C++ history

Born in the 1980s at AT&T Bell Labs

Originally as a `pre-compiler' for C

Source file extensions: .cc, .C or .cpp

Full C grammar and C-library functions

Additional own features as a superset of C

Fast program execution is a main design goal

Online resources

- Frank B. Brokken: C++ annotations
  http://www.icce.rug.nl/documents/cplusplus/
- C++ language reference
  http://cppreference.com/
  http://www.cplusplus.com/
- GNU make manual
  https://www.gnu.org/software/make/manual/
- Doxygen manual
  http://www.doxygen.nl/manual/

Lab environment

Basic tools

- Text editor: kate or gedit
- C++ compiler GNU C++ (g++)

Advanced tools

- Intelligent compilation: GNU make
- Code documentation: doxygen
- Version control system: git

From C to C++

main-C.c

```
// A simple C program

#include <stdio.h>

int main(int argc, char **argv) {
  int i;
  for (i=0; i<argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }
  return(0);
}
```

```
// A simple C++ program

#include <iostream>
int main(int argc, char **argv) {
  for (int i=0; i<argc; i++) {
    std::cout
      << i
      << ": "
      << argv[i]
      << std::endl;
  }
  return(0); // can be omitted
}
```

main-C++.c

Differences between C and C++

```
#include <stdio.h>


printf(
   "%d: %s\n",
   i,
   argv[i]
);
```

Traditional I/O system with format strings.

```
#include <iostream>


std::cout
   << i
   << ": "
   << argv[i]
   << std::endl;
```

Completely new I/O system with operators.

```
void f(); // in C
```

Empty parameter list:

- parameters not specified here

```
void f(void);
// means no parameters
```

```
void f(); // in C++
```

Empty parameter list:

- no parameters at all, C++ is strongly typed

```
void f(void);
// not used in C++,
// use this instead:

void f();
```

typedef

Keyword **`typedef`** is still used in C++, but not required for union, struct or enum definitions:

```
struct MyStruct {
  int    a;
  double b;
};
```

The tag can be used directly as a type name:

```
MyStruct st;
```

New features in C++

function name overloading

overload.cc

```
#include <cstdio>

void show(int val) {
  printf("Integer: %d\n", val); }

void show(double val) {
  printf("Double: %lf\n", val); }

void show(char const *val) {
  printf("String: %s\n", val); }

int main() {
  show(1); show(2.3); show("Hi"); }
```

default parameters

```
struct komplex {
  double re; double im;
};

komplex newKomplex(double r=0, double i=0) {
  komplex z; z.re = r; z.im = i; return(z);
}

int main() {
  komplex a,b,c;
  a=newKomplex(   ); // [0,0]
  b=newKomplex(1  ); // [1,0]
  c=newKomplex(2,3); // [2,3]
}
```

default-param.cc

null pointer

- 0 can be interpreted as an integer as well.
- `NULL` in C is a macro. Avoid macros in C++!
- `NULL` is defined as 0 in many implementations instead of `((void*)0)`
- C++11 introduced `nullptr` which is always a pointer

```
int *ip    = nullptr; // OK
int  value = nullptr; // error: not a pointer
```

constant expressions

- Such functions are also called named constant expressions with parameters. If they are called with compile-time evaluated arguments then the returned value is considered a const value as well.

constexpr.cc

```
constexpr int fib(int n) {
  return n < 3 ? 1 : fib(n-2)+fib(n-1);
}
```

references

- References can be viewed as aliases to other already existing variables.

```
int i = 1;
int &iref = i;
iref++;
std::cout << i; // 2 will be printed
```

- Parameter passing by reference:

```
void increment(int & n) { n++; }
int main() {
  int i = 1;
  increment(i);
  std::cout << i; // 2 will be printed
}
```

simple-ref.cc

operators as functions

- C++ can overload operators as well, enabling them to act on user defined data types. E.g.

```
struct komplex { double re, im; };

komplex operator + (komplex a, komplex b) {
  komplex sum;
  sum.re = a.re+b.re;
  sum.im = a.im+b.im;
  return sum;
}
```

namespaces

- Namespaces can be used to avoid name collisions. A namespace identifier is an additional tag before a name.

```
namespace school {
  struct complex { double re, im; };
} // end of namespace school
school::complex z;
```

- Aliasing a namespace:

```
namespace sc = school;
sc::fun(); // school::fun();
```

namespace.cc

importing names

- It is possible to import names from a namespace. After importing, the namespace tag can be omitted.
- Importing all names:

```
using namespace school; // import all names
fun1(); fun2();          // both from school
```

- Selective import of names:

```
using school::fun1; // import only fun1
fun1(); school::fun2();
```

templates

- Templates are the foundation of generic programming. A template is a blueprint or formula for creating a generic class or a function.

```
template <typename T> T max(T a, T b) {
  return a >= b ? a : b;
}
```

- The Standard Template Library plays a central role in C++. It provides containers, generic algorithms, iterators, function objects, allocators, adaptors and data structures.

exceptions

- C++ offers exceptions as the preferred way of handling abnormal situations. Exceptions are generated by a throw statement within a try-block. Immediately following the try-block, one or more catch-clauses must be defined.

```
try {
  // do something here
  if (someConditionIsTrue)
    throw string("this is an exception");
  }
  // do something else here
}
catch (string error) { /* handle error */ }
```

Scope of variables in C++

- A variable is local to its enclosing block, and is not accessible outside of this block. However some can survive past the end of the block.
- Avoid using global variables!
- Keep scope of variables as limited as you can!
- Define variables when you start using them!

```
for (int i=0; i<10; i++) {
  std::cout << i << std::endl;
}
```

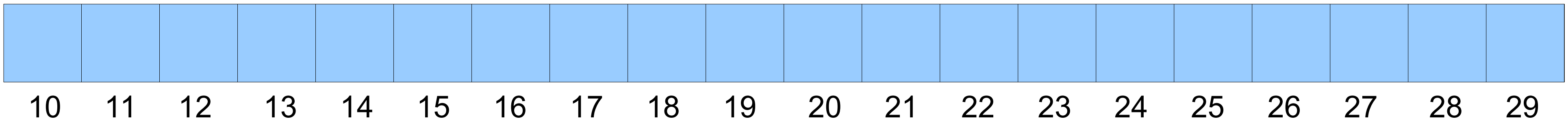
variable-scope.cc

Memory and variables

variables.cc

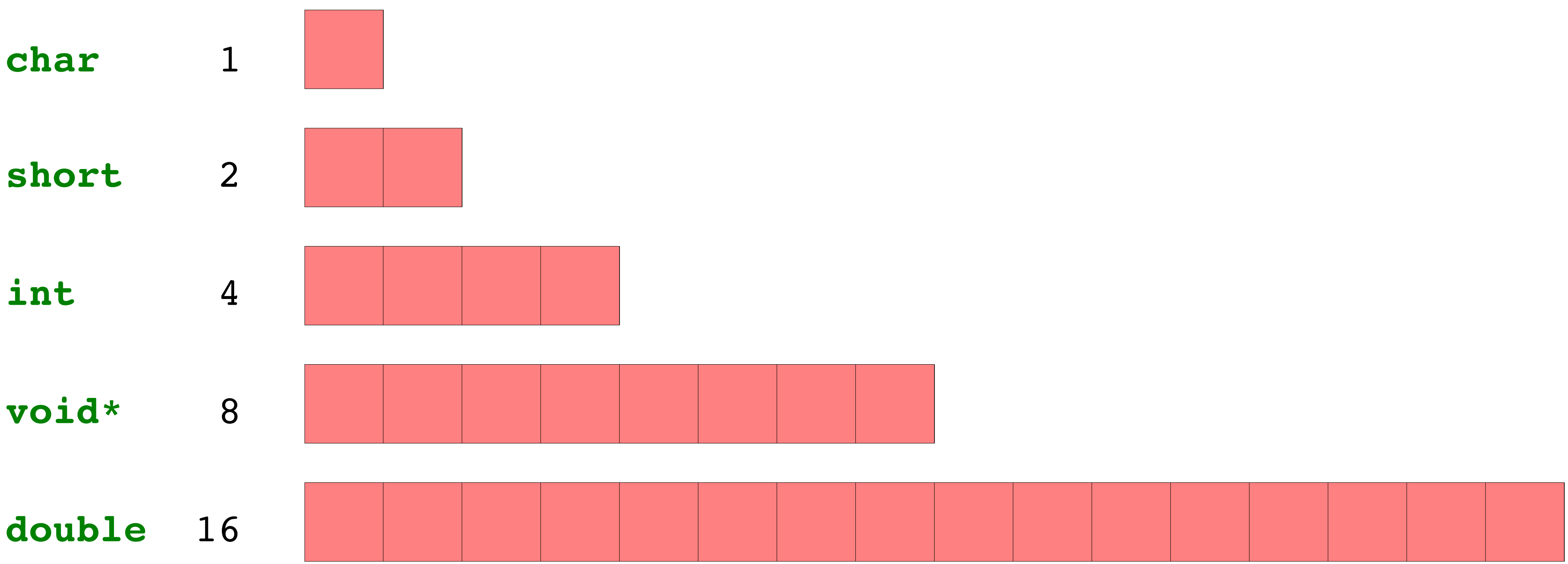
- Memory can be modeled as a long line of uniform boxes. Each box contains 8 bits (one byte) and has a unique serial number (an address). Numbering is continuous.
- We store variables in successive boxes. Each data type requires a certain amount of bytes to store their instances (this can be queried with the `sizeof()` function).
- A pointer tells us the address of the first byte where a variable is stored in memory. Pointers are stored in memory as integer numbers.

System memory is made of bytes (1 byte stores 8 bits). Each byte has a unique address.



Zero is never used to address an existing memory location.
It has a special meaning: a pointer containing zero points to nowhere.
It has its own notation as well: `NULL` in C, `nullptr` in C++.

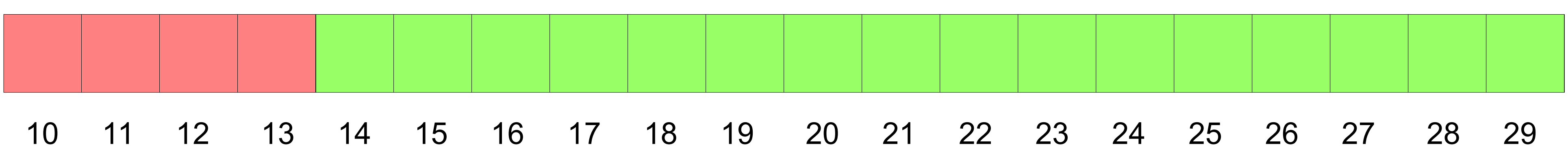
Each data type occupies some amount of bytes in memory.
This can be queried with the `sizeof()` function. An example can be seen below:



- The operating system gives our process a pool of memory to use.
- When a new variable is created, it is given a certain number of consecutive bytes from the free memory pool.
- Variable name is associated with memory address and type information (the latter determines the number of occupied bytes).

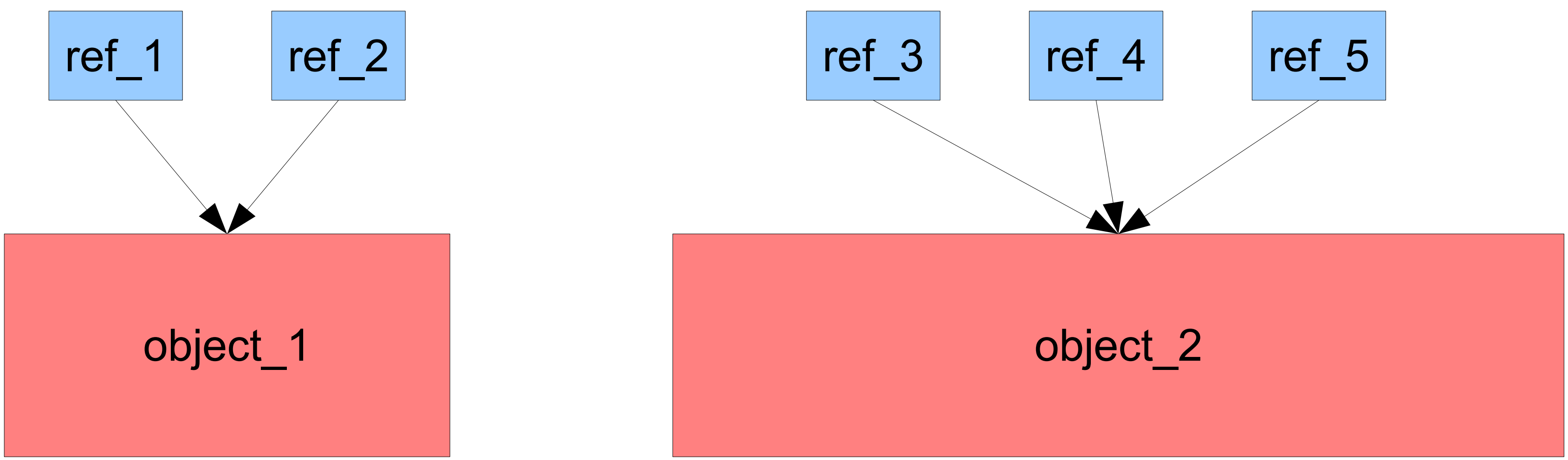
```
int a; // Datasheet:
// name .................... a
// type .................... int
// size .................... 4
// memory location ........ 10
```

Here we defined a 32-bit integer named “a” which was given 4 bytes in memory starting at address 10.



References

- References are somewhere between pointers and conventional variables. In some contexts they behave like a pointer, while in others like a normal variable.
- C++ references always behave like a normal variable with one exception: initialization. This is why another variable of the same type must be there when defining a reference.



Pointers

- Variables which store memory addresses are called pointers.
- Pointers normally carry type information: the type of data that is stored at the memory location pointed to by the pointer.

```
int *p2int; // pointer to an integer
```

- Void pointers: pointers without type information, usually used for advanced purposes.

```
void *p; // pointer to something
```

- Basic pointer operations are **reference** (taking an address of a variable) and **dereference** (looking up the variable at an address).

```
int i;     i   = 3;
int *p2i; p2i = &i;    // reference
int j;     j   = *p2i; // dereference
```

- Pointers can be used to create variables or sequences of variables without names.

```
int *i = new int;      // 1 unnamed integer
int *a = new int [4]; // 4 unnamed integers
```

- The `new` operator will be explained later in this document.

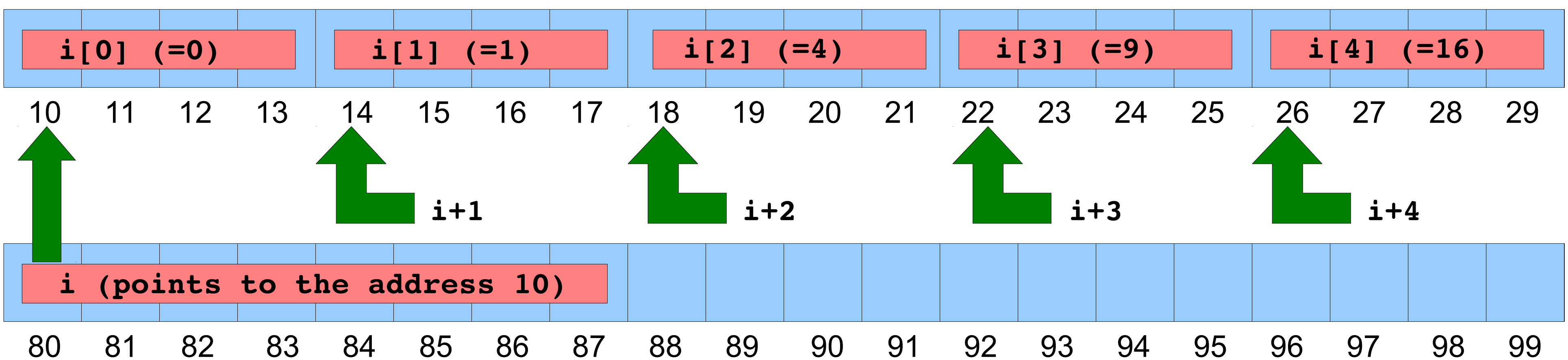
- Dereferencing is possible with the array notation:

```
int *i = new int [5];
i[0]=0; i[1]=1; i[2]=4; i[3]=9; i[4]=16;
```

- Or we can use pointer arithmetic:

```
int *i = new int [5];
*i=0; *(i+1)=1; *(i+2)=4; *(i+3)=9;
*(i+4)=16;
```

- This is how it looks like in system memory:



Summary of pointer operations

pointer-arithmetics.cc

- Declaration:
  `int *p2int;`
- Reference operator (address of a variable):
  `int a = 1; int *p2int = &a;`
- Dereference operator (value stored in memory):
  `int a = 1; int *p2int = &a; int c = *p2int;`
- Dereferencing with array notation:
  `p2int[3] = 1; // same as *(p2int+3) = 1;`
- Arithmetics (addition, subtraction):
  `p2int++; p2int--; p2int += 2; p2int -= 2;`

Reserving and releasing memory

C style	C++ style
`int *ip = (int)malloc(sizeof(int));`	`int *ip = new int;`
`int *ia = (int)malloc( 100*sizeof(int) );`	`int *ia = new int [100];`
`free(ia);`	`delete [] ia;`
`free(ip);`	`delete ip;`

Differences in memory allocation

- `malloc()` is a function which merely reserves bytes in memory.
- `new` and `new[]` are (different) operators which have knowledge about the reserved type.
- `new` is therefore type safe while `malloc()` is not.
- `new` calls constructor, `delete` calls destructor.
- `delete` accepts a null pointer, `free()` does not.
- `malloc()` and `free()` are deprecated in C++, must not be mixed with `new` and `delete`.

Parameter passing

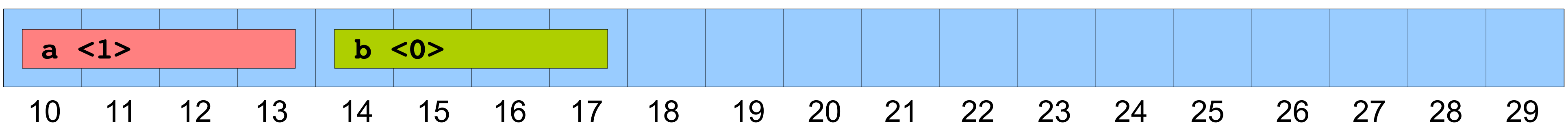
- Passing parameters to functions and returning results from them use the same mechanisms.
- In classic C there is one single mechanism: passing parameters by value. A copy of the original variable is created and this copy is used in the function. The original variable remains intact.
- C++ introduced parameter passing by reference. The function uses the original variable under a different name.

Parameter passing by value

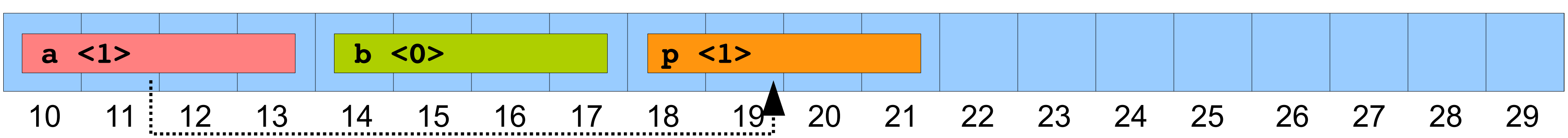
<u>Let's consider the following program lines:</u>

```
int func(int p) { return p*2; }
int main()      { int a=1; int b; b=func(a); return 0; }
```

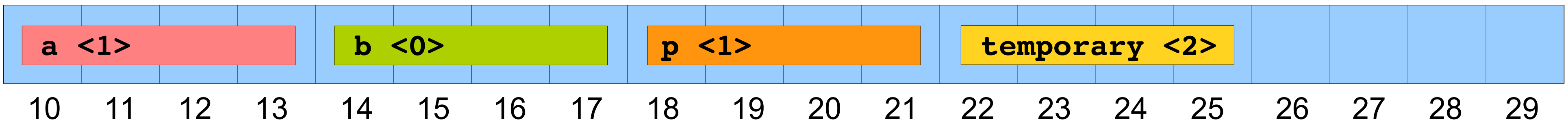
`int a=1; int b;` When **a** and **b** are created in `main()` they are placed in memory.



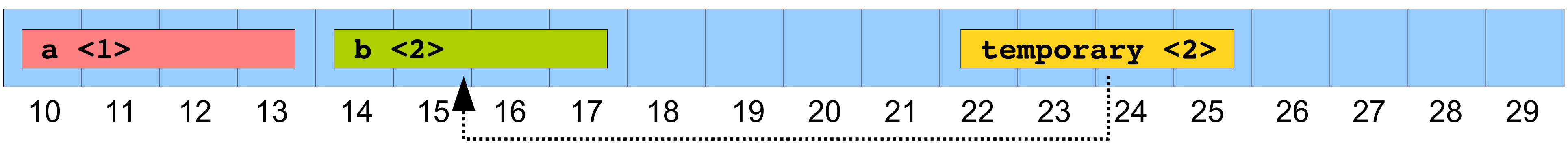
`func(a);` When `func()` is called, parameter **p** is created. Value of **a** is copied into it.



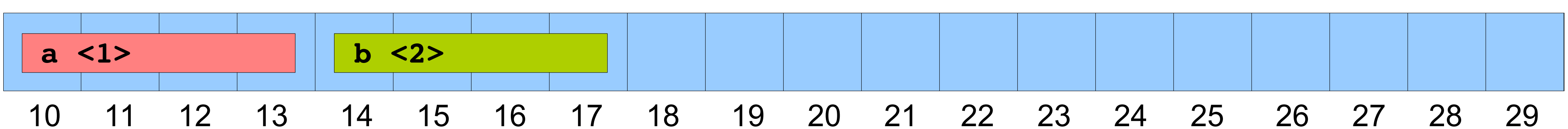
`return p*2;` On return from **`func()`** a temporary variable is created without a name.



When **`func()`** ends, parameter **`p`** is destroyed (it is a local variable). Temporary remains alive until its value is copied into **`b`**, but after that it is destroyed as well.



In the end temporary is also destroyed and only **`a`** and **`b`** remains when execution comes to the next program line (which is **`return 0;`**).



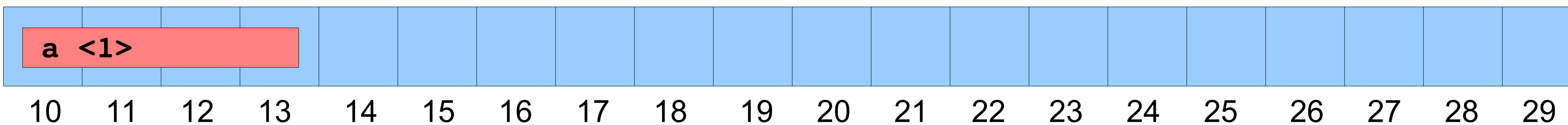
Parameter passing by reference

simple-ref.cc

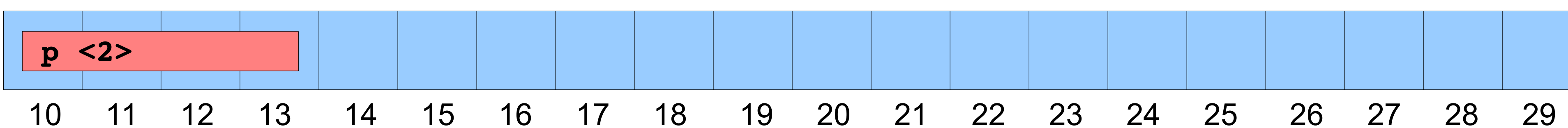
Let's consider the following program lines:

```
void func(int & p) { p*=2; }
int main()         { int a=1; func(a); return 0; }
```

`int a=1;` When **`a`** is created in **`main()`** it is placed in memory.



`func(a);` When **`func()`** is called, no new variable is created for **`p`** because it is a reference. It is only another name for the variable that is called **`a`** in **`main()`**. In **`func()`** it is called **`p`**, but they can be found on the same memory location, so they are the same. After **`p*=2;`** this memory location will contain the value 2.



So in the end **`a`** in **`main()`** will also have the value 2.

Namespaces

- How to prevent naming collisions of independently developed libraries?
- Sticking labels to names makes them unique!
- We can do it with the scope resolution operator.
  `myVariables::a=0;`
- The standard C++ namespace is called `std`.
- Special case: the global scope.
  `::a=0; // designates global scope`
- The `using` clause: use with caution!
  Never put them in a header file!

namespace.cc

Functions inside struct

- Functions do not affect the size of a struct.

```
struct komplex {
  double re; double im;
  void show() {
    std::cout
      << "[" << re << "," << im << "]"
      << std::endl;
    }
};

int main() {
  komplex z; z.re=1; z.im=2; z.show();
}
```

func-in-struct.cc

Constructor and destructor

- These are special member functions.
- Instances of a data structure are created using constructors. At the end of their lives a destructor is called. These calls are automatic.
- Multiple constructors are allowed, but only one destructor.
- Constructors may use default arguments. There are no parameters in a destructor.
- Special syntax: no return value.
- Member initializer syntax for data members.

Automatically created methods

When we do not specify them, these methods are automatically created by the compiler:

- default empty constructor
  `classname();`
- copy constructor
  `classname(const classname &);`
- destructor
  `~classname();`
- assignment operator
  `classname & operator = (const classname &);`

Default methods

According to the C++11 standard we can explicitly request or delete default methods:

- requesting the default copy constructor:
  `classname(const classname &) = default;`
- deleting the default empty constructor:
  `classname() = delete;`

Constructor example

```
struct komplex {
  double re;
  double im;
  komplex(double r=0, double i=0)
  : re(r), im(i)
  {}
};

int main() {
  komplex a;
  komplex b = 1;
  komplex c(2,3);
}
```

constructor.cc

Operators as functions

- C++ can overload operators as well, enabling them to act on user defined data types.
- Adding two `komplex` values is easy this way:

```
komplex a=1; komplex b=2; komplex c=a+b;
```

- Implementation:

```
komplex operator + (komplex a, komplex b) {
  return komplex(a.re+b.re, a.im+b.im);
}
```

- Best practice: define `+=`, `-=`, etc. as class member functions, `+`, `-`, etc. as external functions.

komplex-io-op.cc

Stream insertion operator

- We can easily print our own data types by overloading the C++ stream insertion operator:

```
std::ostream & operator <<
(std::ostream & s, const komplex & z) {
  s<<"komplex["<<z.re<<","<<z.im<<"]";
  return(s);
}
```

- The result of the operator is an ostream, so we can use `<<` infinitely many times in a row.

Special operators

- Index operator: `ValueT operator[] (IntegralT)`
  Accessing elements in a container class. Usually comes in 2 forms returning either an lvalue or an rvalue.
- Function call operator: `ResultT operator() (...)`
  A class having this is called a function object (or functor).
- Type conversion operator: `operator OtherT() const`
- Increment and decrement operators (`++` and `--`):

```
IntegralT & operator++(   ) // prefix, no argument
IntegralT   operator++(int) // postfix, dummy int
```

Reference parameters

- Using references we can pass modifiable arguments to functions:

```
void twoTimes(int & n) {
  n*=2;
}
int main() {
  int a=4; twoTimes(a);
  std::cout << a << std::endl; // 8
}
```

- Using `const` reference parameters we can avoid possibly expensive constructor calls:

```
komplex operator +
(const komplex & a, const komplex & b) {
  return komplex(a.re+b.re, a.im+b.im);
}
```

- A function returning a reference is just the same as passing a parameter to a function, but in the opposite direction.
- Never use local variables as reference return values!

Anatomy of a simple data structure

Now we have everything to assemble our **`komplex`** data structure properly and watch it in action. Components:

- Data fields **`re`** and **`im`**.
- Constructors and destructor with tracing.
- Stream insertion operator for output.
- Basic arithmetical operators.

komplex-struct.cc

Classes

- Technically classes are almost the same as structs, but they provide more complex features.
- New keywords `public`, `private`, `protected` to govern visibility of data and methods.
- The `this` pointer is an implicit parameter of every (non-static) method pointing to the owner instance.

```
class A {
  void f();         // void f(      A *this);
  void f() const;   // void f(const A *this);
};
```

Enhanced komplex class

- To look deeper inside the anatomy of C++ we develop our `komplex` struct into a class.
- Introducing `serial`, a static variable: it exists independently of class instances, similar to globals. Static methods also exist.
- Private `id` uniquely identifies each instance.
- Implementing an assignment operator to avoid copying of `id`.

komplex-class-traced.cc

Komplex class demonstrates

- Different ways of constructor calls.
- Assignment.
- Using arithmetic operators.
- Memory management with `new` and `delete`.
- Parameter passing by value and by reference.

Another simple data structure

LIFO (or stack) – a container class

- LIFO has a container inside. We can put items into the container with the `push()` method.
- We can retrieve the topmost element with `pop()`.
- There are some more convenience methods like `empty()`, `full()`, `top()`, `size()`, `capacity()`.
- We can access data inside the container only with the public interface methods.
- To ensure consistency we must hide inner details (private data members).

main features of LIFO code

- The assignment operator `=`
- Methods designated as `const`. We can create pairs of const and non-const methods:

```
class A {
  void f();       // void f(      A *this);
  void f() const; // void f(const A *this);
};
```

- Memory management with `new` and `delete`
- Friend functions. Classes can also be friends.

lifo-basic.cc

public LIFO interface

- Core functionality

```
void push (const char & c);
char pop  ();
```

- Status check

```
bool empty () const;
bool full  () const;
```

- Convenience methods

```
const char & top      () const;
int          size     () const;
int          capacity () const;
```

private LIFO members

- Data members are private, in order to protect consistency of LIFO state:

```
int    stack_size;
int    stack_capacity;
char * stack_data;
```

- However friends can also access them:

```
friend std::ostream & operator <<
(std::ostream & s, const LIFO & lifo);
```

LIFO enhanced

- Created some **typedef** definitions:

`value_type`	`char`
`pointer`	`char *`
`const_pointer`	`const char *`
`reference`	`char &`
`const_reference`	`const char &`
`iterator`	`char *`
`const_iterator`	`const char *`
`size_type`	`size_t`

- This makes the code more general: we can change the stored type simply by changing these type definitions.

lifo-advanced.cc

Activity: DEQ

- Convert LIFO to a DEQ (double ended queue)!
- Create `push_back()` and `push_front()` instead of `push()`!
- Create `pop_back()` and `pop_front()` instead of `pop()`!
- Add a non-const `operator[]`, add non-const `begin()` and `end()`!
- Create `back()` and `front()` in both const and non-const versions to access elements at both ends of the queue!

deq.cc

Iterators

- In LIFO we introduced iterators for walking through elements one-by-one.
- Iterators can be considered as a generalization of pointers. They play a central role in STL.
- Basic iterator operations:

```
operator == // testing equality
operator != // testing inequality
operator ++ // advancing to next element
operator *  // accessing stored element
```

- Pointer arithmetic may be used for some iterator types (but not all of them).

Iterator concepts

- <u>InputIterator:</u> **`operator++`** for traversing in one direction, **`operator*`** is an rvalue (reading)
- <u>OutputIterator:</u> **`operator++`** for traversing in one direction, **`operator*`** is an lvalue (writing)
- <u>ForwardIterator:</u> traversing in one direction, dereference is read/write capable
- <u>BidirectionalIterator:</u> **`operator++/operator--`** for traversing in both directions, dereference is R/W
- <u>RandomAccessIterator:</u> can use arbitrary pointer arithmetic, dereference is R/W

Templates

- Create more general code: use templates!
- A very simple example: max(a,b)

```
int max(int a, int b) { return(a>b?a:b); }
```

- To create a template we write patterns like this:

```
template<typename T> T max(T a, T b)
{ return(a>b?a:b); }
```

- The compiler can create the actual code from templates using pattern matching:

```
int a=1; int b=2;
std::cout << max(a,b) << std::endl; // 2
```

Template classes

- A template class is a bit more difficult to create, but simple to use.
- Container classes are ideal candidates to be implemented as templates.
- We converted our LIFO class to a template.

lifo-template.cc

Inheritance

- We can extend a class with more methods and variables by using inheritance.
- Suppose we have a base class:
  `class Base { public: int a; void f(); };`
- We can derive class `Derived` from `Base` like this:
  ```
  class Derived : public Base {
    public: int b; void g(); };
  ```
- Now we can access `f()` from a `Derived` instance:
  `Derived derived; derived.f();`
- Each `Derived` instance contains everything defined in `Base`.

Inheritance

the layered model

We can visualize a derived class as multiple layers. Each layer has its own variables and methods. A name in a higher layer obscures the same name in a lower layer.

`Derived d;`	`d.a;`	`d.f();`	`d.b;`	`d.g();`	
`Derived{`		`void f();`	`int b;`	`void g();`	`};`
`Base {`	`int a;`	`void f();`			`};`

Inheritance

functions with base class parameters

- We can call a function which has a formal parameter of class **`Base`** with an actual parameter of class **`Derived`**:

```
void func(Base b) { /*...*/ }
int main() { Derived d; func(d); }
```

- Think of the layered model: when we strip the upper layer from **`d`** we still have the lower layer with everything that **`Base`** contains.

Polymorphism

- In some cases a variable can have a different formal and actual type. This is polymorphism.
- This occurs when we have a reference or a pointer to a base class but we store a derived class behind it.

polymorphism.cc

```
void func(Base & b) { /*...*/ }
Derived d; func(d);
Base *bp = new Derived;
```

- Be very careful with polymorphic pointers! You must use virtual destructors in your classes.

Virtual methods

- With polymorphic variables we have two options when invoking their methods: calling them by the formal type or by the actual type.
- Normal methods are called by the formal type, virtual methods are called by the actual type.
-
```
class A { void n(); virtual void v(); };
class B : public A { void n(); void v(); };
A *x = new B;
x->n(); // call by formal type: A::n()
x->v(); // call by actual type: B::v()
```

Abstract base classes

We can declare a virtual method in a class without a definition. This makes our class abstract. This class cannot be used to create instances. It can only serve as a base class for derived classes.

```
class shape {
  public: virtual double area() const = 0; };

class square : public shape {
  public:
    double a;
    virtual double area() const {
      return(a*a); }
};
```

shapes.cc

Dynamic cast

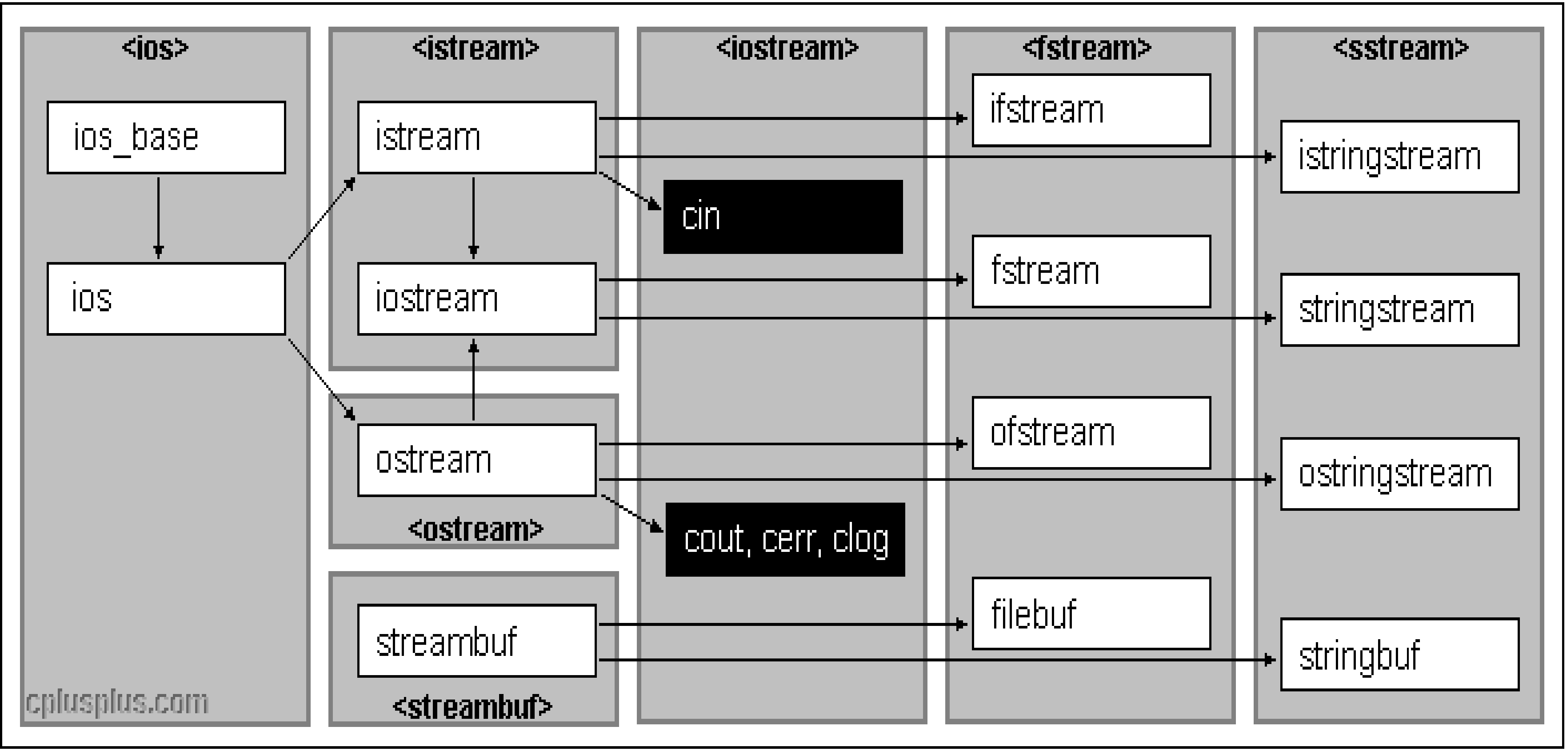
- When using polymorphic variables, we are sometimes forced to use casting to access the right class level.

```
class Base {...};
class Derived : public Base {...};

Base *bp = new Derived;
dynamic_cast<Derived*>(bp)->simpleFunc();

Derived *dp = new Derived;
dynamic_cast<Base*>(dp)->virtFunc();
```

I/O class hierarchy



I/O classes

`ios`, `istream`, `ostream` – do the formatting

`streambuf` – interface to the actual device

`ifstream`, `ofstream` – file I/O

`istringstream`, `ostringstream` – memory I/O

special formatting with I/O manipulators

I/O headers

`<ios>`	types and facilites in the ios class
`<streambuf>`	streambuf or filebuf classes
`<istream>`	classes that do input
`<ostream>`	classes that do output
`<iostream>`	global stream objects (`cin`, `cout`, etc.)
`<fstream>`	file stream classes
`<sstream>`	string stream classes
`<iomanip>`	parameterized manipulators

Stream states

- Stream states are defined in **`ios`**.
- State bits and state query methods:

```
ios::goodbit (=0!)     bool ios::good()
ios::badbit            bool ios::bad ()
ios::failbit           bool ios::fail()
ios::eofbit            bool ios::eof ()
     ios::iostate ios::rdstate()
```

- Streams as bool values: **`!s.fail()`**
- State change:

```
void ios::clear    (                  )
void ios::clear    (ios::iostate state)
void ios::setstate (ios::iostate state)
```

Output formatting

Member function	Manipulator
`ios::fill(char padding)`	`setfill`
`ios::precision(int signif)`	`setprecision`
`ios::width(int nchars)`	`setw`

Format flag	Manipulator
`ios::dec`	`dec`
`ios::hex`	`hex`
`ios::oct`	`oct`
`ios::boolalpha`	`boolalpha`
	`noboolalpha`
`ios::scientific`	`scientific`

They are all defined in the `std` namespace.

I/O manipulators

- Simple manipulators

```
ostream & SM(ostream & s);
cout << SM;
```

- Parametrized manipulators

```
class PM {
  public:
    PM(int n);
    ostream & operator () (ostream & s);
};
ostream& operator<<(ostream& s, PM& pm) {
  return(pm(s));
}
cout << PM(3);
```

komplex-io-op.cc

File I/O

Construction

- `ifstream if(“filename”, ios::in);`
- `ofstream of(“filename”, ios::out);`
- `fstream  f(“filename”, ios::in|ios::out);`

Member functions

- `f.open(“filename”, ios_base::trunc);`
- `if (f.is_open()) {…} else {…}`
- `f.close();`

File open modes

- **`ios::in`** open for input, file must exist
- **`ios::out`** open for output, file recreated if exists
- **`ios::app`** reposition stream to its end before every output command, file is created if doesn't exist, existing content remains the same
- **`ios::ate`** start initially at the end of file, contents are kept only when another flag says so
- **`ios::trunc`** start initially with an empty file, any existing contents lost

String I/O

Construction

- `istringstream is("some string", ios::in);`
- `ostringstream os("some string", ios::out);`
- `stringstream s("string", ios::in|ios::out);`

Member function `str()`

- Set string buffer: `s.str("some other string");`
- Get string buffer: `string str = s.str();`

Configuration file example

- TODO

STL containers

- TODO

STL generic algorithms

- TODO

Multiple source files

- Contents of header files
  - Declarations
  - Inline functions
  - Template code
- Contents of source files
  - Definitions
  - Anything in headers files

Creating makefiles

- TODO

Using GIT

Repository: a database of development history.

`git init` creates a new empty repository

`git add .` adds all files to the repository

`git commit -a -m 'comment'` commit all changes

`git status` status report of changes

`git diff` shows changes not yet staged

`git log` shows development history

`git whatchanged` list changes from the beginning