

We will come back to smart pointers and their use in memory management for classes.

– They take care of these problem.

# Some faults: Seg. and Bus.

Segmentation faults occur when you try to use memory which does not belong to you, typically:

- Out of bounds array references.
- Reference through un-initialized or dangling pointers, the latter being pointers to already freed memory.

```
char *s = “Hello”;
*s=‘H’;
```

Compiled with `CC` or `g++` and we get a seg. fault at run time.

Segmentation faults are to do with memory access violations.

- You aren't allowed to access the memory specified.
- A compiler won't necessarily care or help.

Bus faults are similarly run time problems to do with accessing memory.

- But it relates to trying to access memory that cannot be physically addressed.
- The memory is invalid for the access type specified, usually to do with memory misalignment.

# C++ Foundations V:
# Control structures, loops, and typing

# Outline

Control structures.

Repetition structures: Loops.

`typedef`, `using`, and `auto` type.

Static variables in functions.

# Control structures: if

```
if (Boolean expression is true)
  statement;
```

For example:

```
if ( age >= 18 )
  cout << "You must vote!" << endl;
```

If there is more than one line you use { … }, and it's often a good idea using it anyway.

# Control structures: if-else

```
if (Boolean expression is true)
   statement;
else
   other statement;
```

For example:

```
if ( age >= 18 )
  cout << "You must vote" << endl;
else
  cout << "You cannot vote" << endl;
```

# Compound boolean expressions

AND: `&&`

```
if (age>=18 && countryCode==61)
```

OR: `||`

```
if (countryCode==61 || countryCode==64)
```

# Control structures: switch

The “if” statement is good for Boolean tests, where there are only two possible outcomes.

For multiple outcomes, the “switch - case” structure may be more suitable.

```
switch(variable)
{
   case 1:
              actions;
              break;
   case 2:
              actions;
              break;
   .   .   .
              break;
   default:
              cout << "The case is not defined" << endl;
}
```

# Switch in C++17

Switch has some additional functionality in C++17…

See, my preferred source,
http://en.cppreference.com/w/cpp/language/switch

It's a fairly minor change that supports the inclusion of an initialisation statement.

– Most likely to be useful for declaring a variable only to be used within the switch.

# Repetition statements

Repetition statements are intended to implement loops that repeat an action as long as some condition remains `true` .

There are three basic types of loop in C++:

- Pre-test loop `for`
- Pre-test loop `while`
- Post-test loop `do…while`

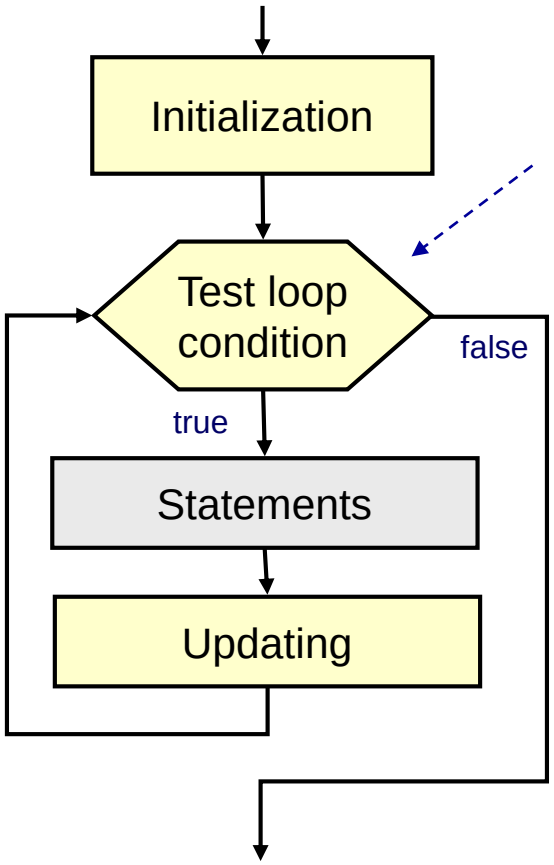
# Repetition statements

Although these have different syntax, all loops contain three components:

- Initialize loop
- Test loop condition
- Update

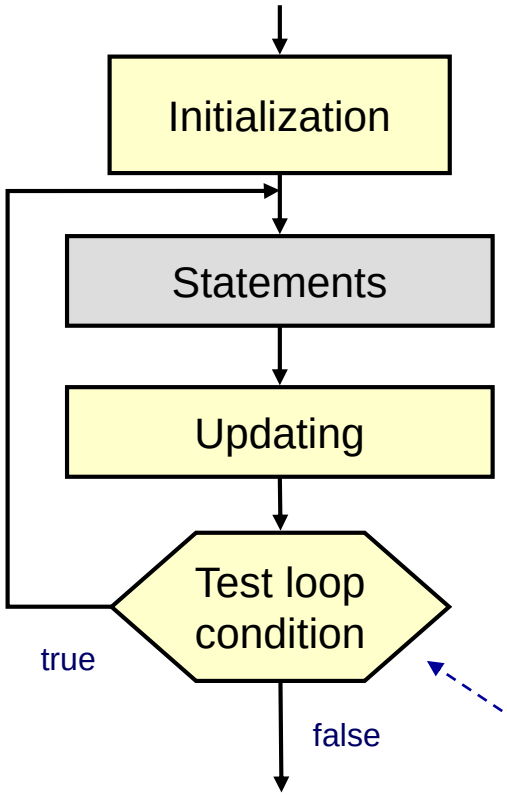
# Pre-test and Post-test loops

Pre-test loop




The loop condition is tested before the statements are executed

Post-test loop




The loop condition is tested after the statements are executed

# The `while` loop

What is the limit of the sequence?

`1 + 1/2 + 1/4 + 1/8 + ...  =>  1, 1.5, 1.75, 1.875, …`

```
#define DIF  0.000001  /* Change */
.  .   .

float oldResult = 0.0 , newResult = 1.0;
float x = 2.0;

while( newResult - oldResult > DIF )
{
    oldResult = newResult;

    newResult += 1/x;
    x *= 2.0;
}

cout<< "The limit is " << newResult;
```

```
old = 0.0  new=1.0    x=2.0

new - old > 0.0001  true
old = 1.0  new=1.5    x=4.0

new - old > 0.0001  true
old = 1.5  new=1.75   x=8.0

new - old > 0.0001  true
old = 1.75  new=1.875  x=16.0
.  .   .  .

new - old > 0.0001  false
STOP

output new
```

# The `for` loop

- The `for` loop is a version of a pre-test loop that has a more convenient syntax to implement a determined number of repetitions.

```
/*
 * sum of numbers from 1 to n
*/
#include <stdio.h>

int main(void)
{
    int n, sum, counter;

    printf("Enter n:  \n");
    scanf("%d", &n);

   sum = 0;
   counter = 1;
   while ( counter<=n )
   {
        sum += counter;
        .   .   .
        counter++;
   }
    . . .
    return (0);
}
```

```
/*
 * sum of numbers from 1 to n
*/
#include <stdio.h>

int main(void)
{
    int n, sum, counter;

    printf("Enter n: ");
    scanf("%d", &n);

    sum = 0;
    for ( counter=1; counter<=n; counter++ )
    {
        sum += counter;
        .   .   .
    }
    . . .
    return (0);
}
```

C style I/O

# Variations on the `for` loop

Several initialization expressions separated by commas.

```
for( int factorial=1, counter=1; counter <= n; ++counter)
    factorial *= counter;
```

No initialization expressions.

```
for(   ; n > 0; n-- )
    printf("*");
```

A simple implementation of a delay (the actual delay time is platform dependent).

```
for( int counter=0; counter < 1000; counter++ ) ;
```

An infinite loop (until it is terminated inside the loop body).

```
for(  ;  ;  )
{
   . . .
}
```

# The range `for` loop

C++ supports a range for statement that allows us to step through the elements in a sequence and operate on each in the same way.

```
for( declaration : expression)
    statement
```

The `declaration` defines the variable to be used when accessing the elements in the sequence, while `expression` is an object representing a sequence.

```
string str("This is a string");
for (char c : str)
      cout << c << endl;
```

# Control structures and repetition

We will go through some examples of control structures and repetition in one of the lecture/tutorials.

I expect you to have a play around with them by yourself anyway.

# typedef

You can rename basic data types …

```
typedef actual_type new_name;
typedef int number;
number one, two, three;
```

The data type `number` has the same properties as `int`.

Using `typedef` has the potential to make code difficult to read, and its use should primarily be restricted to header files which "end users" don't see.

– If used sensibly it can make code tidier.

The renaming `typedef` is probably particularly useful to get rid of deferencing operators that are likely to be around with points …

So we could do something like …

```
typedef DataType* DataPtr;
```

and then use the type to create pointer variables, an array for example …

```
DataPtr Index[10];
```

# Alias declaration …

There is another way of declaring an alias, one new to C++11.

```
typedef double other_double;
using OD = other_double;
OD one = 5.6;
cout << one << endl;
```

The `using` form is likely easier to get around the right way.

# `auto` typing

This is a nice feature of the C++11 standard.

The compiler figures out the type of something for us based on the initializer…

```
auto whatA = 5;
auto whatB = 5.6;
cout << sizeof(whatA) << sizeof(whatB);
```

Using the `sizeof` operator lets us see the variables are at least of different sizes

… but we can do better than that!

# Checking: The `typeid` operator…

This operator returns an object of type `type_info`, allowing us to compare types…

To use this we need to include the header `typeinfo`, and then we can do this …

```
auto whatA = 5;
auto whatB = 5.6;
cout << typeid(whatA).name() << endl;
cout << typeid(whatB).name() << endl;
```

Note the dot operator (`.`) is used to access a member function.

Actually `==` and `!=` are likely more useful operations for this type.

# Back to `auto` ...

The example we gave with an integer and a double was kind of trivial.

The auto type specifier is most useful when the type is either hard to know or hard to write.

```
template<class T> void printall(const vector<T>& v)
{
      for (auto p = v.begin(); p!=v.end(); ++p)
            cout << *p << endl;
}
```

```
template<class T> void printall(const vector<T>& v)
{
      for (typename vector<T>::const_iterator p =
                  v.begin(); p!=v.end(); ++p)
      cout << *p << "\n";
}
```

You can also use `auto` to grab the return type from an operation …

So, for example and following the textbook, if we have a string and want the length we can use the member function `size()` as follows:

```
string word = "elephant";
auto length = word.size();
```

The `size()` member function of the `string` class returns an object of `string::size_type`.

In the range for loop described earlier it's nice to use `auto` so the format can be the same for different sequence types.

```
string str("This is a string");
for (auto c : str)
     cout << c << endl;
```

# Compounding: Referencing, `const`, and `auto`

When we use `auto` on a reference, as in …

```
int integer = 0, &ref = integer;
auto a = ref;
```

… the type will be that of the referenced variable, so `int` here.

When it comes to `const` we need to consider:

- Top-level const: Where the object is a `const`.
- Low-level const: In compound types, with the pointer or reference being to a `const` object.

Why make the distinction here?

- The `const` survival differs between those two types when we, in particular here, `auto` type on a `const`, and generally when we copy…

For example with …

```
const int ci =  integer, &cr = ci;
auto b = ci;
auto c = cr;
auto d = &integer;
auto e = &ci;
```

… we will get `b` and `c` type `int`, `d` of type `int*`, and `e` of type `const int*`.

# More `auto` (C++14)

Since C++14 it's possible to use `auto` rather than return on a function declaration.

```
auto function(…)
```

The return type is now deduced from the operand of the return statement in the function.

So

```
return int;
```

… can be used and `int` inferred.

It can be used to provide abstraction in Lambda expressions in C++14, and as a template parameter in C++17.

# Still more on `auto`

See …

http://en.cppreference.com/w/cpp/language/auto

In particular, note that this is one part of C++ that has changed across C++14 and C++17.

That website has a good history section …

http://en.cppreference.com/w/cpp/language/history

# `auto` doesn't use …

… language recognition,

… pattern recognition,

… or mindreading or magic.

Using statements like …

```
auto number;
```

… or

```
auto x;
```

… isn't going to work.

# decltype

Another C++11 type specifier, one with some similarities to `auto`, is `decltype`.

This lets us base the type of a variable of a function calls return type, but initialise the variable to something else.

```
decltype(f()) variable = x;
```

Here `variable` will have the type returned by `f()`, although the compiler does not itself call `f()`.

# Another qualifier: `static` ...

Be careful with this one, it has a different meaning in the context of classes.

Static variables, declared as such in functions, persist beyond scope (`{}`), and aren't re-initialised with each call to the function.

```
size_t count_calls()
{
     static size_t ctr = 0;
     return ++ctr;
}

int main()
{
     for (size_t i = 0; i < 10; ++i)
          cout << count_calls() << endl;
     return 0;
}
```