LECTURE 2

Lifetime of variables

Creating objects - memory layout

- Global variable - static storage
- Local variable - **stack storage** - object is created in a block ({ . . . }) and automatically destroyed at the end of the block (when goes out of scope)
- Using the `new` operator - free store or **heap** - must be destroyed using the `delete` operator (more about `new` and `delete` under pointers)

```
1  int main()
2  {
3     int i=0;
4
5     if(i==0)
6     {
7        int j=0; // 'j' is created in this block
8
9        i+=j; // this is OK, this block is part of 'i's scope
10             // and hence, 'i' is accessible from here
11    } // 'j' is destroyed at this point
12
13    ++j; // ERROR, 'j' no longer exists
14 }
```

Types of the core language

Recall: when declaring a variable, its type must be specified.
The following types are provided by the core language

- `bool` - represents logical value, takes two possible values: `true` and `false`
- `signed int`, or just `int` - integer, can be positive or negative
- `unsigned int` - integer, only positive, if negative value is assigned, the compiler generates an error message
- `float` - single-precision floating point number
- `double` - double-precision floating point number
- `char` - represents a narrow character
- `void` - "absence of any value", no variable can be declared `void`, however functions can be declared to return `void`
- `enum` - enumerated types (see lecture 2)
- `long ...` - extended precision versions available for `int` and `double`

Using types in general - specifiers

Variables can be declared

- `const` - the value of a `const` object cannot be modified, moreover, a non-`const` member function of a `const` object cannot be called (see later)
  Note: a const object must be initialised at declaration
- `mutable` - denotes a data member that can be modified even if the object is `const`
- `static` - static lifetime objects, its value is remembered when the execution returns to the scope it has been declared in
  Note: `static` data members must be initialised outside of the class declaration (see later)
  `static` member functions will be introduced later.

```
1 for(unsigned int i=0; i<3; ++i)
2   for(unsigned int j=0; j<3; ++j)
3   {
4     static unsigned int iCounter(0);
5     std::cout << ++iCounter << ", ";
6   }
```

```
  1, 2, 3, 4, 5, 6, 7, 8, 9,
```

References

```
1 int a; // 'a' plain simple variable
2 int & b=a; // 'b' is a reference to 'a'
3
4 a=1;
5 std::cout << "After a=1, a: " << a << " b: " << b << std::endl;
6
7 b=2;
8 std::cout << "After b=2, a: " << a << " b: " << b << std::endl;
```

```
After a=1, a: 1 b: 1
After b=2, a: 2 b: 2
```

- `b` is defined to be a reference to `a`, `b` is nothing but another name one can use to refer to `a`.
- The reference **must be initialized at declaration**.
- Note: once `b` is initialized to be the reference to `a`, it will always refer to the variable `a`, and cannot be re-assigned to another variable. (This is NOT equivalent to non-modifiable value!)
- Reference to base can actually take a reference to derived (see later).
- Return type of a function can be reference only if it refers to object existing before the function is called (e.g. class member functions, see later).

References - examples

```
1  // Vector is some vector type with some
2  // algebraic operations defined
3
4  Vector & payoff(const Vector & rFactors)
5  {
6    Vector vPayoff;
7    // some code using, but not modifying rFactors
8    // and writing into vPayoff
9    //...
10   return vPayoff; // ERROR!!
11 }
```

- `vPayoff` is created in the scope of the function
- `vPayoff` is destroyed at the end of the function's scope
- the function returns reference to a non-existing object - **ERROR!**

References - examples

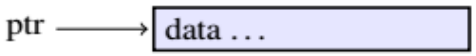
```
1  Vector & payoff(const Vector & rFactors, Vector & rPayoff)
2  {
3    // some code using, but not modifying rFactors
4    // and writing into rPayoff
5    //...
6    return rPayoff;
7  }
```

- `rPayoff` is created before the function call
- the results are written into `rPayoff`
- the function returns reference to `rPayoff`, it's OK now, `rPayoff` is not destroyed at the end of the function execution

An example:

```
1  Vector vFactors(n), vPayoff(m), vSum(m);
2  vSum*=0.0;
3
4  for(int i=0; i<iSampleSize; ++i)
5  {
6    // generate vFactors ...
7    vSum+=payoff(vFactors,vPayoff);
8  }
```

Pointers



What is a pointer?

- Pointer to an object is the value representing the **memory address** of the object.
- In many cases, it's cheaper to pass the pointer than the whole object.
- The object (both data and member functions) is accessible through its pointer.

How to create a pointer?

- Declaration of a pointer to an object of type `int`:

```
1  int * iptr;
```

- Take the address of an existing object using the **address operator** `&`:

```
1  int * iptr= &i; //where i has been declared earlier
```

- Allocate memory for a new object using operator `new` (see later).
- One can define pointers to function, this is one way to pass functions as arguments.

How to access the object through its pointer?

- By the **dereference operator** `*`:

```
1  int i=*iptr;
```

Pointers - as function arguments

Extract from `Lecture010203/Example4.hpp`:

```
1  void swap3(int *a, int *b)
2  {
3     int temp=*a;
4     *a=*b;
5     *b=temp;
6  }
7  void Example4()
8  {
9    int a(1), b(2);
10   std::cout<< "Before calling swap3 on a and b," << std::endl;
11   std::cout<< "a: " << a << " b: " << b << std::endl;
12   swap3(&a,&b); // note: addresses are passed
13   std::cout<< "After calling swap3 on a and b," << std::endl;
14   std::cout<< "a: " << a << " b: " << b << std::endl;
15 }
```

Output:

```
Before calling swap3 on a and b,
a: 2 b: 1
After calling swap3 on a and b,
a: 1 b: 2
```

Pointers - examples

Extract from `Lecture010203/Example7.hpp`:

```
1  int a(1);
2  int * ptr1=&a; //the address of 'a' is taken
3  std::cout<< "Value of 'a' after first assignment: "
4      << a << std::endl;
5  std::cout<< "Value of 'a' accessed through its address: "
6      << *ptr1 << std::endl;
7  *ptr1+=2; //value of 'a' is changed
8  std::cout<< "Changing the value of 'a' through its "
9      << "address: " << a << std::endl;
10 int * ptr2=new int(3); //new variable on the heap
11 std::cout<< "The value stored at the address ptr2: "
12     << *ptr2 << std::endl;
13 delete ptr2; //must be deleted at some point
14 std::cout<< "The value stored at the address ptr2 "
15     << "after releasing memory: " << *ptr2 << std::endl;
```

Output:

```
Value of 'a' after first assignment: 1
Value of 'a' accessed through its address: 1
Changing the value of 'a' through its address: 3
The value stored at the address ptr2: 3
The value stored at the address ptr2 after
releasing memory: 3
```

- The memory for the local variables is allocated and freed automatically on the stack.
- The memory on the heap can be allocated using the operator `new` and **must be** freed using the operator `delete` before the end of the program.
- The use of `new` and `delete` requires care, even if `delete` is part of the code, the execution might take turns different from what normally expected, resulting in **memory leak**.
- One can allocate memory for arrays and free using `delete[]`:

```
1  T * p=new T[n];
2  //...
3  delete [] p;
```

- Safer alternatives - **smart pointers**, for example `std::auto_ptr`, `boost::scoped_ptr`, `boost::shared_ptr`, etc.

- Smart pointers are like standard pointers, the operators `*`, `->` behave like in the standard case

- There is no need to `delete` smart pointers, they do that themselves when going out of scope

Extract from `Lecture010203/Example8.hpp`:

```
  namespace ptrnamespace = boost;
typedef ptrnamespace::shared_ptr<int> int_ptr;
//defining an alternative name for the type name

int_ptr returnPointer(int Arg){
   int_ptr pVal(new int(Arg));
   return pVal;
}

void Example8() {
  int_ptr pTemp=returnPointer(1);
  std::cout<< "Value: " << *pTemp << std::endl;
} // 'pTemp' is out of scope and automatically deleted here
```

```
typedef double (*MyFunType)(double);
```

- It's possible to define type for functions with a given signature (argument types and return type)
- In the above example, the type name `MyFunType` is the type of functions taking one `double` and returning `double`
- This is useful for passing functions as arguments.
- See the example on the next slide.

Pointers to functions

```
typedef double (*MyFunType)(double);

// function declarations
double fn1(double);
double fn2(double);

void Example9() {
  header(9);
  MyFunType fnPtr1=&fn1; //fn1 has been declared, can be used
  MyFunType fnPtr2=fn2; //fn2 has been declared, can be used
  std::cout << "fnPtr1 evaluated at 3.0: "
       << fnPtr1(3.0) << std::endl;
  std::cout << "fnPtr2 evaluated at 3.0: "
       << fnPtr2(3.0) << std::endl;
}

// function implementation
double fn1(double dArg){
  return dArg+2.0;}
double fn2(double dArg){
  return dArg*dArg;}
```

```
fnPtr1 evaluated at 3.0: 5
fnPtr2 evaluated at 3.0: 9
```

Pointers to functions

```
1  typedef double (*PayoffFunction)(double);
2
3  double monteCarlo(int iSampleSize, PayoffFunction payoffArg)
4  {
5     double dVal(0.0);
6     for(int i=0; i<iSampleSize; ++i)
7     {
8        double dStock;
9        //... generate stock price dStock
10       dVal+=payoffArg(dStock);
11    }
12    return dVal;
13 }
14
15 double particularPayoff(double Arg)
16 { //... }
17
18 int main()
19 {
20    int iSamplesize(1000);
21    double dVal=monteCarlo(iSampleSize, particularPayoff);
22    //....
23 }
```

- `MonteCarlo()` takes an argument that is a pointer to a function
- the implementation of `MonteCarlo()` contains the common part
- the payoff-specific part is factorised out (particular payoff)

Namespaces

- With namespaces, one can group functions, classes, typedefs, etc. under a name
- Structure code, divide the global scope into sub-scopes
- A way to avoid name conflicts

```
1  namespace mynamespace
2  {
3     int iA; //declaring a variable
4     int myFun(int);// declaring a function
5  }
6
7  int mynamespace::myFun(int iArg)
8  { //... } // implementing the function
9
10 {
11    int iB=mynamespace::iA; //accessing objects from a namespace
12 }
13
14 {
15    using mynamespace::iA; // introducing particular names
16                           // from a namespace into this scope
17    int iB=iA;
18 }
19
20 {
21    using namespace mynamespace; // introduce an entire namespace
22                                 // into this scope
23 }
```

Naming conventions

There are different name conventions, you might need to adapt to the one used. Examples.

- type names: mixed case, starting with upper-case: `MyType`
- variable names: mixed case, starting with lower case: `sampleSize`
- "Hungarian notation": `iSampleSize` - int, `dStockPrice` - double, `rPayoffVec` - reference, `pMyClass` - pointer, `m_iDataMember` - integer data member
- function names: mixed case, starting with lower case: `monteCarlo()`
- named constants: all upper-case: `MAX_INT`
- names of namespaces: all lower-case
- variables with long scope must have long name, variable with short scope can have short names
- names of boolean variables and methods should start with the prefix `is`: `isKnockedIn`

Summary

stack storage	heap storage
core types	specifiers
pointers	`new`, `delete`
pointers to functions	address `&`, dereference a pointer `*`
reference	
namespaces	`using`
naming conventions	

Types

We have seen types provided by the core language. In C++ it's possible to implement custom types.

What is a type?

- Collection of states described by some data members
- Behaviour implemented by member functions (methods)

Classes - `ComplexNumber`, the interface

Example: `ComplexNumber` is a class with a few members. (The example is taken from Exceptional C++ by Herb Sutter with minor modifications.)

```
1  #include <iostream>
2
3  class ComplexNumber
4  {
5  public:
6     ComplexNumber();
7     explicit ComplexNumber(double , double=0.0);
8
9     ComplexNumber & operator+=(const ComplexNumber &);
10    ComplexNumber & operator-=(const ComplexNumber &);
11    ComplexNumber & operator*=(const ComplexNumber &);
12    ComplexNumber & operator/=(const ComplexNumber &);
13
14    ComplexNumber & operator+=(const double &);
15    ComplexNumber & operator-=(const double &);
16    ComplexNumber & operator*=(const double &);
17    ComplexNumber & operator/=(const double &);
18
19    std::ostream & print(std::ostream &) const;
20
21 private:
22    double dRe_;
23    double dIm_;
24 };
```

General properties of classes

- we distinguish between **class** (the type) and **instance of a class** (a particular value)
- classes can have `public`, `private` and `protected` **members** or **properties** (members of each category can exists simultaneously in one class)
- `public` members are accessible without restrictions (by any function or type)
- `private` members of an instance `C` of class `type_of_C` are only accessible by instances of the same class `type_of_C` and objects declared as `friend` of `type_of_C` (examples . . . maybe later).

- `protected` members of an instance `C` of class `type_of_C` are accessible by `friend`'s and instances of classes derived from `type_of_C`. (...later)
- members of a class can be **data** and **function**.
- unless otherwise specified, class members are private by default
- **struct** is similar to class, however struct members are public by default
- the keywords `class` and `struct` are required when declaring a class or struct respectively.
- members can be accessed (given the permission) using the operator ".": `C.data` or `C.fun();` or
- members can be accessed through pointers using the operator `->`: `C_ptr->data`, `C_ptr->fun()`

Members - in general

Member functions

- member functions of a class are part of its namespace
- members functions take the instance they are called from as an argument, although the instance is not explicitly listed in the argument list
- the instance is a const argument, if the member is declared `const`, note the syntax: `fun(...) const;`
- member functions can return reference to the instance the function was called from, this can be achieved by dereferencing the pointer `this` to self.
- Don't give access to the class' non-`public` members if it's not necessary. In other words: "Prefer non-member non-`friend` functions to member functions." (Scott Meyers, Effective C++)

Members - in general

Data members

- data members can be plain values, references, pointers and const versions of these
- const and reference members must be declared at the construction of the instance
- it's recommended to make data members non-`public`, although in some cases `public` data members are useful (e.g. `first` and `second` are `public` members of `pair<T1,T2>`)

Classes - `ComplexNumber`, members

```
1  class ComplexNumber
2  {
3
4  private:
5     double dRe_;
6     double dIm_;
7  };
```

- `ComplexNumber` has two `private` members, both are data members

Classes - `ComplexNumber`, members

```
1  class ComplexNumber
2  {
3  public:
4     ComplexNumber();
5     explicit ComplexNumber(double , double=0.0);
6
7     ComplexNumber & operator+=(const ComplexNumber &);
8
9     std::ostream & print(std::ostream &) const;
```

- `ComplexNumber` a few member functions, each of them is `public`
- the member `print()` is a const function (does not modify the data members), the other functions or non-const members.

Member functions - constructors & destructors

- **constructors** initialize instances of classes, **destructors** take care of destroying the instance
- constructors and destructors return nothing
- constructors initialize data members, if initialization is not done explicitly, the compiler will call the default constructor of the data members (even if there are assignments in the function body)
- initializing data from constructors has a special syntax (see examples)
- **default constructors** take no arguments
- If no constructor is declared, the compiler will create a default constructor, which initializes the data members calling their default constructors
- **copy constructors** take an instance of the same type (usually by const reference) and create a new instance from that
- If no copy constructor is declared, the compiler will create one, which initializes the data members calling their copy constructors.

- If no destructor is declared, the compiler will create one.
- Note that the compiler generated default constructor, copy constructor and destructor might not work properly in certain cases. (See Effective C++ by Scott Meyers for further details.)

Classes - `ComplexNumber`, constructors and destructors

```
1  class ComplexNumber
2  {
3  public:
4      ComplexNumber();
5      explicit ComplexNumber(double , double=0.0);
```

- A **default constructor** is declared (an defined somewhere), the compiler generated one might not initialize the data members to 0
- A second constructor is declared, takes two `double`'s, specifies a default value to the second argument.
- The argument with a default value might be omitted from the call, and its default value will be passed automatically.
- Note the `explicit` keyword. It protects against implicit type conversion (see Effective C++ by Scott Meyers, or Exceptional C++ by Herb Sutter for details).
- No **copy constructor** is declared, the one generated by the compiler will copy the data members, which is fine here.
- No **destructor** is declared, the compiler will generate one.

Classes - `ComplexNumber`, constructors and destructors

The class is implemented in `Practical02/ComplexNumber.cpp`:

```
1  #include "ComplexNumber.hpp"
2
3
4  ComplexNumber::ComplexNumber() : dRe_(0.0), dIm_(0.0) {}
5
6  ComplexNumber::ComplexNumber(double dReArg, double dImArg)
7      : dRe_(dReArg), dIm_(dImArg) {}
```

- The usual practice is to put the class declaration into a header file and the implementation into a `cpp` file.
- Note that the header file must be `#include`'ed in the `cpp` file to give access to the declarations.
- A **default constructor** initializes both data members to 0. Nothing is done beyond initialization, that's why the function body is empty.
- The second constructor initializes the data members to the values provided by the arguments. If no second argument is specified by the call, the default value 0 will be taken. The function body is empty.
- Note the use of the namespace `ComplexNumber::` when accessing the function members of `ComplexNumber` for implementation.

Special members - overloaded operators

- one can overload most of the C++ operators (see lecture 2 for a list of available operators)
- the declaration of operator members have a special syntax:

```
1 class ComplexNumber
2 {
3 public:
4    ComplexNumber & operator+=(const ComplexNumber &);
```

- i.e. operator members have a special name, built of the keyword `operator` and the sign of the operator.
- the typical two argument operators (e.g. +=, −=, <, <= etc.) can be called two in equivalent ways, e.g.: `C1<C2` is equivalent to `C1.operator<(C2)`
- a special operator is the `()`
- operator `()` can be called in the form: `C1.operator()(arguments)` or `C1(arguments)`
- the latter form looks like a calling a simple function named `C1`, that's why classes with operator `()` implemented are referred to as function objects.
- the operator = with argument (const reference to) the user-defined class is the **copy assignment**. If no copy assignment is declared, the compiler will generate one.

Special members - operator () example

Example: Call option payoff

```
1  class CallOption
2  {
3  public:
4     CallOption(double dArg=1.0) : dK_(dArg) {}
5
6     double operator()(double dArg) {
7        return (dArg<dK_) ? 0.0 : dArg-dK_;
8     }
9
10 private:
11    double dK_;
12 };
```

- The class `CallOption` has an overloaded operator `()` member, taking a `double` and returning a `double`
- The operator `()` is called in the following example.

```
1  // construction and usage
2  CallOption Call1(1.0); //construction
3  cout << "Payoff at S=1.5, K=1 is " << Call1(1.5) << endl;
4  // alternatively, using a temp object:
5  cout << "Payoff at S=1.5, K=1 is " << CallOption(1.0)(1.5) << endl;
```

ComplexNumber - operator implementation

```
1  ComplexNumber & ComplexNumber::operator+=(const ComplexNumber & cnArg)
2  {
3     dRe_+=cnArg.dRe_;
4     dIm_+=cnArg.dIm_;
5     return *this;
6  }
```

- operator += was made a member, because it accesses private members
- it returns a reference to the instance it was called from using the dereferenced `this` pointer
- this makes expressions like `(a+=b) *=c` valid
- once operator += is declared as member, the operator + can be implemented using operator +=, i.e. without accessing non-`public` members of `ComplexNumber`, therefore it is declared as global function and implemented as follows:

```
1  ComplexNumber operator+(const ComplexNumber & Arg1,
2           const ComplexNumber & Arg2)
3  {
4     ComplexNumber Res(Arg1);
5     return Res+=Arg2;
6  }
```

ComplexNumber - operator implementation

- the operator << must be overloaded to display instances of user defined classes
- for displaying, usually the access of non-`public` members is required
- one way is to declare the global operator << a `friend` of the new class (see later)
- or implement a convenience `public` member:

```
1  std::ostream & ComplexNumber::print(std::ostream & os) const
2  {
3     return os << "(" << dRe_ << ", " << dIm_ << "i)";
4  }
```

- now, operator << can be implemented using the member `print`
- note that this global function takes an `ostream` by reference and returns a reference to the same `ostream`
- the second argument of type `ComplexNumber` is the one to be displayed

```
1  std::ostream & operator<<(std::ostream & os, const ComplexNumber & cnArg)
2  {
3     return cnArg.print(os);
4  }
```

Summary

custom types	`class`
data member	member function
`public`	`private`
constructor	destructor
operators	overloading
`operator()`	`ostream` operator <<
`this`	