System Software Crash Couse

Samsung Research Russia
Moscow 2019

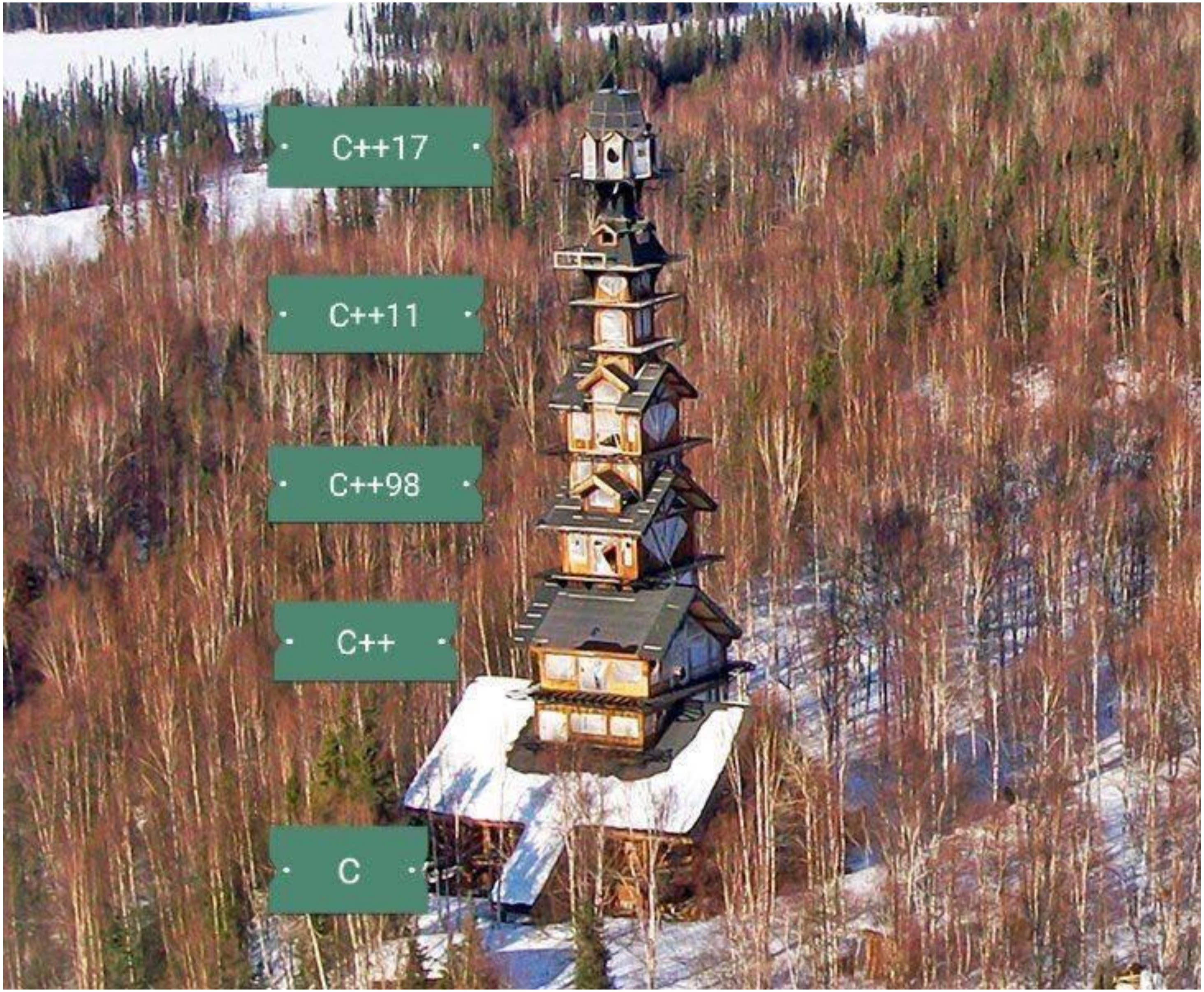
Block G: Advanced C++
1-2. Introduction

Eugene Zouev

Why the Course?

What is this course about?

- C++ is the most popular language for developing large and huge software systems.
- C++ is one of the hardest languages to learn, understand and use (and to implement ☺).
- C++ is evolving quite fast: new language standards come one quick after another – C++11, C++14, C++17 – and all of them carry important new features (and C++20 is coming!)
- The evolution of C++: its new features – gives **new quality** to the language. This is definitely **another language** but not “good old” C++…

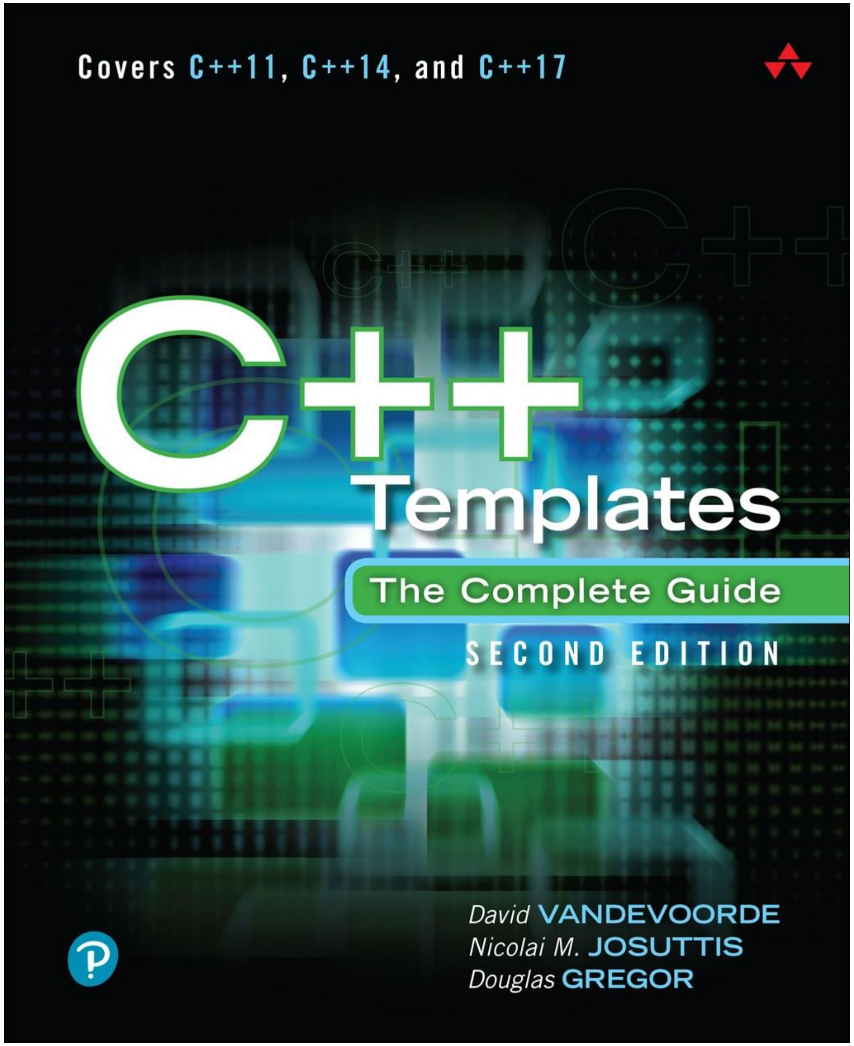

C++17
C++11
C++98
C++
C


C++11
C++17
C
C++
Avito

References 1

- ISO C++ International Standard
  The latest publicly-available "Working Draft" is ok:
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf
- Е.Зуев, А.Чупринов
  Стандарт C++: перевод, комментарии, примеры
- Stroustrup's books
- Internet sources:
  - www.**stackoverflow**.com
  - www.**cppreference**.com
  - https://blog.smartbear.com/c-plus-plus/
  - www.**ibm**.com/developerworks/

References 2



C++ Templates: The Complete Guide, 2nd Edition
By David Vandevoorde,
Nicolai M. Josuttis,
Douglas Gregor
Published Sep 8, 2017
by Addison-Wesley Professional.
ISBN-10: 0-321-71412-1
ISBN-13: 978-0-321-71412-1

Russian translation is available!

References 3

Previous edition:

C++ Templates: The Complete Guide
David Vandevoorde,
Nicolai M. Josuttis
2003

Шаблоны С++. Справочник разработчика
Дэвид Вандевурд, Николаи М. Джосаттис
2003-2016

Some Informal Remarks 1

Do not trust me 100%   ☺☹

Reasons:

- C++ is a very complicated language, and its implementations often treat many language features differently.
- C++ is a very complicated language, and its normative reference ("ISO Standard") contains a number of ambiguities, "white places" and "dark spots".
- C++ is a very complicated language, and I am just a person (not a compiler ☺) therefore I might <u>misunderstand</u> and/or <u>cannot explain</u> some language features (including basic ones ☺)

Conclusion:

Check everything I am saying on your compiler(s)

Some Informal Remarks 2

Pre-requisites & expectations

- I assume **you know** (at least basics of) **OOP** (something like Java/C#/Eiffel). If you don't please help yourself studying it by your own ☺.
- I hope **you know a bit of** (basics of) **C++**. If you don't please take carefully the first two lectures and – again – study any introductory C++ textbook.
- Be noticed that **some important C++ features are out of the course**, e.g., multiple & virtual inheritance, static & dynamic typing etc. So a lot of home studying is really needed from your side…

Two Key Points of the Course

- **Templates**
- New C++ features from the latest Standards (including C++20)

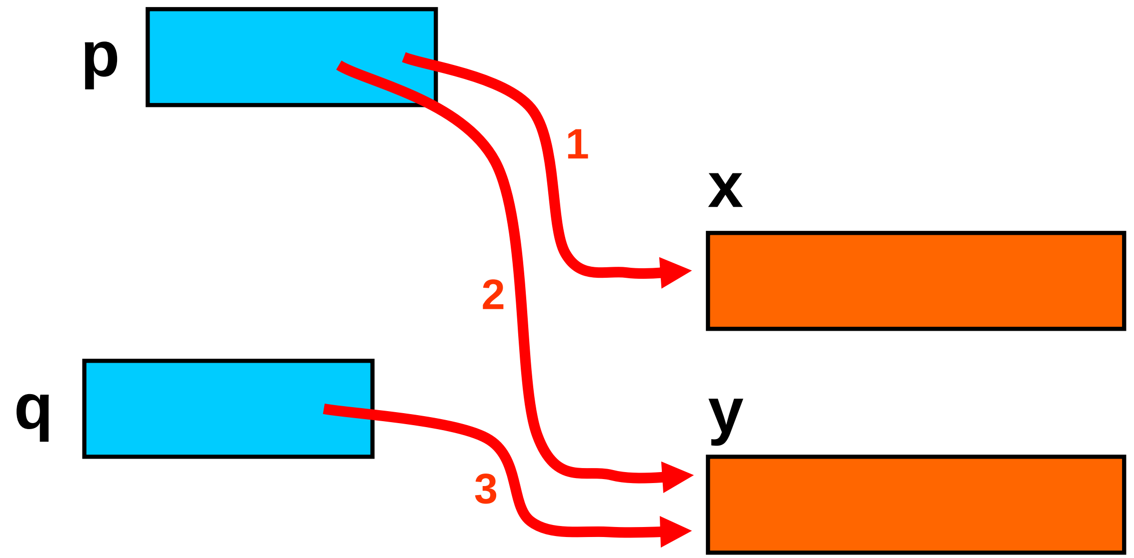
Key C++ Concepts:

Pointers & References

C++ Pointers

1. **Pointer**:
   <u>An object</u> containing an address to some other object

```
int x;
int* p;
...
p = &x;   1
```



Unary "address-of" operator

```
int y;
...
p = &y;  2
```

```
int* q;
...
q = p;   3
```

C++ Pointers

The task for your homework:
Learn the complete syntax of C++ declarations (esp. “declarators”)

2. Pointer types

```
T* p;
```

Declaration of an object of a pointer type, where T denotes a type pointed

Examples:

- **Pointers to (simple) variables;** `int* pv;`
- **Pointers to objects of class types;** `C* pc;`
- **Pointers to arrays;** `int pa[10];`
- **Pointers to functions;** `int (*pf)(int);`
- **Pointers to pointers;** `int** p;`
- **Pointers to values of any type** `void* p;`

C++ Pointers

3. Operators on pointers

```
&object
```

Taking address of object:
Unary prefix operator

```
int x;
int* p;
...
p = &x;
```

```
*pointer
```

Dereferencing: Getting object pointed to by the “pointer”
Unary prefix operator

```
int x;
int* p = &x;
...
*p = 777;      // x is 777
int z = *p+1; // z is 778
```

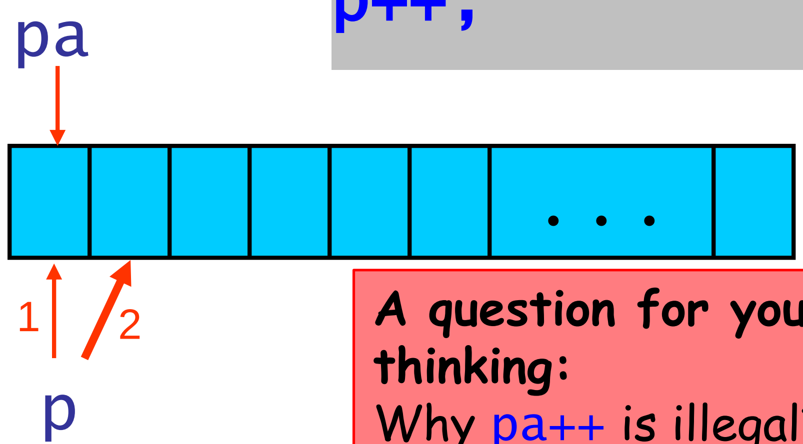
C++ Pointers

4. Operators on pointers: pointer arithmetic

```
pointer+i
pointer-i
pointer++
pointer--
ptr1-ptr2
```

```
int pa[10];
int p = pa;        1

p++;               2
```



A question for your home thinking:
Why `pa++` is illegal?

```
T* p;
p+i // the same as
    // (T*)((char*)p+sizeof(T)*i)
```

C++ Pointers

5. Pointers & “Constness”

```
T* ptr1;
```

Pointer to an object of type `T`; no restrictions on access to the object pointed to by `ptr1`

```
const T* ptr2;
```

Pointer to a **constant object** of type `T`;
cannot use `ptr2` to modify object pointed to by it

```
T*const ptr3 = &v;
```

Constant pointer to an object of type `T`; cannot modify the value of `ptr3` (it must be initialized)

```
const T*const ptr4 = &pc;
```

Constant pointer to a constant object of type `T`; cannot modify the value of `ptr4` (it must be initialized) and cannot use it to modify object pointed to by it

C++ References

1. **Reference:**
   A <u>synonym</u> to some object

```
T& r = o;
```

Declaration of a reference to an object of type T; initializer denotes an object referenced

```
int x;
int& r = x; // r becomes synonym to x
...
r = 7;       // the same as x=7

x = 777;
int v = r;  // v is 777
```

C++ References

2. Reference: Examples

```
void f ( double& a )
{ a += 3.14; }

double d = 7.0;
f(d);  // d has the value of 10.34
```

```
int v[20];

int& f ( int a ) { return v[i]; }

f(3) = 7; // now the 4th element of
          // array v has the value of 7
```

C++ References

3. References: Some rules

References are not objects;
They are synonyms to some objects

No pointers to references
No arrays of references
No references to references
No “constant” references

```
int* p;
int*& rp = p; // OK: reference to a pointer

const int x = 7;
int& ri = x;  // OK: reference to an integer
              // initialized by a reference
              // to the constant integer
```

C++ References

4. Operators on References

No specific operators on references –
just because references **are not objects**

```
int a;
int& r = a;

r = 3;  // a = 3
r++;    // a++
r+=7;   // a += 7
 ...
```

C++ References

5. "Advanced" References

- Usual ("lvalue") references
- "Rvalue" references (together with move semantics) – *see next lecture*

Pointers vs References: A Comparison

	Pointers	References
Syntax	Pointers and references should be explicitly declared	
Status	Pointers are **objects**; they occupy memory	References are **not objects** but synonyms to objects
Value	Pointers' values are **addresses** of objects	References themselves do not have values
Initialization	Pointers can be non-initialized (null pointers)	References should be initialized; they always refer to an object (no "null" references)
Operators	Explicit address-of & dereferencing operators	No special operators on references

Key C++ Concepts:

Classes
Class members
Special members

C++ Classes

1. How to declare a user-defined type?

- By defining a class

(“**Class is a type**” – Std, Chap.9)

Keyword *Class name*

```
class C
{
    // Class members
};
```

Why semicolon?
Is it mandatory?

C++ Classes

2. Which kinds of class members are there in C++?

- (Ordinary or data) members — *Object state*
- Member functions — *Object behavior*
- Special member functions:
  - + Constructors — *The ways objects get created*
  - + Destructor — *The way objects get destroyed*
  - + Operator functions — *The ways objects participate in operations*
  - + Conversion functions — *The ways objects get converted*

C++ Classes

Class members

```
class c
{
    int a;
    ...
    void f ( int p ) { ... }
    ...
    c(int i) { a = i; }
    ~c() { }
    operator < ( int p ) { ... }
    operator int () { return a; }
    ...
};
```

Member functions

Special member functions

Classes & Constructors

3. **Which kinds of constructors are there in C++?**
   - **Default** constructor
   - **Copy** constructor
   - **Move constructor** -- *later*
   - **Conversion** constructor
   - Other constructors

```
class C
{
    int a;
    C()        { a = 0; }
    C(int i)   { a = i; }
    C(C& c)    { a = c.a; }
    C(int i, int j) { a = i+j; }
};
```

Default constructor

Conversion constructor

Copy constructor

Classes & Constructors

4. How to create objects of user-defined types?

- By declaring them (“static” way)
- By creating them (“dynamic” way)

```
class C
{
    ...
};
C c;
C* pc = new C();
```

Class declaration

Object declaration

New operator

Constructors are used for creating objects

Classes & Constructors

5. **When and how various kinds of constructors are used (called)?**

```
class C
{
   ...
   C()          { ... }
   C(int i)     { ... }
   C(C& c)      { ... }
   C(int i, int j);
};
```

```
C c1;
C c2(1);
C c3(c2);
C c4(7,8);
```

Default constructor

Conv. constructor

Copy constructor

Classes & Constructors

6. What's the difference between…

```
class C
{
    ...
    C()           { ... }
    C(int i)      { ... }
    C(C& c)       { ... }
    C(int i, int j);
};
```

```
C c1{1,2};
```

A question for your home work
Consider different kinds of initialization:
`=expr`, `(expr)`, `{expr}`

```
C c1;
C c1 = C();
C c1();
```

- `C c1;` — Default constructor
- `C c1 = C();` — *Default constructor + copy constructor*
- `C c1();` — ***Function declaration!*** ☺

```
C c1(1);
C c1 = 1;
C c1 = C(1);
```

- `C c1(1);` — Conversion constructor
- `C c1 = 1;` — *Conv. constructor + copy constructor*
- `C c1 = C(1);` — Conversion constructor

```
C c1(2);

C c1; c1 = 2;
```

- `C c1(2);` — Conversion constructor
- `C c1; c1 = 2;` — *Conv. constructor + Assignment operator*

Classes & Constructors

```
class C {

public:
    C();          // Default ctor
    C( int i );   // Conversion ctor

    C( C& c );    // Copy ctor
};

int main() {
    C c1;             // Default ctor
    C c2(1);          // Conversion ctor
    C c3 = 1;         // Conversion ctor + Copy ctor
    C c4 = C(1);      // Conversion ctor + Copy ctor
    C c5 = C();       // Default ctor + Copy ctor
    C c6(c3);         // Copy ctor
    C c7 = c4;        // Copy ctor
}
```

A **conceptual scheme**: create a temp object and then use it to create another object (by copying it)

Classes & Constructors

```
#include <iostream>
using namespace std;

class C {
    int a;
public:
    C()         : a(0)   { cout<<"I'm default ctor"<< endl; }
    C( int i ) : a(i)   { cout<<"I'm conv ctor"<< endl; }

    C( C& c )   : a(c.a) { cout<<"I'm copy ctor"<< endl; }
};

int main() {
    C c1;             // Default ctor
    C c2(1);          // Conversion ctor
    C c3 = 1;         // Conversion ctor; no copy ctor!
    C c4 = C(1);      // Conversion ctor; no copy ctor!
    C c5 = C();       // Default ctor; no copy ctor!
    C c6(c3);         // Copy ctor
    C c7 = c4;        // Copy ctor
}
```

Compiler **must create the new object directly** (without using a temp object)

It's allowed for a compiler to create the new object directly (without creating a temp object)

Classes & Constructors

```
#include <iostream>
using namespace std;

class C {
    int a;
public:
    C()          : a(0)   { cout<<"I'm default ctor"<< endl; }
    C( int i )   : a(i)   { cout<<"I'm conv ctor"<< endl; }
private:
    C( C& c )    : a(c.a) { cout<<"I'm copy ctor"<< endl; }
};

int main() {
    C c1;               // Default ctor
    C c2(1);            // Conversion ctor
    C c3 = 1;           // Conversion ctor; no copy ctor!
    C c4 = C(1);        // Error: copy ctor is private!
    C c5 = C();         // Error: copy ctor is private!
    C c6(c3);           // Error: copy ctor is private!
    C c7 = c4;          // Error: copy ctor is private!
}
```

A question for your home work
Check all these examples on your compiler.

Compiler treats this as error **even if the copy ctor is not really used!**

Special Functions by Default

7. The important C++ rule:

- Object of any type should be subject for **creation**, **copying** and **deletion** – by default, even if no means for these operation are provided by programmer.

```
class C
{
    int a;
};
```

```
class C
{
    int a;
    C() { ... }
    C(C& c) { ... }
    C(C&& c) { ... }
   ~C() { ... }
    C& operator=(C& c) { ... }
};
```

To create objects by default

To create objects copying state of other object

*To create objects **moving** state of other object*

To destroy objects

To assign new values to objects

Compiler-generated special functions

Special Functions by Default

Member-wise principle

```
class C : B1, B2, ...
{
   int a;
   C() { ... }
   C(C& c) { ... }

  ~C() { ... }
   C& operator=(C& c) { ... }
};
```

To create objects by default/by copying

- Creating base class subobjects applying their default/copy constructors

- Initializing members applying their default/copy constructors

***To destroy objects**:*

- Apply destructors to base class subobjects

- Apply destructors to members of class types

***To assign new values to objects**:*

- Apply `operator=` *functions to base class subobjects*

- Apply `operator=` *functions to members of class types*

Constructors & Destructors

8. What is destructor for? – to destroy the object

- To release resources the object has acquired
- To release memory the object occupied

```
class C
{
   int a;
 public:
   C() { a = 0; }
};
```

```
void f()
{
   C c;
   C* pc = new C();
   ...
};
...
```

Implicit object creation

Explicit object creation

```
c.C::~C();
```

Compiler-generated destructor call

Object pointed to by `pc` ***still exists***

Constructors & Destructors

9. Explicit destructor calls

```
class C
{
    int a;
 public:
    C() { a = 0; }
};
```

```
void f()
{
    C c;
    C* pc = new C();
    ...
    c.C::~C();
    delete pc;
    ...
};
...
```

Not recommended

```
pc->C::~C();
```

Not recommended

Recommended

Constructors & Conversions

10. Conversion constructors & conversion functions

```
C c(1);
C c = C(1);

bool b = c;
if ( c ) { ... }
```

Conversion constructor:
to convert some type
to a user-defined type

Conversion function:
to convert a user-defined
type to some other type

```
class C
{
    int a;
    C(int i) { a = i; }
    operator bool() { return a==0; }
};
```

Constructors & Conversions

11. Conversion constructors & conversion functions: ambiguity

```
class B;

class A {
   A ( B& b ) { ... }
};

class B {
   operator A() { ... }
};
```

Conversion constructor:
to convert type B *to type* A

Conversion function:
to convert type B *(itself)*
to type A

```
B b;
A a = b;
```

What to apply for conversion?-
either `A a = A(b);`
or `A a = b.operator A();`

Classes & Fundamental Types

12. The great idea of C++: to make user-defined types (classes) very similar to fundamental types

1) Initialization

```
int i(1);
```

```
C c(1);
C c1(c);
```

Conversion constructors

```
class C
{
    int m;
 public:
    C(int i) { m = i; }
    C(C& c) { m = c.m; }
};
```

Classes & Fundamental Types

12. The great idea of C++: to make user-defined types (classes) very similar to fundamental types

2) Assigning new values

```
i = 7;
```

Predefined assignment operator for integer type

```
c1 = c2;
c = 7;
```

Assignment operator

```
class C
{
   int m;
 public:
   C(int i) { m = i; }
   C(C& c) { m = c.m; }
   C& operator=(C& c) { m = c.m; }
};
```

12. The great idea of C++: to make user-defined types (classes) very similar to fundamental types

3) Expressions

```
i = k+m;
```

```
c = c1+c2;
```

Predefined **+** *operator for integer type*

User-defined plus operator

```
class C
{
    int m;
  public:
    C(int i) { m = i; }
    C(C& c) { m = c.m; }
    C& operator=(C& c) { m = c.m; }
    C& operator+(C& c) { return m+c.m; }
};
```

Classes & Fundamental Types

12. The great idea of C++: to make user-defined types (classes) very similar to fundamental types

4) Conversions

```
if ( i ) ...
```

Standard conversion
`int` -> `bool`

```
if ( c ) ...
```

User-defined conversion function

```
class C {
    int m;
  public:
    C(int i) { m = i; }
    C(C& c) { m = c.m; }
    C& operator=(C& c) { m = c.m; }
    C& operator+(C& c) { return m+c.m; }
    operator bool() { return m != 0; }
};
```

Classes & Fundamental Types

This class behaves (almost) exactly as other fundamental types; it can be used everywhere together with other types.

What else:

- Relational operators;
- Conversions to other types;
- Similar support for ***unknown*** types (!?)

```
class C
{
   int m;
 public:
   // Constructors
   C(int i) { m = i; }
   C(C& c) { m = c.m; }
   // Assignment operator
   C& operator=(C& c) { m = c.m; }
   // Arithmetic operators
   C& operator+(C& c) { return m+c.m; }
   ...
   // Conversion functions
   operator bool() { return m != 0; }
   ...
};
```

Base & Derived Classes

13. Base & derived classes

```
class Base
{

 public:
   Base() { ... }
   int member;
   void f ( ) { ... }
};

class Derived : Base
{
 public:
   Derived() { ... }
   int member;
   int memberD;
   void f ( ) { ... }
   void fD( ) { ... }
};
```

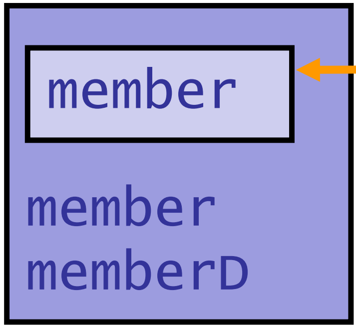
```
Derived d;
```

In class `Derived` *:*

`Derived::member` *hides* `Base::member`

`Derived::f` *hides* `Base::f`

`d:` *Instance of class* `Derived`



`Base` *subobject in* `d`

Instance of class `Derived`

Initializing Members & Bases

14. Object initialization

```
class Base
{
  public:
   Base() { ... }
   Base(int) { ... }
   int member;
};

class Derived : Base
{
  public:
   Derived() { ... }
   int memberD;
};
```

```
Base b;
Derived d;
```

b*'s initialization?*

- *Invoke* Base() *constructor.*

d*'s initialization?*

- *Initialize* Base *subobject in* d *invoking* Base()*;*
- *Initialize* d *itself invoking* Derived()*.*

But: there are two constructors in B; i.e., two ways to initialize (subobject)* Base*?…

Initializing Members & Bases

14. Object initialization (*cont*)

```
class Base
{
  public:
   Base() { ... }
   Base(int) { ... }
   int member;
};

class Derived : Base
{
  public:
   Derived() : Base(1) {...}
   int memberD;
};
```

Ctor-initializer*:*

Explicitly specifies which base class constructor to invoke for the subobject

```
Base b;
Derived d;
```

Initializing Members & Bases

14. Object initialization (*cont*)

```
class Base
{
  public:
   Base() { ... }
   Base(int) { ... }
   int member;
};

class Derived : Base
{
  public:
   Derived() : Base(1), memberD(2), x(3)
   {
       ...
   }
   int memberD;
   const int x;
};
```

How to initialize class members?

Ctor-initializer*:*
Explicitly initializes class member

Ctor-initializer*:*
The only way to initialize constant member

```
Derived d;
```

Key C++ Concepts:

Class member functions
Temporary objects

Member Functions

1. Member function declarations & definitions

File.h

```
...
void f();
...
```

File.c

```
void f()
{
    ...
};
```

C model

```
class C {
 public:
     void f();
};
```

*Member function **declaration**: only function signature*

```
void C::f()
{
    ...
};
```

*Member function **definition**: full function implementation*

Member Functions

2. Class with the member function definition inside

```
class C {
public:
    void f ( ) { . . . }
};

C c;
c.f();
```

*The member function with full definition in the class body is considered as having default **inline specifier** (C++ Standard)*

*Compiler will try **to substitute** the member's body instead of the call*

this & Constant Member Functions

3. this: member function implicit parameter

```
class C {
public:
    int a;
    void f (int a) { this->a = a; }
};
C c;
c.f(1);
```

By definition, this is of type C*const i.e., **constant pointer to the object for which member function is called**

this denotes the **hidden parameter** which is implicitly passed to the member function; the call could be *informally* represented as f(&c,1)

Member function with const modifier:

```
class C {
public:
    void f ( )const { ...this... }
};
```

By definition, here this is of type const C*const i.e., **constant pointer to the constant object**

Constant Member Functions

4. What's the difference?

```
class C
{
public:
    void f1()      { ... }
    void f2()const { ... }
};

C c1;
c1.f1();  // OK
c1.f2();  // OK

const C c2;
c2.f1();  // error: f1 can modify constant c2
c2.f2();  // OK: f2 cannot modify constant c2
```

`f1(C*const this)`

Hence `f1` is considered as the function (potentially) ***modifying*** the object's state: the type of `this` permits modifications like `this->m = ...;`

`f2(const C*const this)`

Hence `f2` is considered as ***non-modifying*** ("safe") function: the type of `this` doesn't permit modifications.

Constant Member Functions

Summary:
If a member function *doesn't really modify* the state of the object then *it is good practice* to make it **"constant"**.

Such function may be applied to both *non-constant* and *constant* object; therefore it is more "generic"…

Last remark:
It's possible to have two "versions" of the member function - "usual" and "safe":

Two **overloaded** member functions. The overloading resolution is performed by ***the first (hidden)*** parameter.

```
class C
{
public:
    void f1()       { ... }
    void f1()const  { ... }
};
```

```
C c1;
const C c2;
c1.f(); // 1st member function
c2.f(); // 2nd member function
```

Temporary Objects

```
class C
{
 public:
    C()    { . . . }
    C(int) { . . . }
    ...
};

void f(C c) { . . . }

f(C(3));
f(C());
```

```
C()
C(7)
```

Explicit constructor call notation may denote **temporary object**

Typical use – passing as an argument or returning as the result value of the function

An unnamed **temporary object** is created and passed to the function as its actual parameter.