Lecture-7

Operators Overloading
&
Function Overloading

O-O Polymorphism

- Polymorphism means that it is possible to 'overload' the symbols we use in a program so that the same symbol can have different meanings in different contexts.
- A symbol can be either
  - operators
  - function (methods) names
- This means that we can have two types of polymorphism
  - Operator Overloading
  - function Overloading

Operators Overloading

- Operator Overloading means making the compiler's built in operator symbols work with classes
- Operator symbols are things like + - = * etc
- We saw how these work with data types such as `int float` etc
- We can also do this with variable of different types by the use of coercion
- To do this with classes we may want to use a syntax as follows

```
Object3 = Object1 + Object2
```

So how do we do this?

- To allow a similar operation as shown previously we have to code the mechanism ourselves
- In effect the + and = operator must be overloaded to be methods of the class
- By overloading operators in this way we can give the classes in a system a common interface, allowing us to perform similar operations on a range of different objects.

Overloading and the assignment Operator

- The = operator is already overloaded to work with objects as well as simple data types
- This has already been see with the copy constructor

```
class_name object1 = object2;
```

- This means that object2 is instantiated with the same state as object1, We can also do this in normal code for example

```
object1 = object2;
```

Overloading and the assignment operator

- This behavior is default for the = operator however we can program our own overloaded = operator within the class.

- This is required when pointers are used as the object will point only to the memory address and if the initial object is destroyed so will the memory address where the pointer points to, Therefore code will be required to create and copy the pointer data

Overloading operators for a class

- Assignment (=) is a fundamental operation of all classes and types, therefore it is provided by the compiler
- However no default behavior is provided for the other operators when used with classes
- Other operations must be specified by the programmer in the class
- Consider the class below

- The class could possibly require a + operator to add two colours together or a - operator to subtract them.

Colour
-R: float -G: float -B: float -A: float
+Colour(r,g,b,c) +Colour + (Colour) +=Colour

C++ Syntax

- The syntax required for operator overloading is :
  - The operator keyword
  - And one of the 40 operator which may be overloaded as shown below

```
new delete + - * / % += -= *= /= %= ++ --
= ^ & | ~ ^= &= |= << >> >>= <<=
< > == != <= >= ! && ||->* ->
, comma operator
() function call operator
[] subscripting operator
```

The default assignment operator

```
class Point2D
{ private :
    float x;
    float y;
 public :
    void SetX(float Xin);
    float GetX(void);
    float GetY(void);
};
void Point2D::SetXY(float Xin,float Yin)
{  x = Xin;
   y = Yin;  }
void Point2D::GetX(float Yin)
{   return x;   }
float Point2D::GetY(void)
{  return y;    }
```

A Simple Program

```
#include”Point2D.h”
#include<iostream>
use namespace std;

int main(void)
{
   Point2D  point, point2;
   point1.setXY(201,213)
   point2 = point1;
   cout << point1.x << “   “ << point1.y << endl;
   cout << point2.x << “   “ << point2.y << endl;
};
```

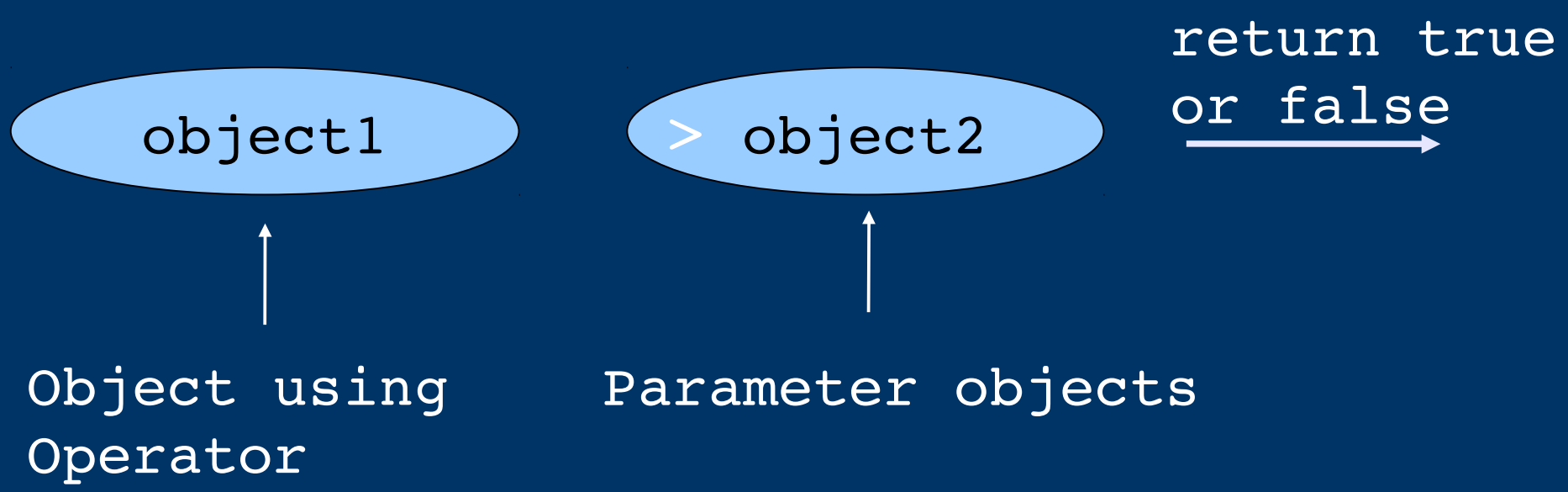
```
Program Output
201   413
201   413
```

Syntax for Operator Overloading

- The syntax for operator overloading is as follows

```
return_type operator symbol(parameter list ...)
```

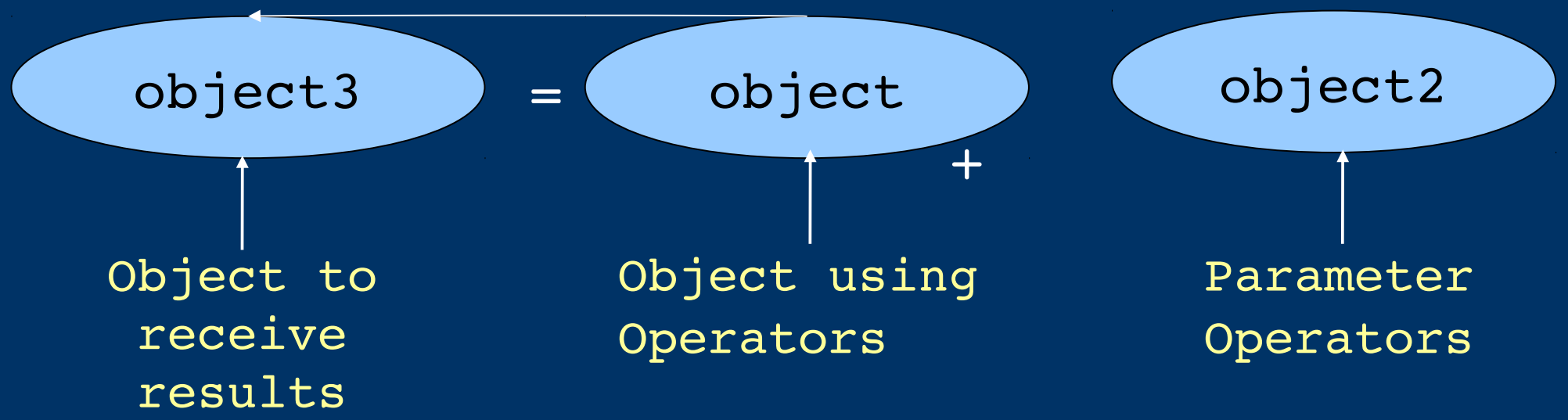
- The return type for a relational operator is a Boolean value



Arithmetic Operator Overloading

- However arithmetic operators return an object of the appropriate class as shown below

```
a temporary object is returned
from the operator method
```



The parameter List

- The parameter list usually consists of an object of the class
- This is passed by reference rather than by value to avoid making a copy of the object
- Therefore the `const` prefix is used to stop the reference object being modified in the class
- For example we may have the following:

less than (relational)

```
int operator < (const Object &object);
```

minus (arithmetic)

```
Object operator -(const Object &object);
```

Overloading relational operators

- These are the easiest to implement as they only return Boolean values
- Using the previous Point Class we can add an equality operator (==)

```
bool operator == (const Point &rhs);
```

- And the method itself

```
bool Point::operator==(const Point& rhs)
{
   return (x==rhs.x && y==rhs.y)
}
```

Using the relational operator

- In the code we can now write the following

```
if(point1 == point2)
  cout << "points are the same" <<endl;
else
  cout << "points are different"<<endl;
```

Overloading Arithmetic Operator

- As mentioned previously the arithmetic operator must return an object of the same class
- So for an + Point operator we would do the following

```
Point operator + (const Point &rhs)
```

- and for the method

```
Point Point::operator +(const Point &rhs)
{
return Point(x+rhs.x,y+rhs.y);
}
```

Using arithmetic Overloading

- Now we can add two Point classes together using the following syntax

```
Point point1,point2,point3;
point3 = point1+point2;
```

- Now the object `point3` contains the sum of the x points and the y points of objects point1 and point2.

Overloading the insertion and extraction operators (1)

- To overload the insertion and extraction operators we need to declare the following in the .h file

```
friend std::ostream& operator<<(std::ostream&
                                output, const Point3& s);
friend std::istream& operator>>(std::istream&
                                output, Point3 &s);
```

Overloading the insertion and extraction operators (2)

- And in the cpp file the actual methods

```
ostream& operator<<(ostream& output, const Point & s)
{
 return output<<"["<<s.x<<","<<s.y<<","<<s.z<<"]";
}
istream& operator>>(istream& input, Point &s)
{
input >> s.x >>s.y  >>s.z;
return input;
}
```

- we can now use these in code

```
Point3D point1;

cin >> point1;

cout << point1<<endl;
```

Function Overloading

- Parametric Overloading allows us to have multiple methods of the same name with different parameters
- We have already seen this with the constructor method in classes
-

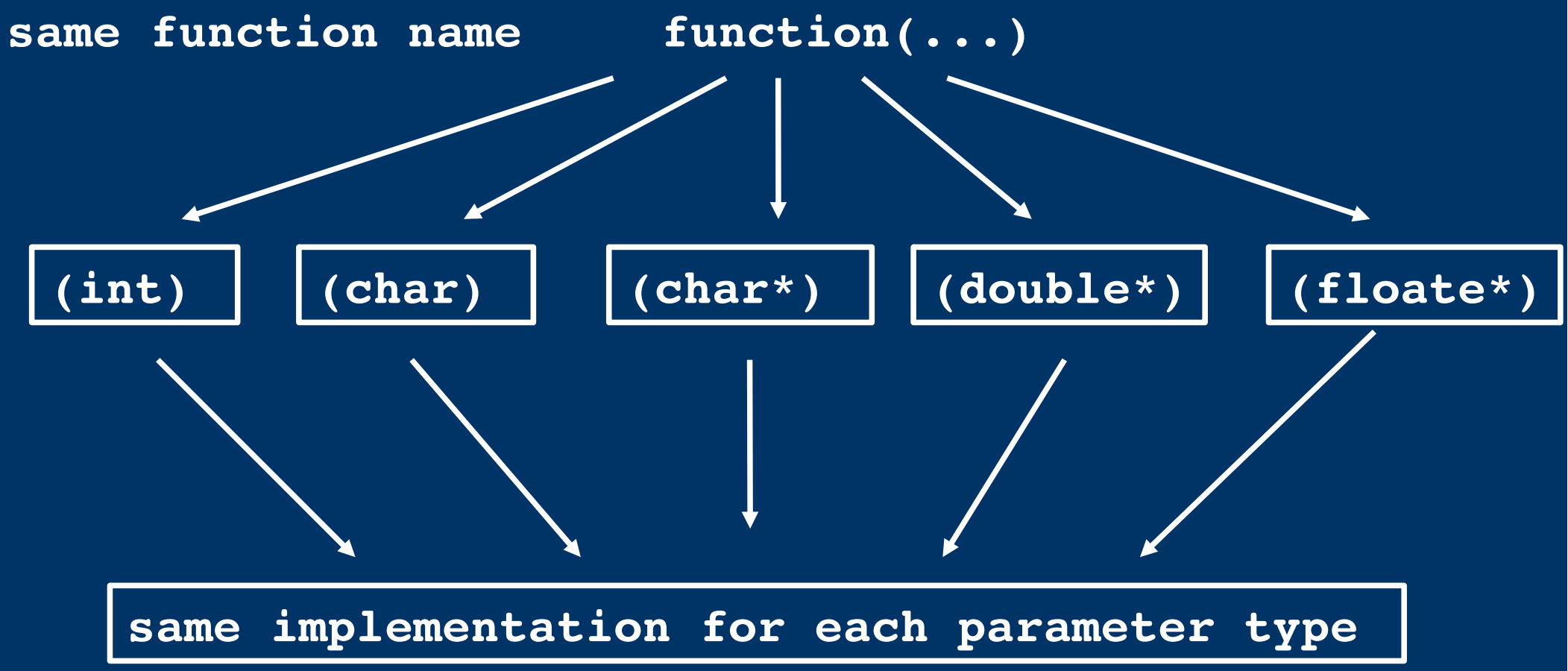
```
float aFunction(int value1);
float aFunction(int value1, int value2);
float aFunction(float value1);
```

- Note that the return type of a function can not be overloaded and must remain the same value for every method.

Parametric Polymorphism (genericity)

- The primary application of parametric polymorphism is in O-O systems to allow methods to work in a generic way.
- However these functions are not truly generic
  - and each different parameter list has it’s own implementation
- In contrast a generic method will execute the same implementation but be able to accept a range of types as parameters

Parametric Polymorphism (genericity)



Generic Functions (using templates)

- In C++ genericity is achieved by the use of templates
- A template will operate on any data type for which the internal implementation is appropriate.
- For example a template function which compares two objects using the ’>’ operator to return the greater will operate on any data type (including Objects) where the ’>’ function is appropriate.
- From the operator overloading examples seen in the previous lectures, we can see that we can now also use a simple template function with objects.

But why use templates

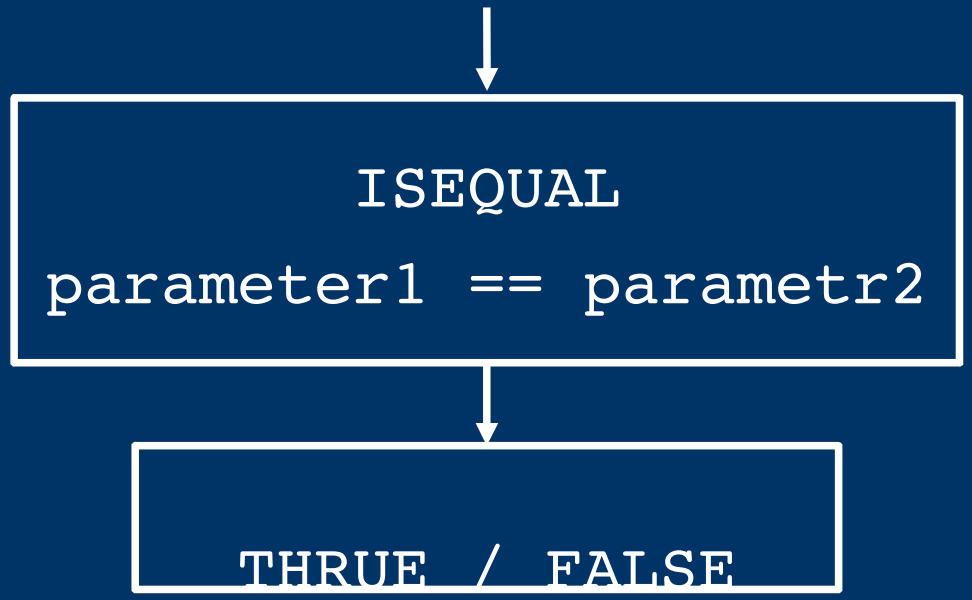
- We often need to perform a similar process on different data types
- The data type(s) being processed are passed to a method as parameters
- With parametric overloading, each type of parameter will cause the compiler to use a different (type specific) method.
- With genericity, a single generic function is able to process all data types, including those defined by the programmer (i.e. objects) - all data types are handled by one (type generic) model.
- Genericity allows us to create generic classes, as well as simply using generic functions as methods

Generic Functions

- Genericity is a more powerful tool than parametric overloading for O-O systems because it does not have to anticipate the type of data parameters which may be supplied at run time.
- This means it is able to handle dynamic objects of disparate types
- However, this approach only works if we are able to process all data types in a particular way, this means that we need to have a common interface for all objects used as parameters

Generic function parameters

ISEQUAL(parameter1,parameter2)



- With generic functions parameters can be of any type
- However they must be able to be processed by the function, which in this case means that able to use the '==' operator
- This will also be true of any objects which use this function as they will need an overloaded '==' operator.

Template Functions

- Template functions are a generic functions which can have parameters of any data type
- The C++ syntax for creating a template function is as follows

```
template <class T> [return type] function(parameters)
```

- template is a C++ keyword, and the name of the generic class name must be enclosed in pointed brackets <.....>
- The class type name can be anything but most examples use T
- This value T acts as an alias for any data type actually passed to the function (whether that data type is int, char, bank account banana etc)

A simple Template example

```
//tgreater.h
template <class T> bool isGreater(T x, T y)
{
if(x>y)
   return true;
else
   return false;
}
```

- The above code is saved in the header file tgreater.h so it can be included in the .cpp file as follows

```
#include "tgreater.h"
int main(void)
{
int   a=15,  b=6;
float aa=12.3, bb=34.53;
char ca='a', cb='z';
// function used with an int
if (isGreater(a,b))
         cout < <"a is greater than b"< <endl;
else
         cout < <"b is greater than a"< <endl;
//function used with a float
if (isGreater(aa,bb))
         cout < <"aa is greater than bb"< <endl;
else
         cout < <"bb is greater than aa"< <endl;
// function used with a char
if (isGreater(ca,cb))
        cout < <"ca is greater than cb"< <endl;
else
        cout < <"cb is greater than ca"< <endl;
  }
```

Template example

```
//tgreater.h
template <class T,class V> bool isGreater(T x, V y)
{
if(x>y)
   return true;
else
   return false;
}
```

- In the previous example the template function would only work for parameters of the same type, by modifying the type name parameters we can change the class to have different (or the same) data types.
- Now we have two parameter types T and V

Testing 2 parameter template

```
int   b=6;
float aa=12.3;
// function used with an int
if (isGreater(aa,b))
         cout < <"aa is greater than b"< <endl;
else
         cout < <"b is greater than aa"< <endl;
```

- Now we can use our generic function to test if an integer is greater than a floating point number.

What happens when templates are used

- As template functions are "generic" functions we do not know what data types are to be used until compile time
- This means that the compiler has to expand the template function for every data type to be used and create separate type specific functions for each use.
- For the previous example the compiler would generate 3 different functions one for **int** one for **float** and one for **char** data types

What happens when templates are used

- This means that at compile time if templates are used compilation will be slower as the compiler has to expand the templates into real functions

- As there is no checking until compilation time for template we do not know whether they will work until the compiler has expanded them. This can lead to error especially when we are using classes as parameters to templates

Template Class

- Templates can also make class definition more general
- Format:

```
template <typename T>
class ClassName{
   definition
   }
```

- To create an object of the class:

```
ClassName< type > myObject;
```

Example: `Stack< double > doubleStack;`

Template Class Functions

- Declared normally, but preceded by **template<class T>**
- Generic data in class listed as type **T**

```
template<typename T>
MyClass< T >::MyClass(int size)
{
   myArray = new T[size];
}
```

- Constructor definition - creates an array of type **T**

Template Class Functions

- Classes can be overridden
  - For template class **Pair**, define a class named **`Pair<userDefinedType>`**
- This new class overrides then class template for **`userDefinedType`**
- The template remains for unoverriden types

Template advantages / disadvantages

- Defining generic classes that are “parametrised” with types i.e. classes can have types as arguments.
- A template parameter representing a “type variable” is denoted as class identifiers
- Template arguments are type names that match “type variable” parameters
- A powerful way of defining families of similar classes.

But

- Pre-processing of templates expands into a full class specification for every different combination of template arguments. This increases the source code size as well as compilation time.