CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S2c**)

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;
```

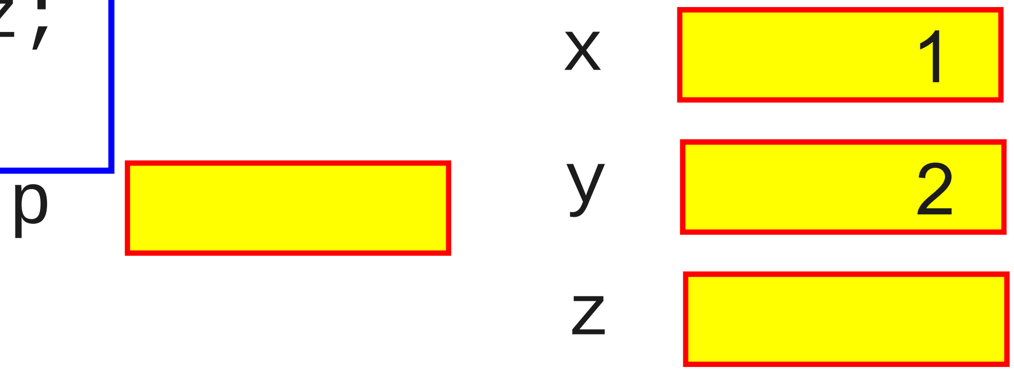
- And now we can make modifications to `value` through `ptr`, for example, …

```
*ptr = 5;
```

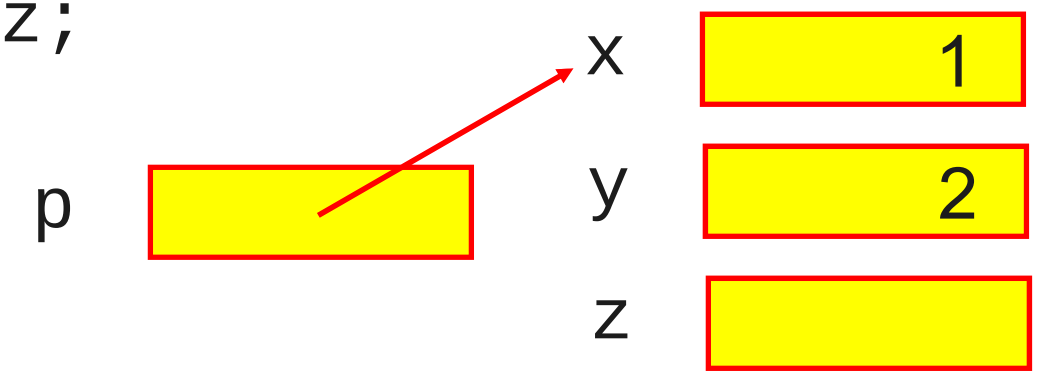
- … sets `value` to 5.
- It's important to initialise pointers before you use them, otherwise you may get runtime errors based on using whatever happens to be the value in the location the pointer points to.

- `*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;
```

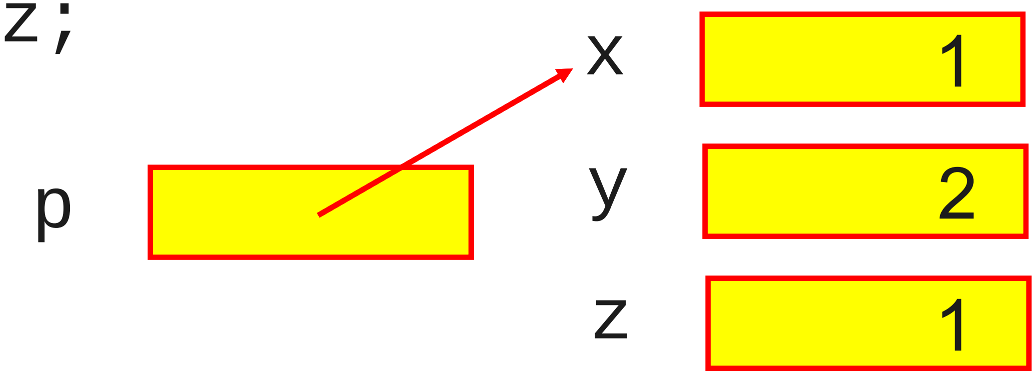


```
int x=1, y=2, z;
int* p;

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



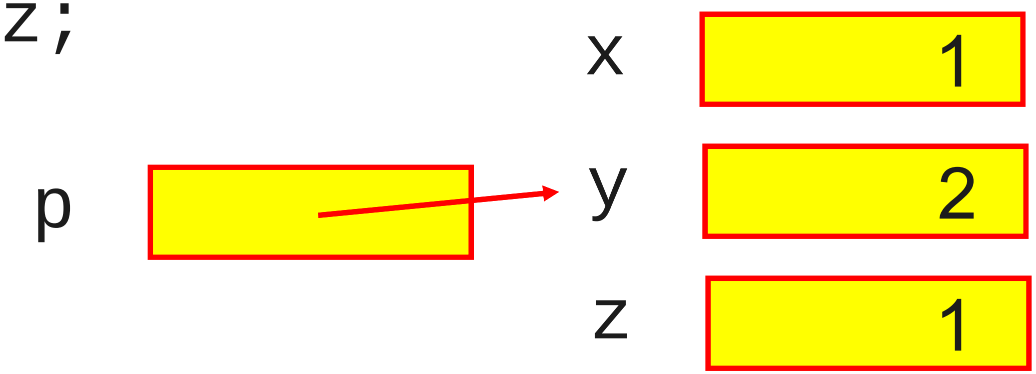
p is set to point at x.

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

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



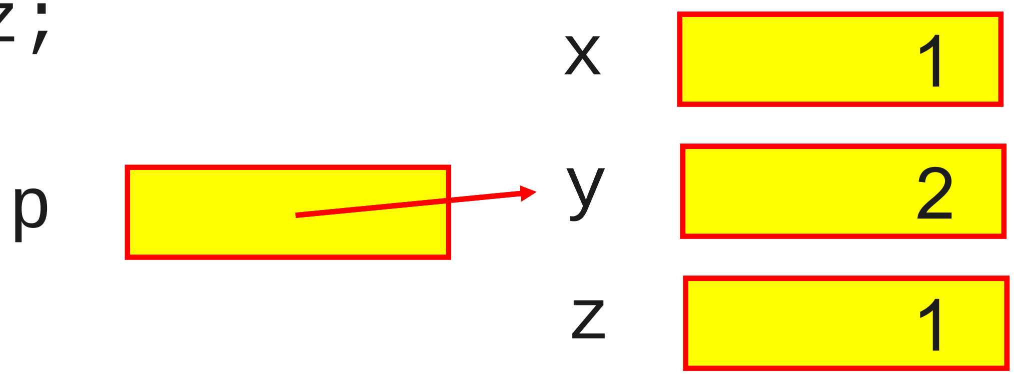
z is set to the value where p points at.

```
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

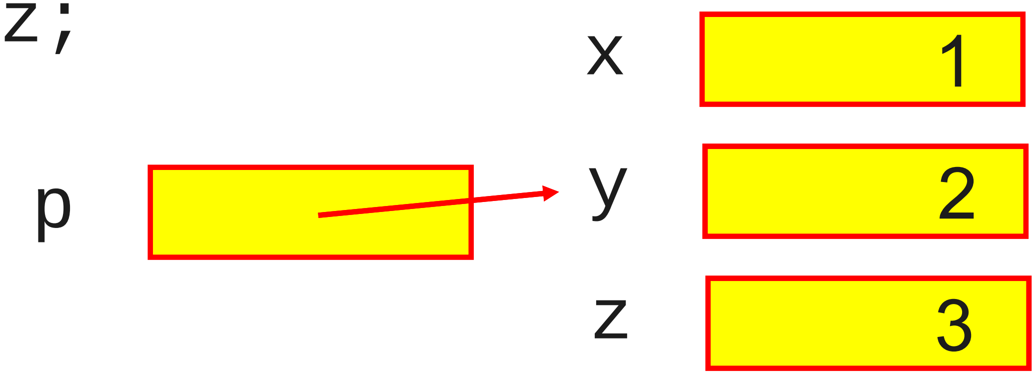
```
int x=1, y=2, z;
int* p;

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`.

```
…
        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;
…
```

null-pointer.cpp

```
$ 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! You should be okay with g++

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();
```

- 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 the same size repeatedly, 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 ☹

Dealing with c-strings…

Warning … against using c-strings… no length checking.

```
function( const char *first, const char *last )
{
  …
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );
}
```

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 subscripting again, so `[]`, like we did with arrays.
- Later we will come across a more generic way of accessing containers, iterators.

A special type: `void`?

- This is in Java so shouldn't be a big deal for the undergraduates.
- 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!