# Data Structures

# Advances in C++ (1)

Ling-Chieh Kung

Department of Information Management
National Taiwan University

# Outline

- **Pointers**
- Classes
- Inheritance and polymorphism

# Pointers

- A **pointer** is a variable which stores a **memory address**.
  - An **array** variable is a pointer.
- To declare a pointer, use *****.

```
type pointed* pointer name;
```

```
type pointed *pointer name;
```

- Examples:

```
int *ptrInt;
```

```
double* ptrDou;
```

  - These pointers will store addresses.
  - These pointers will store addresses of **int**/**double** variables.
- We may point to **any** type.
- To point to different types, use different types of pointers.

# Pointer assignment

- We use the **address-of operator `&`** to obtain a variable's address:

```
pointer name = &variable name
```

- The address-of operator **`&`** returns the (beginning) **address** of a variable.
- Example:
  - **`ptr`** points to **`a`**, i.e., **`ptr`** stores **the address of `a`**.

```
int a = 5;
int* ptr = &a;
```

- When assigning an address, the two types must **match**.

```
int a = 5;
double* ptr = &a; // error!
```

# Variables in memory

```
• int a = 5;
• double b = 10.5;
• int* aPtr = &a;
• double* bPtr = &b;
• cout << &a; // 0x20c644
• cout << &b; // 0x20c660
• cout << &aPtr; // 0x20c658
• cout << &bPtr; // 0x20c64c
```

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | a | 5 |
| | | |
| 0x20c64c | bPtr | 0x20c660 |
| 0x20c650 | | |
| | | |
| 0x20c658 | aPtr | 0x20c644 |
| 0x20c65c | | |
| 0x20c660 | b | 10.5 |
| 0x20c664 | | |
| | | |

Memory

# Address operators

- There are two address operators.
  - `&`: The **address-of operator**. It returns a variable's address.
  - `*`: The **dereference operator**. It returns the pointed variable (not the value!).
- For `int a = 5`:
  - `a` equals 5.
  - `&a` returns an address (e.g., 0x22ff78).
- For `int* ptrA = &a`:
  - `ptrA` stores an address (e.g., 0x22ff78).
  - `*ptrA` returns `a`, **the variable** pointed by the pointer.
- A pointer pointing to nothing should be assigned `nullptr` or `0`.

# Address operators

- Example:

```
int a = 10;
int* p1 = &a;
cout << "value of a = " << a << endl;
cout << "value of p1 = " << p1 << endl;
cout << "address of a = " << &a << endl;
cout << "address of p1 = " << &p1 << endl;
cout << "value of the variable pointed by p1 = " << *p1 << endl;
```

# Address operators and `nullptr`

- Examples:

```
int a = 10;
int* ptr = nullptr;
ptr = &a;
cout << *ptr; // 10
*ptr = 5;
cout << a;    // 5
a = 18;
cout << *ptr; // 18
```

```
int a = 10;
int* ptr1 = nullptr;
int* ptr2 = nullptr;
ptr1 = ptr2 = &a;
cout << *ptr1; // 10
*ptr2 = 5;
cout << *ptr1; // 5
(*ptr1)++;
cout << a;     // 6
```

# Address operators and `nullptr`

- Dereferencing a null pointer shutdowns the program (a run-time error).

```
int* p2 = nullptr;
cout << "value of p2 = " << p2 << endl;
cout << "address of p2 = " << &p2 << endl;
cout << "the variable pointed by p2 = " << *p2 << endl;
```

# Pointers and arrays

- An array variable **is** a pointer!
  - It records the address of the **first** element of the array.
  - When passing an array, we pass a pointer.
  - The array indexing operator `[]` indicates **offsetting**.
- To further understand this issue, let's study **pointer arithmetic**.
  - Using **+**, **−**, **++**, and **−−** on pointers.

# Indexing and pointer arithmetic

- The array indexing operator **[]** is just an **interface** for doing pointer arithmetic.

```
int x[3] = {1, 2, 3};
int* y = x;
for(int i = 0; i < 3; i++)
  cout << x[i] << " "; // x[i] == *(x + i)
for(int i = 0; i < 3; i++)
  cout << *(y++) << " "; // bad!
```

  - An array variable (e.g., **x**) stores an address, but **++** and **--** work only on pointer variables (e.g., **y**).
- Interface: a (typically safer and easier) way of completing a task.
  - **x[i]** and ***(x + i)** are identical.
  - But using the former is safer and easier.

# References and pointers

- Recall this example:
- When invoking a function and passing parameters, the default scheme is to "**call by value**" (or "pass by value").
  - The function declares its own local variables, using a copy of the arguments' values as initial values.
  - Thus we swapped the two local variables declared in the function, not the original two we want to swap.
- To solve this, we can use "**call by reference**" or "call by pointer."

```
void swap (int x, int y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << endl;
  swap(a, b);
  cout << a << " " << b << endl;
}
void swap (int x, int y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

# Call by reference

- A **reference** is a variable's alias.
- The reference is another variable that refers to the variable.
- Thus, using the reference is the same as using the variable.

```
int c = 10;
int& d = c; // declare d as c's reference
d = 20;
cout << c << endl; // 20
```

- `int& d = c` is to declare `d` as `c`'s reference.
  - This `&` is different from the `&` operator which returns a variable's address.
- `int& d = 10` is an error.
  - A literal cannot have an alias!

# Call by reference

- Now we know how to change a parameter's value:
  - Instead of declaring a usual local variable as a parameter, declare a **reference** variable.
- This is to "call by reference".

```
void swap (int& x, int& y);
int main()
{
  int a = 10, b = 20;
  cout << a << " " << b << endl;
  cout << &a << "\n";
  swap(a, b);
  cout << a << " " << b << endl;
}
void swap (int& x, int& y)
{
  cout << &x << "\n";
  int temp = x;
  x = y;
  y = temp;
}
```

# Call by pointers

- To call by pointers:
  - Declare a **pointer** variable as a parameter.
  - Pass a pointer variable or an address (returned by `&`) at invocation.
- For the `swap()` example:

```
void swap(int* ptrA, int* ptrB)
{
  int temp = *ptrA;
  *ptrA = *ptrB;
  *ptrB = temp;
}
```

- Invocation becomes `swap(&a, &b);`

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | |
| 0x20c648 | | |
| 0x20c64c | | |
| 0x20c650 | | |
| 0x20c654 | | |
| 0x20c658 | | |
| 0x20c65c | | |
| 0x20c660 | a | 20 |
| 0x20c664 | b | 10 |
| | | |

Memory

# Call by pointers

- How about the following implementation?

```
void swap(int* ptrA, int* ptrB)
{
  int* temp = ptrA;
  ptrA = ptrB;
  ptrB = temp;
}
```

  - Invocation: `swap(&a, &b);`
- Will the two arguments be swapped? What really happens?

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | |
| 0x20c648 | | |
| 0x20c64c | | |
| 0x20c650 | | |
| 0x20c654 | | |
| 0x20c658 | | |
| 0x20c65c | | |
| 0x20c660 | a | 10 |
| 0x20c664 | b | 20 |
| | | |

Memory

# Static memory allocation

- In C/C++, we declare an array by specifying it's length as a constant variable or a literal.
  - `int a[100];`
- A memory space will be allocated to an array during the compilation time.
  - 400 bytes will be allocated for the above statement.
- This is called "**static memory allocation**".
- We may decide the length of an array "**dynamically**".
  - That is, during the **run** time.
- To do so, we must use a different syntax.
  - All types of variables may also be declared in this way.

# Dynamic memory allocation

- The operator **`new`** allocates a memory space **and** returns the address.
  - In C, we use a different keyword **`melloc`**.
- `new int;` allocates 4 bytes without recording the address.
- `int* a = new int;` makes `a` store the address of the space.
- `int* a = new int(5);` makes the space contains 5 as the value.
- `int* a = new int[5];` allocates 20 bytes (for 5 integers).
  - `a` points to the first integer.
- Dynamically allocated arrays **cannot be initialized** with a single statement.
  - A loop, for example, is needed.

# Dynamic memory allocation

- All of these spaces are allocated during the **run time**.
- So we may write

```
int len = 0;
cin >> len;
int* a = new int[len];
```

- This allocates a space according to the input from users.

# Dynamic memory allocation

- A space allocated during the run time has **no name**!
  - On the other hand, every space allocated during compilation time has a name.
- To access a dynamically-allocated space, we use a **pointer** to store its address.

```
int len = 0;
cin >> len; // 3
int* a = new int[len];
for (int i = 0; i < len; i++)
  a[i] = i + 1;
```

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | 1 |
| 0x20c648 | N/A | 2 |
| 0x20c64c | | 3 |
| 0x20c650 | | |
| 0x20c654 | | |
| 0x20c658 | len | 3 |
| 0x20c65c | | |
| 0x20c660 | a | 0x20c644 |
| 0x20c664 | | |
| | | |

Memory

# Example: Fibonacci sequence

- Recall the repetitive implementation of generating the Fibonacci sequence.
- After we get the value of sequence length *n*, we dynamically declare an array of length *n*.
- Then just use that array!

```
double fibRepetitive (int n)
{
  if (n == 1)
    return 1;
  else if (n == 2)
    return 1;
  double* fib = new double[n];
  fib[0] = 1;
  fib[1] = 1;
  for (int i = 2; i < n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
  double result = fib[n - 1];
  delete[] fib; // to be explained
  return result;
}
```

# Memory leak

- For spaces allocated during the **compilation** time, the system will **release these spaces** automatically when the corresponding variables no longer exist.

```
void func(int a)
{
  double b;
} // 4 + 8 bytes are released
int main()
{
  func(10);
  return 0;
}
```

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | |
| 0x20c648 | | |
| 0x20c64c | | |
| 0x20c650 | | |
| 0x20c654 | | |
| 0x20c658 | | |
| 0x20c65c | | |
| 0x20c660 | | |
| 0x20c664 | | |
| | | |

Memory

# Memory leak

- For spaces allocated during the **run** time, the system will **NOT** release these spaces unless it is asked to do so.
  - Because the space has no name!

```
void func()
{
  int* bPtr = new int[3];
}
// 8 bytes for bPtr are released
// 12 bytes for integers are not
int main()
{
  func( );
  return 0;
}
```

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | |
| 0x20c648 | N/A | ? |
| 0x20c64c | N/A | ? |
| 0x20c650 | N/A | ? |
| 0x20c654 | | |
| 0x20c658 | | |
| 0x20c65c | | |
| 0x20c660 | | |
| 0x20c664 | | |
| | | |

Memory

# Memory leak

- Programmers must keep a record for all spaced allocated dynamically.

```
double* b = new double;
*b = 5.2;
double c = 10.6;
b = &c; // now no one can access
        // the space containing 5.2
```

- This problem is called **memory leak**.
  - We lose the control of allocated spaces.
  - These spaces are **wasted**.
  - They will not be released unit the program ends

| Address | Identifier | Value |
|---|---|---|
| | | |
| 0x20c644 | | |
| 0x20c648 | b | 0x20c660 |
| 0x20c650 | | |
| 0x20c654 | N/A | 5.2 |
| 0x20c65c | | |
| 0x20c660 | c | 10.6 |
| | | |

Memory

# Releasing spaces manually

- The **delete** operator will release a dynamically-allocated space.

```
int* a = new int;
delete a; // release 4 bytes
int* b = new int[5];
delete b; // release only 4 bytes!
          // Unpredictable results may happen
delete [] b; // release all 20 bytes
```

- The **delete** operator will do nothing to the pointer. To avoid reusing the released space, set the pointer to **nullptr**.

```
int* a = new int;
delete a;  // a is still pointing to the address
a = nullptr;  // now a points to nothing
int* b = new int[5];
delete [] b; // b is still pointing to the address
b = nullptr;    // now b points to nothing
```

# Two-dimensional dynamic arrays

- With static arrays, we may create matrices as two-dimensional arrays.
- An *m* by *n* two-dimensional array has:
  - *m* rows (single-dimensional arrays).
  - Each row has *n* elements.
- With dynamic arrays, we now may create matrices **with different row lengths**.
  - We may still have *m* rows.
  - Now each row may have different number of elements.
  - E.g., a **lower triangular matrix**.

# Example: lower triangular arrays

- `int* array = new int[10];` declares an array of integers.
- `int** array = new int*[10];` declares **an array of integer pointers**!
  - The type of `array[0]` is `int*`.
  - The type of `array[1]` is `int*`.
- Then each of these integer pointers may store the address of a dynamic integer array.
  - And their lengths can be different.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r); // later
  return 0;
}
```

# Example: lower triangular arrays

- Let's visualize the memory events.
- In general, the spaces of the three 1-dim dynamic arrays may be **separated**.
- However, the spaces of the array elements in each array are **contiguous**.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r); // later
  return 0;
}
```

| Address | Identifier | Value |
|---|---|---|
| 0x20c644 | r | 3 |
| 0x20c648 | Array | 0x20c654 |
| 0x20c650 | | |
| 0x20c654 | N/A | 0x20c66c |
| 0x20c65c | N/A | 0x20c670 |
| 0x20c664 | N/A | 0x20c678 |
| 0x20c66c | N/A | 1 |
| 0x20c670 | N/A | 1 |
| 0x20c674 | N/A | 2 |
| 0x20c678 | N/A | 1 |
| 0x20c67c | N/A | 2 |
| 0x20c680 | N/A | 3 |

Memory

# Example: lower triangular arrays

- To pass a two-dimensional dynamic array, just pass that pointer.

```
int main()
{
  int r = 3;
  int** array = new int*[r];
  for(int i = 0; i < r; i++)
  {
    array[i] = new int[i + 1];
    for(int j = 0; j <= i; j++)
      array[i][j] = j + 1;
  }
  print(array, r);
  return 0;
}
```

```
int print(int** arr, int r)
{
  for(int i = 0; i < r; i++)
  {
    for(int j = 0; j < i; j++)
      cout << arr[i][j] << " ";
    cout << "\n";
  }
}
```

# Outline

- Pointers
- **Classes**
- Inheritance and polymorphism

# Class definition

- To define a class:
  - Simply change **struct** to **class**.
  - We may also define the function inside the class definition block.
- Compilation error! Why?

```
int main()
{
  MyVector v;
  v.init(5);
  delete [] v.m;
  return 0;
}
```

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

```
class MyVector
{
  int n;
  int* m;
  void init(int dim);
  void print();
};
void MyVector::init(int dim)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = 0;
}
```

# Visibility

- We can/must set visibility of members in a class:
  - **Public** members can be accessed **anywhere**.
  - **Private** members can be accessed only **in the class**.
  - **Protected** members will be discussed later in this semester.
- These three keywords are the **visibility modifiers**.
- By **default**, all members' visibility level is **private**.
  - That is why **`v.init(5)`** generates a compilation error; **`init()`** is private and cannot be invoked outside the class (e.g., in the main function).
- By setting visibility, we can **hide**/**open** our instance members.
  - Usually all instance variables are private.
  - Let's see how to do this.

# Visibility

- A class with different visibility levels:
- Private instance members can only be accessed **inside** the **definition** of **instance functions**.
  - E.g., `init()` and `print()`.
- Public instance members can be accessed everywhere.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  void init(int dim);
  void print();
};
```

```
int main()
{
  MyVector v;
  v.init(5); // OK!
  delete [] v.m;
  return 0;
}
```

# Why data hiding?

- Setting members to private is to do **data hiding**.
- Why bother?
- By setting members to private, we **control** the way that they are accessed.
  - We can better predict how others may use our class.
- As an example, now we can prevent inconsistency between **n** and the length of **m**!

```
int main()
{
  MyVector v;
  v.init(5); // fine
  v.n = 3; // compilation error!
  delete [] v.m;
  return 0;
}
```

# Why data hiding?

- As another example, we do not want a vector to be printed out in strange formats, such as {0, 10, 20}, [0, 10, 20), (0-10-20), etc.
  - We want they all look the same, like (5, 6, 7).
  - If we allow other programmers to access **`n`** and **`m`**, they can print out a vector in any way they like!
  - So we privatize instance variables and **provide** a public member function **`print()`** to control (restrict) the way of printing a vector.
- These public member functions are often called **interfaces**. All others should communicate with the class through interfaces.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  void init(int dim);
  void print();
};
```

# Encapsulation

- The concepts of **packaging** (grouping member variables and member functions) and **data hiding** together form the concept of "**encapsulation**".
  - Roughly speaking, we pack data (member variables) into a **black box** and provide only **controlled interfaces** (member functions) for others to access these data.
  - Others should not even know how those interfaces are implemented.
- For OOP, there are three main characteristics/functionalities:
  - **Encapsulation**.
  - **Inheritance**.
  - **Polymorphism**.

# Constructors

- A **constructor** is an instance function of a class.
  - However, it is very special.
- A constructor will be invoked **automatically** when the object is **created**.
  - It must be invoked.
  - It cannot be invoked twice.
  - It cannot be invoked by the programmer manually.
- Usually it is used to initialize the object.

# Constructors

- A constructor's name is **the same as** the class.
- It does not return anything, not even `void`.
- You can (and usually will) overload them.
- The constructor with **no parameter** is the **default constructor**.
- If, and only if, a programmer does not define any constructor, the **compiler** makes a default one which **does nothing**.
- A constructor may be private.
  - Be invoked only by other constructors.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim);
  MyVector(int dim, int value);
  void print();
};
```

# Constructors for MyVector

- Let's define our class **MyVector** with constructors:

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int value = 0);
  void print();
};
```

```
MyVector::MyVector()
{
  n = 0;
  m = nullptr;
}
MyVector::MyVector(int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

- Just like usual functions, a constructor may have a default argument.

# Constructors for MyVector

- Now, in the main function, we assign initial values when we declare objects:

```
int main()
{
  MyVector v1(1);
  MyVector v2(3, 8);
  v1.print(); // (0)
  v2.print(); // (8, 8, 8)
  return 0;
}
```

- If any member variable **needs an initial value** when an object is created, you should write a constructor to initialize it.
- Use **constructor overloading** to provide flexibility.

# Destructors

- A destructor is invoked right before an object is **destroyed**.
  - It must be public and have no parameter.
- The compiler provides a default destructor that does nothing.
- To define your own destructor, use **~**:

```
class MyVector
{
  // ...
public:
  // ...
  ~MyVector() { cout << "Bye~\n"; }
};
```

# Why destructors?

- Suppose we do not define our own destructor.
- Then there may be **memory leak** when an object is destroyed.
  - When there is **dynamic memory allocation**.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  // ...
  // no destructor
};
```

```
MyVector::MyVector
  (int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

```
int main()
{
  if (true)
    MyVector v1(1);
    // memory leak
  return 0;
}
```

# Why destructors?

- One typical mission for a destructor is to release those **dynamically allocated memory spaces** pointed by member variables.
  - The default destructor does not do this. We must do this by ourselves.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  // ...
  ~MyVector() {
    delete [] m;
  }
};
```

```
MyVector::MyVector
  (int dim, int value)
{
  n = dim;
  m = new int[n];
  for(int i = 0; i < n; i++)
    m[i] = value;
}
```

```
int main()
{
  if (true)
    MyVector v1(1);
    // no memory leak
  return 0;
}
```

# Object pointers

- A class is a (self-defined) data type.
- A pointer may point to any data type.
  - A pointer may point to an **object**, i.e., store the address of an object.
- Recall the class **MyVector**:

```
int main()
{
  MyVector v(5);
  MyVector* ptrV = &v; // object pointer
  return 0;
}
```

# Object pointers

- What we have done is to use an object to invoke instance functions.
  - E.g., **`a.print()`** where **`a`** is an object and **`print()`** is an instance function.
- If we have a pointer **`ptrA`** pointing to the object **`a`**, we may write **`(*ptrA).print()`** to invoke the instance function **`print()`**.
  - **`*ptrA`** returns the object **`a`**.
- To simplify this, C++ offers the member access operator **`->`**.
  - This is specifically for an object pointer to access its members.
  - **`(*ptrA).print()`** is **equivalent** to **`ptrA->print()`**.
  - **`(*ptrA).x`** is equivalent to **`ptrA->x`**.

# Object pointers

- An example of using an object pointer:
  - `new MyVector(5)` dynamically allocates a memory space.

```
int main()
{
  MyVector v(5);
  MyVector* ptrV = &v;
  v.print();
  ptrV->print();
  return 0;
}
```

```
int main()
{
  // an object pointer
  MyVector* ptrV = new MyVector(5);
  // instance function invocation
  ptrA->print();
  delete ptrV;
  return 0;
}
```

# Why object pointers?

- Object pointers are more useful than pointers for basic data types. Why?
- Passing a pointer into a function is **more efficient** than passing the object.
  - A pointer can be much **smaller** than an object.
  - Copying a pointer is easier than **copying an object**.
- Other reasons will be discussed in other lectures.

# Passing objects into a function

- Consider a function that takes three vectors and returns their sum.

```
MyVector sum
  (MyVector v1, MyVector v2, MyVector v3)
{
  // assume that their dimensions are identical
  int n = v1.getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1.getM(i) + v2.getM(i) + v3.getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

```
int MyVector::getN()
{ return n; }
int MyVector::getM(int i)
{ return m[i]; }
MyVector::MyVector
  (int d, int v[])
{
  n = d;
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
```

  – We need to create **four** **`MyVector`** objects in this function.

# Passing object pointers into a function

- We may **pass pointers** rather than objects into this function:

```
MyVector sum(MyVector* v1, MyVector* v2, MyVector* v3)
{
  // assume that their dimensions are identical
  int n = v1->getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1->getM(i) + v2->getM(i) + v3->getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

  – We need to create **only one** `MyVector` object in this function.
  – Nevertheless, using pointers to access members requires more time.

# Passing object references

- We may also **pass references**:

```
MyVector cenGrav(MyVector& v1, MyVector& v2, MyVector& v3)
{
  // assume that their dimensions are identical
  int n = v1.getN();
  int* sov = new int[n];
  for(int i = 0; i < n; i++)
    sov[i] = v1.getM(i) + v2.getM(i) + v3.getM(i);
  MyVector sumOfVec(n, sov);
  return sumOfVec;
}
```

  – We create **only one** `MyVector` object in this function.

# Constant references

- While we may want to pass references to save time, we need to protect our arguments from being modified.

```
MyVector cenGrav
   (const MyVector& v1, const MyVector& v2, const MyVector& v3)
{
  // ...
}
```

  - Save time while being safe!
- Should we do the same thing when passing object pointers?

# Copying an object

- Consider the following program:

```
class A
{
private:
  int i;
public:
  A() { cout << "A"; }
};
void f(A a1, A a2, A a3)
{
  A a4;
}
```

```
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  f(a1, a2, a3); // A
  A a4 = a1; // nothing!
  return 0;
}
```

- Why just one "**A**" when invoking **f()**? Why no "**A**" when **a4** is created?

# Copying an object

- Creating an object by "copying" an object is a special operation.
  - When we pass an object into a function using the call-by-value mechanism.
  - When we assign an object to another object.
  - When we create an object with another object as the argument of the constructor.

```
f(a1, a2, a3);
```

```
A a4 = a1;
```

```
A a5(a1);
```

- When this happens, the **copy constructor** will be invoked.
  - If the programmer does not define one, the compiler adds a **default copy constructor** (which of course does not print out anything) into the class.
  - The default copy constructor simply **copies all member variables** one by one, regardless of the variable types.

# Copy constructors

- We may implement our own copy constructor.
- In the C++ standard, the parameter must be a **constant reference**.
  - If calling by value, it will invoke itself infinitely many times.

```
class A
{
private:
  int i;
public:
  A() { cout << "A"; }
  A(const A& a) { cout << "a"; }
};
```

```
void f(A a1, A a2, A a3)
{
  A a4;
}
int main()
{
  A a1, a2, a3; // AAA
  cout << "\n===\n";
  f(a1, a2, a3); // aaaA
  return 0;
}
```

# Shallow copy

- If no member variable is an array/pointer, the default copy constructor is fine.
- If there is any array or pointer member variable, the default copy constructor does "**shallow copy**".
  - And two different vectors may share the same space for values.
  - Modifying one vector affects the other!

```
MyVector::MyVector(const MyVector& v)
{ // this is what done by the default
  // copy constructor
  n = v.n;
  m = v.m; // shallow copy
}
```

# Deep copy

- To correctly copy a vector (by creating new values), we need to write our own copy constructor.
- We say that we implement "**deep copy**" by ourselves.
  - In the self-defined copy constructor, we **manually create another dynamic array**, set its elements' values according to the original array, and use **m** to record its address.

```
MyVector::MyVector(const MyVector& v)
{ // this is what should be done
  n = v.n;
  m = new int[n]; // deep copy
  for(int i = 0; i < n; i++)
    m[i] = v.m[i];
}
```

# Static members

- A class contains some instance variables and functions.
  - Each object has its own copy of instance variables and functions.
- A member variable/function may be an attribute/operation **of a class**.
  - When the attribute/operation is **class-specific** rather than object-specific.
  - A class-specific attribute/operation should be identical for all objects.
- These variables/functions are called **static members**.

# Static members: an example

- In MS Windows, each window is an object.
- Each window has some object-specific attributes.
- They also share one class-specific attribute: the color of their title bars.

```
class Window
{
private:
  int width;
  int height;
  int locationX;
  int locationY;
  int status; // 0: min, 1: usual, 2: max
  static int barColor; // 0: gray, ...
  // ...
public:
  static int getBarColor();
  static void setBarColor(int color);
  // ...
};
```

# Static members: an example

- We have to initialize a static variable **globally**.

```
int Window::barColor = 0; // default

int Window::getBarColor()
{
  return barColor;
}

void Window::setBarColor(int color)
{
  barColor = color;
}
```

- To access static members, use ***class name*::*member name***.

```
int main()
{
  Window w; // not used
  cout << Window::getBarColor();
  cout << endl;
  Window::setBarColor(1);
  return 0;
}
```

# Good programming style

- If one attribute should be identical for all objects, it should be declared as a static variable.
  - Do not make it an instance variable and try to maintain consistency.
- Some rules regarding static members:
  - We **may** access a static member inside an instance function.
  - We **cannot** access an instance member inside a static function.
- Though **not suggested**, we **may** access a static member through an object.
  - This will confuse the reader.

# Another way of using static members

- One may use a **static variable** to count the number of **active** (**alive**) objects.

```
class A
{
private:
  static int count;
public:
  A() { A::count++; }
  ~A() { A::count--; }
  static int getCount()
  { return A::count; }
};
```

```
int A::count = 0;

int main()
{
  if(true)
    A a1, a2, a3;
  cout << A::getCount() << endl; // 0
  return 0;
}
```

# Getters and setters

- In most cases, instance variables are private.
- For them to be accessed, sometimes people implement **getters** and **setters** for them.
  - A getter simply returns the value of a private instance variable.
  - A setter simply modifies a private instance variables to a given value.
- What are the benefits and costs for having getters and setters?

```
class MyVector
{
private:
  int n;
  int* m;
public:
  // ...
  int getN() {
    return n;
  }
  void setN(int v) {
    n = v;
  }
};
```

# `friend` for functions and classes

- To "open" private members, another way is to declare "**friends**."
- One class can allow its friends to access its private members.
- Its friends can be **global functions** or other **classes**.
  - Then inside **`test()`** and member functions of **`Test`**, those private members of **`MyVector`** can be accessed.
  - **`MyVector`** cannot access **`Test`**'s members.
- A friend can be declared in either the public or private section. It does not matter.
- A class must declare its friends **by itself**.
  - One cannot declare itself as another one's friend!

```
class MyVector
{
  // ...
friend void test();
friend class Test;
};
```

# friend: an example

```
void test() {
  MyVector v;
  v.n = 100; // syntax error if not a friend
  cout << v.n; // syntax error if not a friend
}
```

```
class Test {
public:
  void test(MyVector v) {
    v.x = 200; // syntax error if not a friend
    cout << v.x; // syntax error if not a friend
  }
};
```

# `friend` for functions and classes

- Declare friends only if data hiding is preserved.
  - Do not set everything public!
  - Use structures rather than classes when nothing should be private.
  - Write appropriate public member functions (e.g., getters and setters).
- **`friend`** may also help you hide data.
  - If a private member should be accessed only by another class/function, we should declare a friend instead of writing a getter/setter.

# this

- When you create an object, it occupies a memory space.
- Inside an instance function, **this** is a **pointer** storing the address of that object.
  - **this** is a C++ keyword.
- When the compiler reads **this**, it looks at the memory space to find the object.
- The two implementations are identical:

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < this->n - 1; i++)
    cout << this->m[i] << ", ";
  cout << this->m[this->n - 1] << ")\n";
}
```

```
void MyVector::print()
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n - 1] << ")\n";
}
```

# `this`

- Suppose that **`x`** is an instance variable.
  - Usually you can use **`x`** directly instead of **`this->x`**.
  - However, if you want to have a **local variable** or **function parameter** having the same name as an instance variable, you need **`this->`**.

```
MyVector::MyVector(int d, int v[])
{
  n = d;
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
```

```
MyVector::MyVector(int n, int m[])
{
  this->n = n;
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
```

- A local variable hides the instance variable with the same name.
  - **`this->x`** is the instance variable and **`x`** is the local variable.

# Constant objects

- Some variables are by nature **constants**.

```
const double PI = 3.1416;
```

- We may also have **constant objects**.

```
const MyVector ORIGIN_3D(3, 0);
```

  – This is the origin in $\mathbf{R}^3$. It should not be modified.
- Should there be any restriction on **instance function invocation**?

# Constant objects

- A constant object cannot invoke a function that modifies its instance variables.
  - In C++, functions that may be invoked by a constant object must be declared as a **constant instance function**.
- For a constant instance function:
  - It can be called by non-constant objects.
  - It cannot modify any instance variable.
- For a non-constant instance function:
  - It cannot be called by constant objects even if no instance variable is modified.

```
class MyVector
{
private:
  int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  ~MyVector();
  int getN() const;
  int getM() const;
  void print();
};
```

# Constant instance variables

- We may have **constant instance variables**.
  - E.g., for a vector, its dimension should be fixed once it is determined.
- Obviously, a constant instance variable should be initialized in the constructor(s).
  - However:

```
MyVector::MyVector()
{
  n = 0; // error!
  m = nullptr;
}
```

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  ~MyVector();
  int getN() const;
  int getM() const;
  void print();
};
```

# Member initializers

- For a constant instance variable:
  - It cannot be assigned a value.
  - It cannot be initialized globally.
- We need a **member initializer**.
  - A specific operation for initializing an instance variable.
  - Can also be used for initializing non-constant instance variables.

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector() : n(0), m(nullptr) {}
  MyVector(int dim, int v[]) : n(dim)
  {
    for(int i = 0; i < n; i++)
      m[i] = v[i];
  }
  // ...
};
```

# Initializing constant instance variables

- Member initializers can also be used when constructors are implemented outside the class definition block.

```
MyVector::MyVector()
  : n(0), m(nullptr)
{
}
MyVector::MyVector(int dim, int v[])
  : n(dim)
{
  for(int i = 0; i < n; i++)
    m[i] = v[i];
}
```

```
class MyVector
{
private:
  const int n;
  int* m;
public:
  MyVector();
  MyVector(int dim, int v[]);
  // ...
};
```

- Member initializers are used a lot in general.

# Outline

- Pointers
- Classes
- **Inheritance and polymorphism**

# Inheritance

- Through inheritance, we may **create new classes from existing classes**.
  - A **derived** (**child**) class inherits a **base** (**parent**) class.
  - A child class has (some) members defined in the parent class.
- Recall that we have defined **MyVector**.
  - A two-dimensional (2D) vector is a vector!
- Let's create a class for 2D vector by inheritance.

```
class MyVector
{
protected: // to be explained
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector()
  void print() const;
};
```

# Child class `MyVector2D`

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
};
MyVector2D::MyVector2D()
{
  this->n = 2;
}
MyVector2D::MyVector2D(double m[]) : MyVector(2, m)
{
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();

  return 0;
}
```

- That is all for **`MyVector2D`**!
  - The modifier **`public`** will be discussed later.

# Inheriting parent class' members

- Members in the parent class are **automatically** defined in the child class.
  - **Except** private members, constructors, and the destructor.
  - A **protected** member can only be accessed by itself and its successors.
- What are the members of **MyVector2D**?

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
};
```

```
class MyVector
{
protected:
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector()
  void print() const;
};
```

# Invoking parent class' constructors

- The parent class' constructor will not be inherited.
- One of them will be invoked **before** the child class' constructor is invoked.
  - Create the parent before creating the child!
- If not specified, the parent's **default** constructor will be invoked.

```
MyVector::MyVector()
  : n(0), m(nullptr)
{
}

MyVector2D::MyVector2D()
{
  this->n = 2;
  // this->m = nullptr is redundant
}
```

```
int main()
{
  MyVector2D v;

  return 0;
}
```

# Invoking parent class' constructors

- To **specify** a parent's constructor to call, use the syntax for member initializer:
  - **Pass appropriate arguments** to control the behavior.

```
MyVector::MyVector(int n, double m[])
{
  this->n = n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
MyVector2D::MyVector2D(double m[]) : MyVector(2, m)
{
  // not MyVector(2, m) here!
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();

  return 0;
}
```

# Invoking copy constructors

- How about the copy constructor?
- If we do not define one for the child, the system provides a **default** one.
- **Before** the child's default copy constructor is invoked, the parent's copy constructor will be **automatically** invoked.

```
MyVector::MyVector(const MyVector& v)
{
  this->n = v.n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no copy constructor
};
```

# Invoking copy constructors

- If we define a copy constructor for the child, we must **specify** the constructor we want to invoke!
  - Otherwise the parent's **default** constructor will be invoked.

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  MyVector2D(const MyVector2D& v) {}
};
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  MyVector2D w(v);
  w.print(); // error

  return 0;
}
```

# Invoking parent's member functions

- Once member variables are set properly, typically all the member functions of the parent can be used with no error.

```
void MyVector::print() const
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();

  return 0;
}
```

# Invoking parent class' destructor

- When an object of the child class is to be destroyed:
  - First the child's destructor is invoked.
  - **Then** the parent's destructor is invoked **automatically**, even if we do not define a destructor for the child.

```
MyVector::~MyVector()
{
  delete [] m;
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no destructor
};
```

# Defining new members for the child

- A child may have **its own members**.
  - The parent has no way to access a child's member.
- Let's define a **setValue()** function without using arrays:
  - Note that this should never be a member of **MyVector**.
- We may also define new member variables and static members.

```
class MyVector2D : public MyVector
{
public:
  MyVector2D() { this-> n = 2; }
  MyVector2D(double m[]) : MyVector(2, m) {}
  void setValue(double i1, double i2);
};
void MyVector2D::setValue(double i1, double i2)
{
  if(this->m == nullptr)
    this->m = new double[2];
  this->m[0] = i1;
  this->m[1] = i2;
}
```

# Function overriding

- We may also redefine existing member inherited from a parent.
  - This typically happens to member functions.
  - We say that we **override** the member function.
- As an example, let's override **`print()`**:

```
class MyVector2D : public MyVector
{
public:
  MyVector2D() { this-> n = 2; }
  MyVector2D(double m[]) : MyVector(2, m) {}
  void setValue(double i1, double i2);
  void print() const;
};
void MyVector2D::print() const
{
  cout << "2D: (";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

# Function overriding

- To override a parent's member function, define a child's member function with exactly the same **function signature**.
  - A child object will invoke the child's implementation.
  - The parent's implementation becomes hidden to a child object.
- Inside the child class, we may invoke a parent's member function by using `::`.

```
void MyVector2D::print() const
{
  cout << "2D: ";
  MyVector::print();
}
```

  - Use it if consistency can be enhanced.

# Overriding a constant function

- What will happen to the following program?

```
int main()
{
  double i[2] = {1, 2};
  const MyVector2D v(i);
  v.print(); // 2D

  MyVector2D u;
  u.setValue(3, 4);
  u.print(); // No 2D

  return 0;
}
```

```
class MyVector
{
  // ...
  void print() const;
};
class MyVector2D : public MyVector
{
  // ...
  void print() { MyVector::print(); }
  void print() const
  {
    cout << "2D: ";
    MyVector::print();
  }
};
```

# Overriding a constant function

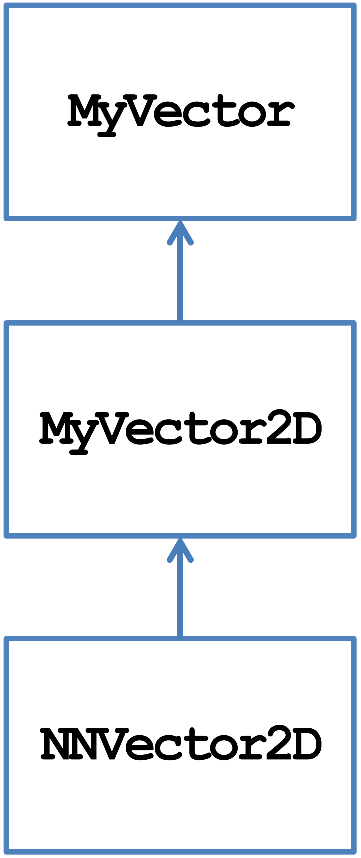
- How about this?

```
int main()
{
  double i[2] = {1, 2};
  const MyVector2D v(i);
  v.print(); // error!

  MyVector2D u;
  u.setValue(3, 4);
  u.print(); // No 2D

  return 0;
}
```

```
class MyVector
{
  // ...
  void print() const;
};
class MyVector2D : public MyVector
{
  // ...
  void print() { MyVector::print(); }
};
```

# Cascade inheritance

- While a child inherits its parent, it may have a grandchild inheriting itself.
- How may we create a class for two-dimensional nonnegative vectors?
  - $\{(x, y) \mid x \geqq 0, y \geqq 0\}$.
- A 2D nonnegative vector **is a** 2D vector!
- Let's use inheritance again.



# Child class NNVector2D

- Defining **NNVector2D** is simple:

```
class NNVector2D : public MyVector2D
{
public:
  NNVector2D(); // do we need it?
  NNVector2D(double m[]);
  void setValue(double i1, double i2);
};
NNVector2D::NNVector2D()
{
}
```

```
NNVector2D::NNVector2D(double m[])
{
  this->m = new double[2];
  this->m[0] = m[0] >= 0 ? m[0] : 0;
  this->m[1] = m[1] >= 0 ? m[1] : 0;
}
void NNVector2D::setValue
  (double i1, double i2)
{
  if(this->m == nullptr)
    this->m = new double[2];
  this->m[0] = i1 >= 0 ? i1 : 0;
  this->m[1] = i2 >= 0 ? i2 : 0;
}
```

  - Why not specifying a parent's constructor?
  - What happens when an **NNVector2D** object is created?

# Cascade inheritance

- In general, a class has all the protected and public members (excluding constructors and destructors) of its predecessors.
- When an object is created:
  - Constructors are invoked from the oldest class to the youngest class.
  - Each constructor can specify a **one-level-above** constructor to invoke.
  - Only one level!
- When an object is destroyed:
  - Destructors are invoked from the youngest to the oldest.

# Inheritance visibility

- Recall that we added the modifier **`public`** when **`MyVector2D`** inherits **`MyVector`** and when **`NNVector2D`** inherits **`MyVector2D`**.
  - This modifier specifies the **inheritance visibility**.
  - It shows how this child modify the member visibility set by its predecessors.
- When one inherits something from its parent, it may **narrow** the **visibility** of these members.
  - E.g., if my parent set its to protected, I may set it to private.
  - E.g., if my parent set its to private, I cannot set it to public.
- Why only narrowing?

# Inheritance visibility

- In general, the visibility of a member in a child class depends on:
  - The member visibility by the parent.
  - The inheritance modifier.

| Member visibility by the parent | Inheritance modifier | | |
|---|---|---|---|
| | **public** | **protected** | **private** |
| **public** | **public** | **protected** | **private** |
| **protected** | **protected** | **protected** | **private** |
| **private** | **private** | **private** | **private** |

- If you have no idea, just use public inheritance.

# Class Character

- There is a public function **beatMonster(int exp)**:
  - It is invoked when the character beats a monster.
  - **exp** is the number of experience points earns in this battle.
  - This function increments the accumulated experience points and checks whether there should be a level up. If so, a private member function **levelUp()** is invoked.
- There is a private function **levelUp()**:
  - The character's **level** will be incremented.
  - However, her abilities will remain the same because characters of different occupations should get different improvements.
  - This should be specified in **Warrior** and **Wizard**.

# Class Character

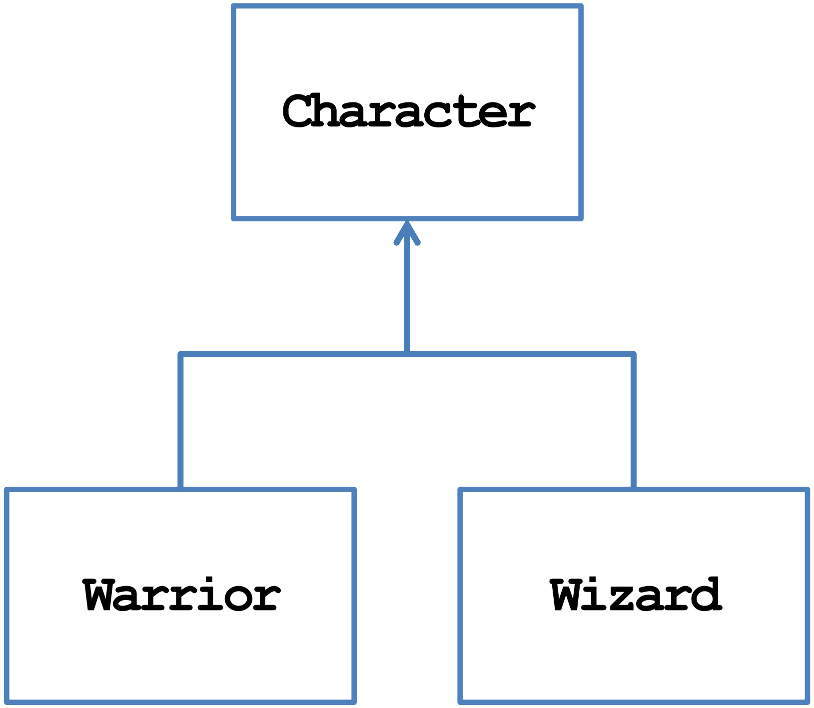
```
class Character
{
protected:
  string name;
  int level;
  int exp;
  int power;
  int knowledge;
  int luck;
  static const int expForLevel = 100;
  void levelUp(int pInc, int kInc, int lInc); // private member function
public:
  Character(string n, int lv, int po, int kn, int lu);
  void beatMonster(int exp);
  void print();
  string getName();
};
```

# Class Character

```
Character::Character(string n, int lv, int po, int kn, int lu)
  : name(n), level(lv), exp(pow(lv - 1, 2) * expForLevel), power(po), knowledge(kn), luck(lu) {}
void Character::beatMonster(int exp) {
  this->exp += exp;
  while(this->exp >= pow(this->level, 2) * expForLevel)
    this->levelUp(0, 0, 0); // No improvement when advancing to the next level
}
void Character::print() {
  cout << this->name
       << ": Level " << this->level << " (" << this->exp << "/" << pow(this->level, 2) * expForLevel
       << "), " << this->power << "-" << this->knowledge << "-" << this->luck << "\n";
}
void Character::levelUp(int pInc, int kInc, int lInc) {
  this->level++; this->power += pInc; this->knowledge += kInc; this->luck += lInc;
}
string Character::getName() {
  return this->name;
}
```

# Character, Warrior, and Wizard

- **Character** should **not** be used to create an object.
  - No improvement when advancing to the next level.
  - Personal attributes for improvements per level are not defined.
- We define two derived classes **Warrior** and **Wizard**:
  - **Character** is an **abstract class**.
  - **Warrior** and **Wizard** are **concrete classes**.



# Classes Warrior and Wizard

```
class Warrior : public Character
{
private:
  static const int powerPerLevel = 10;
  static const int knowledgePerLevel = 5;
  static const int luckPerLevel = 5;
public:
  Warrior(string n) : Character(n, 1, powerPerLevel, knowledgePerLevel, luckPerLevel) {}
  Warrior(string n, int lv)
    : Character(n, lv, lv * powerPerLevel, lv * knowledgePerLevel, lv * luckPerLevel) {}
  void print() { cout << "Warrior "; Character::print(); }
  void beatMonster(int exp) // function overriding
  {
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * expForLevel)
      this->levelUp(powerPerLevel, knowledgePerLevel, luckPerLevel);
  }
};
```

# Classes Warrior and Wizard

```
class Wizard : public Character
{
private:
  static const int powerPerLevel = 4;
  static const int knowledgePerLevel = 9;
  static const int luckPerLevel = 7;
public:
  Wizard(string n) : Character(n, 1, powerPerLevel, knowledgePerLevel, luckPerLevel) {}
  Wizard(string n, int lv)
    : Character(n, lv, lv * powerPerLevel, lv * knowledgePerLevel, lv * luckPerLevel) {}
  void print() { cout << "Wizard "; Character::print(); }
  void beatMonster(int exp) // function overriding
  {
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * expForLevel)
      this->levelUp(powerPerLevel, knowledgePerLevel, luckPerLevel);
  }
};
```

# Some questions

- We may create **`Warrior`** and **`Wizard`** objects in our program.
  - May we **prevent** one from creating a **`Character`** object?
- A "team" has at most ten members.
  - We create two arrays, one for warriors and one for wizards. Each of them has a length of 10.
  - Why **wasting spaces**?

```
class Team
{
private:
  int warriorCount;
  int wizardCount;
  Warrior* warrior[10];
  Wizard* wizard[10];
public:
  Team();
  ~Team();
  // some other functions
};
```

# Some questions

- We may need to add a warrior/wizard, let a warrior/wizard beat a monster, and print the current status of a warrior/wizard.
  - Characters' names are all different.
- Either we write two functions for a task, or write just one.
  - Two: **tedious** and **inconsistent**.
  - One: **Inefficient**.

```
class Team
{
private:
  int warriorCount;
  int wizardCount;
  Warrior* warrior[10];
  Wizard* wizard[10];
public:
  Team();
  ~Team();
  void addWar(string name, int lv);
  void addWiz(string name, int lv);
  void warBeatMonster(string name, int exp);
  void wizBeatMonster(string name, int exp);
  void printWar(string name);
  void printWiz(string name);
};
```

# Polymorphism

- The key flaw is to create two arrays, one for warriors and one for wizards.
  - May we use **only one array** to store the ten members?
  - But **`Warrior`** and **`Wizard`** are different classes.
- While they are different classes, they have **the same base class**.
  - They are all **`Character`**s!
  - May we declare a **`Character`** array to store **`Warrior`** and **`Wizard`** objects?
- We can. This is called **polymorphism**.
  - In C++, the way we implement polymorphism is to

"***Use a variable of a parent type to store a value of a child type***."

# Variables vs. values

- Let's differentiate a **variable's type** and a **value's type**.
- A variable can store values and must have a type.
  - E.g., a **`double`** variable is a **container** which "should" store a **`double`** value.
- A value is the thing that is stored in a variable.
  - E.g., **`12.5`** or **`7`**.
- A value has its own type, which may be **different** from the variable's type.
- In C++, a **parent variable** can store a **child object**.
  - A **`Character`** variable can store a **`Warrior`** or a **`Wizard`** object.
  - Because a warrior/wizard is a character!

# Examples of polymorphism

- For example, we may do this:

```
int main
{
  Warrior w("Alice", 10);
  Character c = w; // copy constructor
  cout << c.getName() << endl; // Alice
  return 0;
}
```

- Or we may do this with pointers:

```
int main
{
  Warrior w("Alice", 10);
  Character* c = &w;
  cout << c->getName() << endl; // Alice
  return 0;
}
```

# Polymorphism with arrays

- Polymorphism is useful typically with **functions** or **arrays**:

```
int main
{
  Character c[3]; // Need a default constructor!
  Warrior w1("Alice", 10);
  Wizard w2("Sophie", 8);
  Warrior w3("Amy", 12);
  c[0] = w1;
  c[1] = w2;
  c[2] = w3;
  for(int i = 0; i < 3; i++)
    c[i].print();
  return 0;
}
```

```
int main
{
  Character* c[3];
  c[0] = new Warrior("Alice", 10);
  c[1] = new Wizard("Sophie", 8);
  c[2] = new Warrior("Amy", 12);
  for(int i = 0; i < 3; i++)
    c[i]->print();
  for(int i = 0; i < 3; i++)
    delete c[i];
  // do not delete [] c;
  return 0;
}
```

# Class Team with Polymorphism

- With polymorphism, we may redefine the class **Team**:

```
class Team
{
private:
  int warriorCount;
  int wizardCount;
  Warrior* warrior[10];
  Wizard* wizard[10];
public:
  Team();
  ~Team();
  void addWarrior(string name, int lv);
  void addWizard(string name, int lv);
  void warriorBeatMonster(string name, int exp);
  void wizardBeatMonster(string name, int exp);
  void printWarrior(string name);
  void printWizard(string name);
};
```

```
class Team
{
private:
  int memberCount;
  Character* member[10];
public:
  Team();
  ~Team();
  void addMember
    (string name, int lv, char occupation);
  void memberBeatMonster(string name, int exp);
  void printMember(string name);
};
```

# Class Team with Polymorphism

- With polymorphism, we may redefine the class **Team**:

```
Team::Team()
{
  this->memberCount = 0;
  for(int i = 0; i < 10; i++)
    member[i] = nullptr;
}
Team::~Team()
{
  for(int i = 0;
      i < this->memberCount;
      i++)
    delete this->member[i];
}
```

```
void Team::addMember
  (string name, int lv, char occupation)
{
  if(this->memberCount < 10)
  {
    if(occupation == 'R')
      this->member[this->memberCount] = new Warrior(name, lv);
    else if(occupation == 'D')
      this->member[this->memberCount] = new Wizard(name, lv);
    this->memberCount++;
  }
}
```

# Class `Team` with Polymorphism

- With polymorphism, we may redefine the class **`Team`**:

```
void Team::memberBeatMonster(string name, int exp)
{
  for(int i = 0; i < this->memberCount; i++)
  {
    if(this->member[i]->getName() == name)
    {
      this->member[i]->beatMonster(exp);
      break;
    }
  }
}
```

```
void Team::printMember(string name)
{
  for(int i = 0; i < this->memberCount; i++)
  {
    if(this->member[i]->getName() == name)
    {
      this->member[i]->print();
      break;
    }
  }
}
```

# Remaining questions

- We still cannot prevent one from creating a **`Character`** object.
- What happens to the following program:
  - No "Warrior " and "Wizard " printed out.
  - No experience point accumulated.
- Why?
  - Because the default setting is to invoke the parent's implementation.
  - To invoke the child's one, we need **virtual functions**.

```
int main()
{
  Character* c[3];
  for(int i = 0; i < 3; i++)
    c[i]->print();
  c[0] = new Warrior("Alice", 10);
  c[1] = new Wizard("Sophie", 8);
  c[2] = new Warrior("Amy", 12);
  c[0]->beatMonster(10000);
  for(int i = 0; i < 3; i++)
    c[i]->print();
  for(int i = 0; i < 3; i++)
    delete c[i];
  return 0;
}
```

# Early binding vs. late binding

- When we do **`A a = b`** or **`A* a = &b`**, we are using polymorphism.
- For **`A a = b`**, the system does **early binding**:
  - **`a`** occupies only four bytes for storing **`i`**.
  - **`a`** does not have a space for storing **`j`**.
  - Its type is determined to be **`A`** at **compilation**.
- For **`A* a = &b`**, the system does **late binding**:
  - **`a`** is just a pointer.
  - It can point to an **`A`** object or a **`B`** object.
  - Its "type" can be determined at the **run time**.

```
class A
{
protected:
  int i;
public:
  void a() { cout << "a\n"; }
  void f() { cout << "af\n"; }
};

class B : public A
{
private:
  int j;
public:
  void b() { cout << "b\n"; }
  void f() { cout << "bf\n"; }
};
```

# Early binding vs. late binding

- But we still see the parent's implementation being invoked. Why?

```
int main()
{
  A a;
  B b;
  A* who = &a;
  who->f(); // af
  who = &b;
  who->f(); // af

  return 0;
}
```

- To ask the system to invoke the child's implementation, we need to declare **virtual functions**.

# Virtual functions

- If we declare a parent's member function to be **virtual**, its invocation priority will be lower than a child's (if we use late binding).
  - To do so, simply add **the modifier `virtual`** into the function header:
  - The child's implementation is invoked!
- No need to do that at the child's side.
  - A parent can declare its function as a virtual function.
  - A child cannot declare a parent's function as virtual (it is of no use).
- In summary, we need:
  - Late binding + virtual functions.

```
class A
{
private:
  int i;
public:
  void a() { cout << "a\n"; }
  virtual void f() { cout << "af\n"; }
};
```

# Virtual functions

- For our **Character** class, simply declare **beatMonster()** and **print()** as virtual.

```
class Character
{
protected:
  // ...
public:
  Character(string n, int lv, int po, int kn, int lu);
  virtual void beatMonster(int exp);
  virtual void print();
  string getName();
};
```

- **Warrior** and **Wizard** override the two functions. Now their implementations get invoked.

```
int main
{
  Character* c[3];
  for(int i = 0; i < 3; i++)
    c[i]->print();
  c[0] = new Warrior("Alice", 10);
  c[1] = new Wizard("Sophie", 8);
  c[2] = new Warrior("Amy", 12);
  c[0]->beatMonstor(10000);
  for(int i = 0; i < 3; i++)
    c[i]->print();
  for(int i = 0; i < 3; i++)
    delete c[i];
  return 0;
}
```

# Abstract classes

- The two virtual functions are different in their natures:
  - **print()** is invoked in the children's implementations.
  - **beatMonster()** should not be invoked by any one.
- We may set **beatMonster()** to be a **pure virtual function**:

```
class Character
{
  // ...
  virtual void beatMonster(int exp) = 0;
};
```

  - Now we do not need to implement it.
  - Moreover, we **cannot** create **Character** objects!

# Summary

- Polymorphism is a technique to make our program clearer, more flexible and more powerful.
  - It is based on **inheritance**.
  - It is tightly related to **function overriding**, **late binding**, and **virtual functions**.
- The key action is to "use a parent pointer to point to a child object".
- To implement late binding, you need to
  - Declare and override virtual functions.
  - Do late binding by using parent pointers to point to child objects.