ACA Summer School 2014
Advanced C++

Pankaj Prateek

ACA, CSE, IIT Kanpur

July 3, 2014

Dynamic Memory allocation

- Until now, the variables we saw must be declared at compile time.
- Array sizes need to be declared at compile time.
- Issues:
  - Difficult to conditionally declare a variable
  - Size of all arrays must be declared in advance.

Dynamic Memory allocation

- Until now, the variables we saw must be declared at compile time.
- Array sizes need to be declared at compile time.
- Issues:
  - Difficult to conditionally declare a variable
  - Size of all arrays must be declared in advance.

Dynamic Memory allocation

- Until now, the variables we saw must be declared at compile time.
- Array sizes need to be declared at compile time.
- Issues:
  - Difficult to conditionally declare a variable
  - Size of all arrays must be declared in advance.

Dynamic Memory allocation

- Until now, the variables we saw must be declared at compile time.
- Array sizes need to be declared at compile time.
- Issues:
  - Difficult to conditionally declare a variable
  - Size of all arrays must be declared in advance.

Dynamic Memory allocation

- Until now, the variables we saw must be declared at compile time.
- Array sizes need to be declared at compile time.
- Issues:
  - Difficult to conditionally declare a variable
  - Size of all arrays must be declared in advance.

Dynamic Memory allocation

- Best we can do is to guess the maximum size and hope that would be enough.
- Poor Solution:
  - Wasted memory in case the variables are not used.
  - Artificial Limitations and buffer overflows

Dynamic Memory allocation

- Best we can do is to guess the maximum size and hope that would be enough.
- Poor Solution:
  - Wasted memory in case the variables are not used.
  - Artificial Limitations and buffer overflows

Dynamic Memory allocation

- Best we can do is to guess the maximum size and hope that would be enough.
- Poor Solution:
  - Wasted memory in case the variables are not used.
  - Artificial Limitations and buffer overflows

Dynamic Memory allocation

- Best we can do is to guess the maximum size and hope that would be enough.
- Poor Solution:
  - Wasted memory in case the variables are not used.
  - Artificial Limitations and buffer overflows

Dynamic Memory allocation

- Dynamic Memory Allocation: Allocate memory of whatever size we need, whenever we need it

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Single Variables

- Scalar(non-array) form of `new` operator is used

```
int *value = new int;
*value = 7;
```

- `new` operator returns the address of the variable allocated, which can be stored in a pointer.
- When done with the dynamically allocated memory, explicitly free it for reuse

```
delete value;
// Reinitialize value pointer to NULL
value = 0;
```

- `delete` does not delete the pointer, it deletes the memory the pointer was pointing to

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new(new[])` and `delete(delete[])`

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new`(`new[]`) and `delete`(`delete[]`)

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new`(`new[]`) and `delete`(`delete[]`)

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new`(`new[]`) and `delete`(`delete[]`)

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new`(`new[]`) and `delete`(`delete[]`)

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Dynamic Memory allocation: Arrays

- Allows to choose their size dynamically while program is running
- Use array form of `new`(`new[]`) and `delete`(`delete[]`)

```
int n = 10;
int *array = new int[n];
delete[] array;
```

- While declaring array, `new[]` is called even though not explicitly written
- While deleting, need to use `delete[]` to tell CPU to delete multiple variables instead of a single one.
- Access is same as even normal arrays are const pointers

Operator Overloading

- Allows the programmer to define how operators should interact with various data type.
- Because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading.

Operator Overloading

- Allows the programmer to define how operators should interact with various data type.
- Because operators in C++ are implemented as functions, operator overloading works very analogously to function overloading.

Operator Overloading

```
class complex {
  double re, im;
public:
  // functions etc.
};
```

- c1+c2 is not valid as the compiler does not know what the operator + should do.
- Almost any operator can be overloading in C++. Exceptions are arithmetic if(?:), sizeof, scope(::), member selector(.) and member pointer selector(.*)
- Allows to use classes in a more intuitive way.

Operator Overloading

```
class complex {
  double re, im;
public:
  // functions etc.
};
```

- c1+c2 is not valid as the compiler does not know what the operator + should do.
- Almost any operator can be overloading in C++. Exceptions are arithmetic if(?:), sizeof, scope(::), member selector(.) and member pointer selector(.*)
- Allows to use classes in a more intuitive way.

Operator Overloading

```
class complex {
  double re, im;
public:
  // functions etc.
};
```

- c1+c2 is not valid as the compiler does not know what the operator + should do.
- Almost any operator can be overloading in C++. Exceptions are arithmetic if(?:), sizeof, scope(::), member selector(.) and member pointer selector(.*)
- Allows to use classes in a more intuitive way.

Operator Overloading

```
class complex {
  double re, im;
public:
  // functions etc.
};
```

- c1+c2 is not valid as the compiler does not know what the operator + should do.
- Almost any operator can be overloading in C++. Exceptions are arithmetic if(?:), sizeof, scope(::), member selector(.) and member pointer selector(.*)
- Allows to use classes in a more intuitive way.

Operator Overloading: Basic Rules

- Atleast one of the operands in any overloaded operator must be a user-defined type.
- Only operators which exist can be overloaded. Cannot create an operator ** to do exponentiation
- All operators keep their current precedence and associativity. Eg: Bitwise XOR(^) can be used to do exponents, but it would have wrong precedence and associavity.

Operator Overloading: Basic Rules

- Atleast one of the operands in any overloaded operator must be a user-defined type.
- Only operators which exist can be overloaded. Cannot create an operator ** to do exponentiation
- All operators keep their current precedence and associativity. Eg: Bitwise XOR(^) can be used to do exponents, but it would have wrong precedence and associavity.

Operator Overloading: Basic Rules

- Atleast one of the operands in any overloaded operator must be a user-defined type.
- Only operators which exist can be overloaded. Cannot create an operator ** to do exponentiation
- All operators keep their current precedence and associativity. Eg: Bitwise XOR(^) can be used to do exponents, but it would have wrong precedence and associavity.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.

Operator Overloading

Operators as functions

- nx + ny is equivalent to operator+(nx, ny)
- Function overloading is used to resolve the function calls to different versions of the function based on parameter type(s).
- When evaluating an expression with operators, C++ looks at the operands around the operator to see what type they are.
  - If all operands are built-in types, C++ calls a built-in routine.
  - If any of the operands are user data types, it looks to see whether the class has an overloaded operator function that it can call.
  - If the compiler finds an overloaded operator whose parameters match the types of the operands, it calls that function.
  - Otherwise, it produces a compiler error.