Introduction to Scientific Programming with C++

Session 2: More data types

Martin Uhrin

UCL

February 11-13th 2013

Table of Contents

Arrays

Definition

array a series of elements of the same type occupying a contiguous block of memory.

Format for declaring an array is:

```
type name[num_elements];
```

Where `type` is any valid data type and `num_elements` is a constant positive integer.

Arrays

Definition

array a series of elements of the same type occupying a contiguous block of memory.

Format for declaring an array is:

```
type name[num_elements];
```

Where `type` is any valid data type and `num_elements` is a constant positive integer. Some examples:

```
unsigned int lotteryNumbers[7];

double planetMasses[8];

const unsigned int numParticles = 128;
double xPositions[numParticles];
double yPositions[numParticles];
```

Last example shows how we can use constant variable as array size.

Initialising arrays

When declaring an array it can be initialised as follows:

```
unsigned int lotteryNumbers[7] = {16, 3, 28, 9, 24, 10, 8}
```

the size can be left out, in which case the number of values given is used:

```
unsigned int lotteryNumbers[] = {16, 3, 28, 9, 24, 10, 8}
```

Accessing elements

To access an element of an array the format is:

```
name[index]
```

Warning!

In C++ array numbering starts at 0! This is a huge source of confusion especially if you're used to a programming language like Fortran where arrays start at 1.

Accessing elements

To access an element of an array the format is:

```
name[index]
```

Warning!

In C++ array numbering starts at 0! This is a huge source of confusion especially if you're used to a programming language like Fortran where arrays start at 1.

For example, to read the 3^{rd} lottery number use:

```
unsigned int third = lotteryNumbers[2];
```

Accessing elements

To access an element of an array the format is:

```
name[index]
```

Warning!

In C++ array numbering starts at 0! This is a huge source of confusion especially if you're used to a programming language like Fortran where arrays start at 1.

For example, to read the 3^{rd} lottery number use:

```
unsigned int third = lotteryNumbers[2];
```

To write the 3^{rd} lottery number use:

```
lotteryNumber[2] = 23;
```

Accessing elements

To access an element of an array the format is:

```
name[index]
```

Warning!

In C++ array numbering starts at 0! This is a huge source of confusion especially if you're used to a programming language like Fortran where arrays start at 1.

For example, to read the 3^{rd} lottery number use:

```
unsigned int third = lotteryNumbers[2];
```

To write the 3^{rd} lottery number use:

```
lotteryNumber[2] = 23;
```

You can also access the elements using a variable:

```
for(int i = 0; i < 7; ++i)
   std::cout << lotteryNumbers[i] << " ";
```

Accessing elements

Array index out of bounds

Warning!

Be careful not to access elements past the end of an array. Consider:

```
const unsigned int numPlanets = 8;
double masses[numPlanets];
for(int i = 0; i <= numPlanets; ++i)
  masses[i] = random();
```

Seems ok, but wait. The last iteration will try to set `masses[8]` which is past the end of the array!

Multidimensional arrays

Because with only two friends 1D is sooo boring

Think of multidimensional arrays as being "arrays of arrays". An example:

```
const unsigned int numParticles = 10;
double positions[numParticles][3];  // positions x,y,z
double masses[numParticles];
double centreOfMass[3] = {0.0, 0.0, 0.0};

// Populate arrays with random masses and positions

for(int i = 0; i < numParticles; ++i)
{
  for(int dim = 0; dim < 3; ++dim)
  {
    centreOfMass[dim] += masses[i] * positions[i][dim] /
      numParticles;
  }
}

std::cout << "Centre of mass: " << centreOfMass[0] << " "
```

../code/2_more_data_types/lectures/centre_of_mass.cpp

Multidimensional arrays

Why stop at two

In theory you can have as many array dimensions as you want. For example a 3D array of ising spins could be represented as:

```
bool isingSpins[nX][nY][nZ];
```

In practice you have to worry about how much memory your array needs!

Multidimensional arrays

Why stop at two

In theory you can have as many array dimensions as you want. For example a 3D array of ising spins could be represented as:

```
bool isingSpins[nX][nY][nZ];
```

In practice you have to worry about how much memory your array needs! If you want to find out use:

```
std::cout << "Need: " << sizeof(bool) * nX * nY * nZ <<
  " bytes";
```

std::array

Like c-style arrays but safer and easier to use

```
std::array<double,5> myArray={2.3, 4.2, 7.5, 8.2, 9.1};
```

These arrays are just as fast, have built in protection and are easier to manipulate. They protect against code such as this:

```
double SingleValue = myArray[8];
```

Accessing elements outside a C-style array is dangerous.
To use them the following header file must be included

```
#include<array>
```

You may also need to compile with the option –std=+11 for g++.

Strings

A string of characters, used to store text.

```
std::string message = "Physics rocks!";
```

Strings use the c-style char arrays internally but are much easier to use. To use strings make sure to use the following include:

```
#include <string>  // At the top of your file
```

String variables

All strung out

So what can we do with strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Equivalent to above
```

String variables

All strung out

So what can we do with strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Equivalent to above
```

Concatenate (add two or more together)

```
std::string fullName = firstName + " " + lastName;
```

Notice ability to mix string variables and literals (i.e. things in quotes).

String variables

All strung out

So what can we do with strings?

Initialise

```
std::string firstName = "Bjarne";
std::string lastName("Stroustrup"); // Equivalent to above
```

Concatenate (add two or more together)

```
std::string fullName = firstName + " " + lastName;
```

Notice ability to mix string variables and literals (i.e. things in quotes).

Read from user

```
std::cout << "Enter first name: "
std::cin >> firstName;
```

Pointers

Pointers are low level C-stlye code that can be error prone and difficult to use. In modern C++ they can usually be avoided.

Every variable lives at a memory address. A pointer is a special data type that stores such an addresses.

Pointer declaration

To declare a pointer the format is:

```
type * name;
```

This tells the compiler that `name` is a pointer that points to the address of a variable of type `type`. Got it?

Pointers

Pointers are low level C-stlye code that can be error prone and difficult to use. In modern C++ they can usually be avoided.

Every variable lives at a memory address. A pointer is a special data type that stores such an addresses.

Pointer declaration

To declare a pointer the format is:

```
type * name;
```

This tells the compiler that `name` is a pointer that points to the address of a variable of type `type`. Got it?
Some examples:

```
int * pointerToInt;
std::string * pointerToString;
```

Using pointers

Pointer initialisation

To set pointers we can use the reference operator: & (read "address of").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

line 2 tells the compiler to:

Using pointers

Pointer initialisation

To set pointers we can use the reference operator: & (read "address of").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

line 2 tells the compiler to:

1. Create a pointer named `spinsPointer` that points to an `int`.

Using pointers

Pointer initialisation

To set pointers we can use the reference operator: & (read "address of").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

line 2 tells the compiler to:

1. Create a pointer named `spinsPointer` that points to an `int`.
2. Set it to address of `upSpins`.

Using pointers

Pointer initialisation

To set pointers we can use the reference operator: & (read "address of").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

line 2 tells the compiler to:

1. Create a pointer named spinsPointer that points to an int.
2. Set it to address of upSpins.

So what's in a pointer?

```
std::cout << "Address is: "
  << spinsPointer
  << "\n";
```

Using pointers

Pointer initialisation

To set pointers we can use the reference operator: & (read "address of").

```
int upSpins = 10;
int * spinsPointer = &upSpins;
```

line 2 tells the compiler to:

1. Create a pointer named `spinsPointer` that points to an `int`.
2. Set it to address of `upSpins`.

So what's in a pointer?

```
std::cout << "Address is: "
  << spinsPointer
  << "\n";
```

Output: `Address is: 0x00CBF748`
This is how C++ prints memory addresses.

Using pointers

Reading the value

To access, use the the dereference operator: ∗ (read ”value pointed by”).

```
std::cout << "Value is: "
   << *spinsPointer
```

Using pointers

Reading the value

To access, use the the dereference operator: $*$ (read "value pointed by").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Using pointers

Reading the value

To access, use the the dereference operator: $*$ (read "value pointed by").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Setting the value

To set the value pointed to by a pointer also use dereference operator:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Using pointers

Reading the value

To access, use the the dereference operator: ∗ (read ”value pointed by”).

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is:  10`

Setting the value

To set the value pointed to by a pointer also use dereference operator:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Output: `New upSpins:  20`

Using pointers

Reading the value

To access, use the the dereference operator: * (read "value pointed by").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Setting the value

To set the value pointed to by a pointer also use dereference operator:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Output: `New upSpins: 20`

Assigning pointers

Later on we may want to assign the pointer to point to a different value:

```
int downSpins = 7;
spinsPointer = &downSpins;
std::cout << "Down spins : "
  << *spinsPointer;
```

Using pointers

Reading the value

To access, use the the dereference operator: ∗ (read "value pointed by").

```
std::cout << "Value is: "
  << *spinsPointer
```

Output: `Value is: 10`

Setting the value

To set the value pointed to by a pointer also use dereference operator:

```
*spinsPointer = 20;
std::cout << "New upSpins: "
  << upSpins
```

Output: `New upSpins: 20`

Assigning pointers

Later on we may want to assign the pointer to point to a different value:

```
int downSpins = 7;
spinsPointer = &downSpins;
std::cout << "Down spins : "
  << *spinsPointer;
```

Output: `Down spins: 7`

Pointers

Pointers can be very dangerous they have literally cost lives!

Warning!

C++ pointers can be dangerous! Consider:

```
int * upSpinsPointer;
std::cout << *upSpinsPointer;
```

I've asked for the value pointed by `upSpinsPointer`. But what's it pointing to? It could be a valid address or garbage.

Pointers

Pointers can be very dangerous they have literally cost lives!

Warning!

C++ pointers can be dangerous! Consider:

```
int * upSpinsPointer;
std::cout << *upSpinsPointer;
```

I've asked for the value pointed by `upSpinsPointer`. But what's it pointing to? It could be a valid address or garbage.

Do

Set pointers to the special value `0` upon declaration which indicates that it is not pointing to a valid memory address:

```
int * upSpinsPointer = 0;
```

This is called a null pointer. The program will crash immediately if you try to dereference it and you will find out straight away what went wrong.

Dynamic memory

So far our programs have used a fixed amount of memory equal to the total of all declared variables. Sometimes we don't know how much memory we will need before the program starts.

Dynamic memory

So far our programs have used a fixed amount of memory equal to the total of all declared variables. Sometimes we don't know how much memory we will need before the program starts. Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

But we can only create arrays of known, *constant*, size!

Dynamic memory

So far our programs have used a fixed amount of memory equal to the total of all declared variables. Sometimes we don't know how much memory we will need before the program starts. Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

But we can only create arrays of known, *constant*, size!
Solution: dynamic memory.

`new` and `new[]` operators

To allocate dynamic memory the format is:

```
  pointer = new type;                // single variable
  pointer = new type[num_elements]; // array
```

Dynamic memory

So far our programs have used a fixed amount of memory equal to the total of all declared variables. Sometimes we don't know how much memory we will need before the program starts. Consider:

```
unsigned int spinChainLength;
std::cout << "Enter spin chain length: ";
std::cin >> spinChainLength;
bool spinChain[spinChainLength]; // ERROR!
```

But we can only create arrays of known, *constant*, size!
Solution: dynamic memory.

`new` and `new[]` operators

To allocate dynamic memory the format is:

```
  pointer = new type;                // single variable
  pointer = new type[num_elements]; // array
```

Let's try again:

```
bool * spinChain = new bool[spinChainLength]; // Good
```

Dynamic memory

Don't forget to clean up

`delete` and `delete[]` operators

To free dynamic memory the format is:

```
delete pointer;   // single variable
delete[] pointer; // array
```

Make sure you use the correct version, otherwise bad things *will* happen.

Dynamic memory

Don't forget to clean up

`delete` and `delete[]` operators

To free dynamic memory the format is:

```
delete pointer;    // single variable
delete[] pointer;  // array
```

Make sure you use the correct version, otherwise bad things *will* happen.

Don't

Forget the free your memory once you're done with it. Otherwise it will leak and you won't get it back until your program ends!

Dynamic memory

Don't forget to clean up

`delete` and `delete[]` operators

To free dynamic memory the format is:

```
delete pointer;    // single variable
delete[] pointer; // array
```

Make sure you use the correct version, otherwise bad things *will* happen.

Don't

Forget the free your memory once you're done with it. Otherwise it will leak and you won't get it back until your program ends!

Do

Set the pointer to 0 after the memory has been freed:

```
delete[] spinChain;
spinChain = 0;
```

std::vector

Dynamic arrays made easy, all the memory allocation with new[] and delete[] is done for you.

```
std::vector<double> myVector;
```

Elements can easily be added on the fly using push_back().

```
myVector.push_back(2.3);
myVector.push_back(3.3);
myVector.push_back(5.6);
```

The elements can be accessed as normal with [] or myVector.at() if you want to use a more modern syntax.

In order to use vectors you must include the following header.

```
#include<vector>
```

References

More safe than keeping your money in a Swiss bank account

A reference is similar to a pointer only more limited, and more safe.

Reference declaration and initialisation

To declare a reference and initialise it the format is:

```
type & name = variable_name;
```

This tells the compiler that name is a reference to an existing variable called `variable_name` which is of type `type`. References *cannot* be uninitialised!

Example:

```
int upSpins = 10;
int & spinsReference = upSpins;
int & downSpins; // Error: cannot be uninitialised
```

References

More safe than keeping your money in a Swiss bank account

A reference is similar to a pointer only more limited, and more safe.

Reference declaration and initialisation

To declare a reference and initialise it the format is:

```
type & name = variable_name;
```

This tells the compiler that name is a reference to an existing variable called `variable_name` which is of type `type`. References *cannot* be uninitialised!

Example:

```
int upSpins = 10;
int & spinsReference = upSpins;
int & downSpins; // Error: cannot be uninitialised
```

Using references

Once a reference is declared it can be used almost exactly as if it were an ordinary variable.

Pointers vs. references

Practical comparison

```
#include <iostream>

int main()
{
  int upSpins = 10;
  int downSpins = 7;
  int * spinsPointer = &upSpins;

  std::cout << "Address is: "
    << spinsPointer
    << "\n";

  std::cout << "Value is: "
    << *spinsPointer
    << "\n";

  *spinsPointer = 20;
  std::cout << "New upSpins: "
    << upSpins
    << "\n";

  spinsPointer = &downSpins;
  std::cout << "Down spins : "
    << *spinsPointer;

  return 0;
}
```

```
#include <iostream>

int main()
{
  int upSpins = 10;
  int downSpins = 7;
  int & spinsReference = upSpins;

//  std::cout << "Address is: "
//    << spinsReference
//    << "\n";

  std::cout << "Value is: "
    << spinsReference
    << "\n";

  spinsReference = 20;
  std::cout << "New upSpins: "
    << upSpins
    << "\n";

//  spinsReference = downSpins;
//  std::cout << "Down spins : "
//    << spinsReference;

  return 0;
}
```

Changing variable value in a function

Consider:

```
void runningSum(int sum, int value)
{
  sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i < 100; ++i)
    runningSum(sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is:  0`

But wait. What's happened? `sum` is still 0[1].

[1] Although our program fails to calculate this sum, legend has it that Gauss produced the correct answer within seconds when asked during a primary school lesson.

Changing variable value in a function

Consider:

```
void runningSum(int sum, int value)
{
  sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i < 100; ++i)
    runningSum(sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is:  0`

But wait. What's happened? `sum` is still 0[1].

Problem is that `runningSum` got a *copy* of the value in `sum` at the time it was called.

[1] Although our program fails to calculate this sum, legend has it that Gauss produced the correct answer within seconds when asked during a primary school lesson.

Passing using pointers

Let's try again but this time using a pointer:

```
void runningSum(int * sum, int
    value)
{
  *sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i <= 100; ++i)
    runningSum(&sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is: 5050`

Passing using references

Let's try again but this time using a references:

```
void runningSum(int & sum, int
    value)
{
  sum += value;
}
int main()
{
  int sum = 0;
  for(int i = 1; i <= 100; ++i)
    runningSum(sum, i);

  std::cout << "Sum is: "
    << sum << "\n";
}
```

Output: `Sum is: 5050`

Passing by value, pointer and reference

What's the difference?

So we have three ways to pass parameters:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

Imagine you're the function and I'm passing you an Edward Hopper[2] painting I keep on my living room wall:

[2] Nighthawks to be exact (`http://en.wikipedia.org/wiki/Nighthawks`)

Passing by value, pointer and reference

What's the difference?

So we have three ways to pass parameters:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

Imagine you're the function and I'm passing you an Edward Hopper[2] painting I keep on my living room wall:

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.

[2] Nighthawks to be exact (http://en.wikipedia.org/wiki/Nighthawks)

Passing by value, pointer and reference

What's the difference?

So we have three ways to pass parameters:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

Imagine you're the function and I'm passing you an Edward Hopper[2] painting I keep on my living room wall:

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.
2. I give you my address and you can view and change the painting by visiting (dereferencing) my address.

[2] Nighthawks to be exact (http://en.wikipedia.org/wiki/Nighthawks)

Passing by value, pointer and reference

What's the difference?

So we have three ways to pass parameters:

```
/*1.*/ void runningSum(int sum, int value);  // by value
/*2.*/ void runningSum(int * sum, int value);// by pointer
/*3.*/ void runningSum(int & sum, int value);// by reference
```

Imagine you're the function and I'm passing you an Edward Hopper[2] painting I keep on my living room wall:

1. I make an exact duplicate and give it to you. Any changes you make to yours don't affect mine.
2. I give you my address and you can view and change the painting by visiting (dereferencing) my address.
3. I create a second painting that is quantum entangled with mine. Any changes you make to yours affect mine instantly.

[2] Nighthawks to be exact (`http://en.wikipedia.org/wiki/Nighthawks`)