Introduction to C++
Part 1

Slides based on book:
Thinking in C++
by Bruce Eckel
Second edition
www.bruceeckel.com

FHA, SS 2005

Table of Contents

3 C and C++

Functions
Using the C library
Operators
Pointers
C++ references
Specifying Storage Allocation
make

Functions I 3

- The prototype declares the function. Parameter types are required if any. Parameter names are optional.
- Possible in a *definition* in C++: Unnamed parameters in the argument list. This gives the programmer a way to "reserve space" in the list.
- C++: func() - an empty list
  C: func() - no list checking
  C, C++: func(void) - an empty list

Functions II 3

- Use ellipses (...) to represent a variable argument list in C only: `func(int i, ...)`.
- There are better ways for variable lists in C++ (introduced later).
- In C++ functions a return type must always be specified.

Using the C library 3

- While programming in C++ the C function library is still available.
- Use `#include` to include libraries.
- Local path:
  `#include "lib1"`
  Include directory:
  `#include <lib2>`

Controlling execution 3

- C++ uses all of C's execution control statements.
- In C++ there is a **`bool`**(ean) type, **`true`** and **`false`**.
- **`==`**, **`>`**, **`<`** etc. evaluate to **`true`** or **`false`**.
- **`break`**, **`continue`** and **`switch`** are used the same way as in C.

Operators 3

- Operators in C++ are a special type of functions.
- An operator takes one, two or several arguments (unary, binary operator) to form a new value.
- Operators can be overloaded (discussed later on).

Data types 3

- Built-in data types are intrinsically understood by the compiler.
- C++ knows the built-in type **`bool`** (**`true`** and **`false`**).
- **`bool`**, **`true`** and **`false`** are keywords in C++.
- To guarantee compatibility with older C programs **`int`** is implicitly converted to **`bool`** when necessary.

Pointers 3

```
// pets.cpp
#include <iostream>
using namespace std;

int dog, bird, fish;

void f(int pet) {
  cout << "pet id number: " << pet << endl;
}

int main() {
  int i, j;
  cout << "dog: " << (long)&dog << endl;
  cout << "bird: " << (long)&bird << endl;
  cout << "fish: " << (long)&fish << endl;
  cout << "f(): " << (long)&f << endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
}
```


Example annotations 3

- Prefer **<iostream>** over **<iostream.h>** .
- Then provide the namespace **std**. Necessary if **<iostream>** used instead of **<iostream.h>** .
- **(long)** is a cast saying: treat the variable as if of type **long**.

Pass by value 3

```
// passByValue.cpp
#include <iostream>
using namespace std;

void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
}
```

Output

```
x = 47
a = 47
a = 5
x = 47
```

Pass address 3

```
// passAddress.cpp
#include <iostream>
using namespace std;

void f(int *p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
}
```

Output

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```


C++ references 3

- C++ adds an additional way to pass an address to a function.
- This is called pass-by-reference.
- You designate a variable being a reference by means of the operator & used in the declaration:
  `int& r;`

Pass by reference 3

```
// passByReference.cpp
#include <iostream>
using namespace std;

void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}

int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x);
  cout << "x = " << x << endl;
}
```

Output

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

Example annotations I 3

- To pass a reference via `f()`'s argument list `int& r` is used.
- Inside `f()` `r` is used to get the value in the variable that `r` references (same semantics as with Java).
- Assigning a value to `r` means assigning this value to the the variable that `r` references.

Example annotations II 3

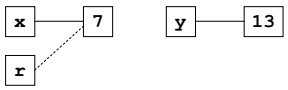
- To obtain the variable's address use the address-of operator `&`:
  ```
  int *p = &r;
  ```
- `f(x)` looks like an ordinary pass-by-value call, *but it isn't* due to the declaration.
  The effect of the reference `x` is that the address is passed into the function rather than a copy of the value of `x`.

References 3

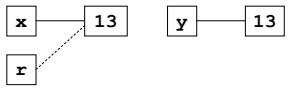
- A variable declared as a reference is just another name (an alias) of the referenced variable.
- Important:
  A reference must be initialized when it is created.
- Once a reference is initialized, it can't be changed to refer to another object.
- There are no null references.

Example 3

```
int x = 7;
int y = 13;
int& r = x;
```




```
r = y;
```



`r` is used as an alias for `x`. There is no difference whether you use `r` or `x`.

Defining variables 3

- C++ (not C) allows to declare variables anywhere in the scope.
- Variables can be initialized at the point where they are declared.
- Variables can also be defined inside the control expressions and inside the conditional of an `if` statement.

```
// ...
for (int i = 0; ... )
```


Storage allocation I 3

- Global variables are unaffected by scope.
- Global variables live throughout the program.

Storage allocation I 3

- The **`extern`** keyword tells the compiler that a variable or function exists even if not yet seen so far.

```
// global.cpp
#include <iostream>
using namespace std;

int globe;
void func();
int main() {
  globe = 12;
  cout << globe << endl;
  func();
  cout << globe << endl;
}
```

```
// global2.cpp

extern int globe;
// The linker resolves
// the reference.

void func() {
  globe = 47;
}
```


Storage allocation II 3

- Local variables occur within scope. Often they are called *automatic* variables.
- Local variables default to the keyword **`auto`**. This keyword is rarly used.
- The keyword **`register`** tells the compiler to make accesses to this variable as fast as possible.

Storage allocation III 3

- **`static`** variables in functions are like global variables but with scope to the function only.
- **`static`** variables outside a function make this variable unavailable to other files. The variable has file scope. The same is true for **`static`** functions.
- **`static`** inside a class has yet another meaning.

Constants I

3

- C++ introduces the concept of a named constant to be used like a variable except that the value can't be changed.
- The type of the **`const`** must be specified: **`const int i = 77;`**
- A C++ constant must be initialized at the place of its declaration (not true in C).

Constants II

3

- **`void printBinary(`**
  `const unsigned char val)`

  `val` is declared as a constant, i.e. **`val`** can't be changed in the function block.

make I

3

- The *make* program is a convenient utility to manage individual files in a project.
- It follows the instructions in a text file named *makefile* to build an executable of the source files.
- *make* only recompiles the source code files that were changed and dependent files.

make II 3

- The *make* utility is available for virtually all C/C++ compilers.
- Some IDEs use *make*, some have similar tools which use a *project file*.

Writing your own *makefile* 3

- The *makefile* lists dependencies between source code. *make* checks the date stamps on the listed files.
- Comments in a *makefile* all start with **#**.

make activities 3

```
# A comment
hello.exe: hello.cpp
←——→mycompiler hello.cpp
<tab>
```

- Dependency: The target *hello.exe* depends on *hello.cpp*
- Rule: execute *mycompiler hello.cpp*

make macros 3

```
CPP = g++
hello.exe: hello.cpp
     $(CPP) hello.cpp
```

- Macros serve as string replacement.
- **=** identifies a macro.
- **$** and the parentheses expand the macro.

Suffix rules 3

```
CPP = g++
.SUFFIXES: .exe .cpp
.exe.cpp:
     $(CPP) $<
```

- With suffix rules *make* is taught how to convert a file of a certain suffix (extension) to a file of another suffix.
- **$<** macro used with suffix rules: insert the dependent (here the *cpp* file).

Invocation 3

```
commandline> make hello.exe
```

- The suffix rule will use *hello* in the *makefile* wherever necessary.

Default targets I

3

```
CPP = g++
.SUFFIXES: .exe .cpp
.exe.cpp:
     $(CPP) $<
target1.exe:
target2.exe:
```

- *make* builds *target1.exe* using the default suffix rule.
- To have *target2.exe* built instead write *make target2.exe*.

Default targets II

3

```
CPP = g++
.SUFFIXES: .exe .cpp
.exe.cpp:
     $(CPP) $<
all: target1.exe target2.exe
```

- Using the non-existent default dummy target *all* that depends on the rest of the targets *make* builds all the dependencies wherever necessary.

Summary

3

- Most fundamental features of C++ are inherited from C.
- C++ adds improvements for better type and function argument checking.
- Some C features are restricted, e.g. constant declaration.
- C++ adds the reference concept to the language.

Table of Contents

4 Data Abstraction

`CStash` - an example
Dynamic storage allocation
The basic object
Abstract data typing
Header files
Nested structures

`CStash` - an example 4

- **`CStash`** is a **`struct`**ure in a C library that comes with a collection of functions that act on this **`struct`**.
- The above situation is typical for libraries and APIs.
- The **`CStash`** example realizes an array whose size can be established during run-time.
- **`CStash`** accepts any type of data which is copied *bytewise* into it.

A C-like library (h-file) 4

```
// cStash.h

typedef struct _CStash {
  int size;            // Size of each space
  int quantity;        // Number of storage spaces
  int next;            // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *storage;
} CStash;


void initialize(CStash *s, int size);
void cleanup(CStash *s);
int add(CStash *s, const void *element);
void *fetch(CStash *s, int index);
int count(CStash *s);
void inflate(CStash *s, int increase);
```


A C-like library I (cpp-file) 4

```
// cStash.cpp
#include "cStash.h"
#include <iostream>
#include <cassert>
using namespace std;

// Quantity of elements to add when increasing
// storage:
const int increment = 100;

void initialize(CStash *s, int size) {
  s->size = size;
  s->quantity = 0;
  s->next = 0;
  s->storage = 0;
}

// cont'd
```


A C-like library II (cpp-file) 4

```
int add(CStash *s, const void *element) {
  // Enough space left?
  if (s->next >= s->quantity) {
    inflate(s, increment);
  }

  // Copy element starting at next empty space:
  int startBytes = s->next * s->size;
  unsigned char *e = (unsigned char *)element;
  for (int i = 0; i < s->size; i++) {
    s->storage[startBytes + i] = e[i];
  }

  s->next++;
  return (s->next - 1);   // Return index number
}
// cont'd
```

A C-like library III (cpp-file) 4

```
void *fetch(CStash *s, int index) {
  // Check index boundaries:
  assert(0 <= index);
  if (index >= s->next) {
    return 0;         // Indicate the end
  }
  // Produce pointer to desired element:
  return &(s->storage[index * s->size]);
}


int count(CStash *s) {
  return s->next;     // Return elements in CStash
}


// cont'd
```


A C-like library IV (cpp-file) 4

```
void inflate(CStash *s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;

  unsigned char *b = new unsigned char[newBytes];
  for (int i = 0; i < oldBytes; i++) {
    b[i] = s->storage[i];   // Copy old to new
  }

  delete [](s->storage);    // Delete old storage
  s->storage = b;           // Point to new memory
  s->quantity = newQuantity;
}

// cont'd
```


A C-like library V (cpp-file) 4

```
void cleanup(CStash *s) {
  if (s->storage != 0) {
    cout << "freeing storage" << endl;
    delete[] s->storage;
  }
}
```

Dynamic storage allocation 4

- Memory is allocated from the heap.
- C++ has a more sophisticated approach for memory allocation than C and uses the keyword **`new`** .
- General form for allocation:
  `new` <type>**`;`**
  You get back a pointer to a <type>.
  If memory allocation fails C++ throws an exceptions (cp. later).

Freeing memory 4

- The keword **`delete`** is a complement of **`new`**. It releases allocated memory.
- There is a special syntax when releasing an array:
  `<type> *array =`
  `new` <type>**`[`**<numElements>**`];`**
  `delete []array;`
- When writing library functions **`cleanup()`** is the place to release all variable memory.

Using `CStash` I 4

```
// cStashTest.cpp
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

int main() {
  CStash intStash, stringStash;
  int i;
  char *cp;
  ifstream in;
  string line;
  const int bufsize = 80;

// cont'd
```

Using `CStash` II 4

```
  // Holds ints:
  initialize(&intStash, sizeof(int));
  for (i = 0; i < 100; i++) {
    add(&intStash, &i);
  }

  for (i = 0; i < count(&intStash); i++) {
    cout << "fetch(&intStash, " << i << "): "
         << *(int *)fetch(&intStash, i) << endl;
  }
```


Using `CStash` III 4

```
  // Holds 80-character strings:
  initialize(&stringStash, sizeof(char) * bufsize);
  in.open("cStashTest.cpp");
  assert(in);

  while (getline(in, line)) {
    add(&stringStash, line.c_str());
  }
  i = 0;
  while ((cp = (char *)fetch(&stringStash, i++))
         != 0) {
    cout << "fetch(&stringStash, " << i << "): "
         << cp << endl;
  }
  cleanup(&intStash);
  cleanup(&stringStash);
}
```


Comments on using `cstash` 4

- In C-like style all variables are declared at the beginning of the scope.
- Initialization and cleanup is the (application) programmer's responsibility.
- Stricter type checking in C++:
  `cp = (char *)fetch(&stringStash, i++);`
  cast is necessary.

Comments cont'd 4

- In C++ it is not possible to call a function that has not previously been declared.
- **`CStash`** is somewhat awkward to use: every function is in need of the address of the structure.
- With libraries from different vendors using identical names (e.g. **`cleanup()`**) name clashes are inevitable.

The basic object 4

- In C++ to avoid name clashes functions can be put into **`struct`**s.
- In C++ no **`typedef`** is necessary with **`struct`**s.

A C++ library (h-file) 4

```
// stash.h

struct Stash {
  // Members:
  int size;            // Size of each space
  int quantity;        // Number of storage spaces
  int next;            // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *storage;

  // Member functions:
  void initialize(int size);
  void cleanup();
  int add(const void *element);
  void *fetch(int index);
  int count();
  void inflate(int increase);
};
```

Comments 4

- The data members are the same as in the C-like library.
- The compiler secretly passes the address of the **`struct`** item into the function.
- The functions are in the namespace of the **`struct`**. To avoid clashes you specify, e.g.

```
Stash::initialize(int size)
```

Scope resolution operator

A C++ library I (cpp-file) 4

```
// stash.cpp
#include "stash.h"
#include <iostream>
#include <cassert>
using namespace std;

const int increment = 100;

void Stash::initialize(int size) {
  this->size = size ;
  quantity = 0;
  next = 0;
  storage = 0;
}
```


A C++ library II (cpp-file) 4

```
int Stash::add(const void *element) {
  if (next >= quantity) {  // Enough space left?
    inflate(increment);
  }
  // Copy element into storage,
  //  starting at next empty space:
  int startBytes = next * size;
  unsigned char *e = (unsigned char *)element;
  for (int i = 0; i < size; i++) {
    storage[startBytes + i] = e[i];
  }
  next++;
  return (next - 1);       // Return index number
}
```

A C++ library III (cpp-file) 4

```
void *Stash::fetch(int index) {
  // Check index boundaries:
  assert(0 <= index);
  if (index >= next) {
    return 0;      // Indicate the end
  }
  // Produce pointer to desired element:
  return &(storage[index * size]);
}


int Stash::count() {
  return next;     // Number of elements in CStash
}
```


A C++ library IV (cpp-file) 4

```
void Stash::inflate(int increase) {
  assert(increase > 0);

  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char *b = new unsigned char[newBytes];

  for (int i = 0; i < oldBytes; i++) {
    b[i] = storage[i];     // Copy old to new
  }
  delete[] storage;        // Old storage
  storage = b;             // Point to new memory
  quantity = newQuantity;
}
```


A C++ library V (cpp-file) 4

```
void Stash::cleanup() {
  if (storage != 0) {
    cout << "freeing storage" << endl;
    delete[] storage;
  }
}
```

Comments 4

- In C++ declarations are mandatory and must be known to the compiler before the definitions.
- Declarations and definitions can reside in the same file.
- The header file is a good place for **`struct`**s. Private **`struct`**s go into the cpp file.
- The address of the **`struct`** item is accessible via the keyword **`this`**.

Comments cont'd 4

- C++ is more particular about type information:

```
int i = 10;
void *vp = &i;  // OK in both C and C++
int *ip = vp;   // Only acceptable in C
```

A cast is required.

Using **`Stash`** I 4

```
// stashTest.cpp
#include "stash.cpp"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>

using namespace std;


int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));



// cont'd
```

Using `Stash` II 4

```
  for (int i = 0; i < 100; i++) {
    intStash.add(&i);
  }

  for (int j = 0; j < intStash.count(); j++) {
    cout << "intStash.fetch(" << j << ") = "
         << *(int *)intStash.fetch(j) << endl;
  }


  // Holds 80-character strings:
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);

// cont'd
```


Using `Stash` III 4

```
  ifstream in("stashTest.cpp");
  assure(in, "stashTest.cpp");
  string line;
  while (getline(in, line)) {
    stringStash.add(line.c_str());
  }

  int k = 0;
  char *cp;
  while ((cp = (char *)stringStash.fetch(k++))
         != 0)
  {
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  }
  intStash.cleanup();
  stringStash.cleanup();
}
```


Comments 4

- Variables are defined "on the fly" (not possible in C).
- The member selection operator "." is used.
- *require.h* is provided by Eckel for more sophisticated error checking than **`assert()`**.
  `assure()` checks if a file could be opened successfully etc.

Abstract data typing 4

- The ability to package data with functions allows to create a new data type.
- Abstract data type are sometimes called user-defined types.
- "Calling a member function for an object" in OO parlance is "sending a message to an object".

`sizeof` 4

Can be used as expected:

```
// ...
struct A a;
Stash stash;

// ...
sizeof(struct A);
sizeof(a);
sizeof(Stash);
sizeof(stash);
```


Header files 4

- Recipe for consistent declarations:
  - Place all your function declarations in a header file.
  - Include the header file everywhere you use and define these functions.
  - If a `struct` is in the header file include it wherever it is used and where `struct` member functions are called.

Multiple declarations 4

- Both C and C++ allow to redeclare a function, as long as the two declarations match.
- Neither allows the redeclaration of a structure.
- To avoid errors use **`#define`**, **`#if`** and **`#endif`** in both languages (often called *include guards*).

Namespaces in h files 4

- **`std`** is the namespace that surrounds the entire Standard C++ library.
- Writing **`using namespace std;`** (in a cpp file) allows to use the library without qualification.
- Don't use the **`using`** directive in a header file (outside of a scope). It would mean to lose the "namespace protection" once this header is included.

Nested structures 4

- A structure can be nested within another structure.
- The syntax is straightforward (cp. next slide).

Example 4

```
// stack.h
#ifndef STACK_H
#define STACK_H

struct Stack {
  struct Link {           // Nested structure
    void *data;
    Link *next;
    void initialize(void *dat, Link *nxt);
  } *head;

  void initialize();
  void push(void *dat);
  void *peek();
  void *pop();
  void cleanup();
};
#endif
```


Example cont'd 4

```
// stack.cpp
#include "stack2.h"
#include "../require.h"
using namespace std;

void Stack::Link::initialize(void *dat, Link *nxt)
{
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void *dat) {
  Link *newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}
```


Example cont'd 4

```
void *Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}

void *Stack::pop() {
  if (head == 0) return 0;
  void *result = head->data;
  Link *oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Stack not empty");
}
```

Global scope resolution 4

- The compiler defaults to the "nearest" name if no qualification is provided.
- Qualification is provided with the "`::`" operator.
- Global scope is addressed with the "`::`" operator with nothing in front of it.

Example 4

```
// scoperes.cpp -- global scope resolution
int a;
void f() {}


struct S {
  int a;
  void f();
};


void S::f() {
  ::f();      // Would be recursive otherwise!
  ::a++;      // Select the global a
  a--;        // The a member at struct scope
}

int main() { S s; f(); }
```


Summary 4

- Functions can be placed inside structures.
- A new type of structure is called an *abstract data type* (ADT). Variables created using an ADT are called *objects* or *instances*.
- Calling a member function for an object is called *sending a message* to that object.

Table of Contents

5 Hiding the Implementation

C++ access control
Friends
Class
Handle classes

C++ access control 5

- C++ introduces three keywords: **`public`**, **`private`** and **`protected`**.
- These *access specifiers* are only used in a structure declaration.
- An access specifier must be followed by a colon.

Example *public* 5

```
// public.cpp
struct A {
  int i;
  char j;
  void func();
};

void A::func() {}


struct B {
public:
  int i;
  char j;
  void func();
};

void B::func() {}
```


Example *public* cont'd 5

```
int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.func();
  b.func();
}
```


Example *private* 5

```
// private.cpp
struct B {
private:
  char j;
  float f;
public:
  int i;
  void func();
};


void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};
```

Example *private* cont'd 5

```
int main() {
  B b;

  b.i = 1;       // OK, public

//  b.j = '1';   // Illegal, private
//  b.f = 1.0;   // Illegal, private

}
```


Friends 5

- Access to a function that isn't a member of the structure can be granted.
- This is accomplished by declaring this function a **`friend`** *inside* that structure.
- There is no way to grant access to **`private`** or **`protected`** members from *outside* a structure.

Example *global friend* 5

```
// globalFriend.cpp
struct X {      // Definition
private:
  int i;
public:
  void initialize();
  friend void g(X *, int);    // Global friend
};

void X::initialize() {
  i = 0;
}

void g(X *x, int i) {   // g() globally defined
  x->i = i;             // Access to private
}                       //  friend data

// ...
```

Exp. *struct member friend I* 5

```
// structMemberFriend.cpp
struct X;          // Incomplete declaration


struct Y {
  void f(X *);
};


struct X {         // Definition
private:
  int i;
public:
  void initialize();
  friend void Y::f(X *);  // Struct member friend
};
```


Exp. *struct member friend II* 5

```
void X::initialize() {
  i = 0;
}


void Y::f(X *x) {     // f() defined for Y struct
  x->i = 47;          // Access to private
}                     //  friend data


// ...
```


Exp. *struct friend I* 5

```
// structFriend.cpp
struct X {
private:
  int i;
public:
  void initialize();
  friend struct Z;    // Entire struct is friend
};                    // Incomplete declaration


void X::initialize() {
  i = 0;
}
```

Exp. *struct friend II* 5

```
struct Z {
private:
  int j;
public:
  void initialize();
  void g(X *x);
};


void Z::initialize() {
  j = 99;
}

void Z::g(X *x) {    // g() defined for Z struct
  x->i += j;         // Access to private
}                    //  friend data

// ..
```


More about **`friend`**s 5

- If a function is a **`friend`**, it means it is *not* a member.
- Permission is given to this **`friend`** to modify private data.
- The **`friend`** concept makes the language less pure with respect to OO.
- **`friend`** can help to increase performance.

`class` 5

- The keyword **`class`** in C++ comes close to being an unnecessary keyword.
- It is identical to **`struct`** with one exception:
  `class` defaults to **`private`**, whereas **`struct`** defaults to **`public`**.

`struct` - `class` 5

```
struct A {
private:
  int i, j;
public:
  int f();
  void g();
};


int A::f() {
  return i + j;
}

void A::g() {
  i = j = 0;
}
```

```
class B {

  int i, j; // private
public:
  int f();
  void g();
};


int B::f() {
  return i + j;
}

void B::g() {
  i = j = 0;
}
```


Access control 5

- The `Stash` example is modified using `class` and access control.
- Notice the clear distinction of the client programmer portion of the interface.
  Client programmers can't accidentally manipulate the (private) part of the class.
- All access control is exclusively done by the compiler.

`Stash` rewritten 5

```
// Stash.h - use access control
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;           // Size of each space
  int quantity;       // Number of storage spaces
  int next;           // Next empty space
  unsigned char *storage;  // Dynamically
                      // allocated array of bytes:
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void *element);
  void *fetch(int index);
  int count();
};
#endif
```

Reducing compilation 5

- If a file is changed in a project this file and its dependents will be recompiled.
- Every time a class is changed this causes a recompilation (fragile base-class problem) with a possible impact on rapid project turnaround.
- This can be avoided using handle classes (a form of the bridge design pattern).

Handle class I 5

```
// Handle.h
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire;      // Class declaration only
  Cheshire *smile;

public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
};


#endif
```


Handle class II 5

```
// Handle.cpp
#include "Handle.h"
#include "../require.h"

// Define Handle's implementation:
struct Handle::Cheshire {
  int i;
};

void Handle::initialize() {
  smile = new Cheshire;
  smile->i = 0;
}

void Handle::cleanup() { delete smile; }

int Handle::read() { return smile->i; }

void Handle::change(int x) { smile->i = x; }
```

Handle class annotations 5

- **`Chesire`** is a nested **`struct`**, so scope resolution is necessary.
- In **`Handle::initialize()`** memory is allocated for the **`struct`**.
- Should **`Cheshire`** be changed then the header file will not be influenced and *Handle.cpp* is the only file which needs recompilation.

Handle class usage 5

```
// UseHandle.cpp - use the Handle class

#include "Handle.h"

int main() {
  Handle u;
  u.initialize();
  u.read();
  u.change(1);
  u.cleanup();
}
```

The client programmer can only access the public interface. As long as only the implementation changes, the above code needs no recompilation.

Summary 5

- Access control is supported by means of access specifiers.
- C++ introduces the *friends* concept: Access can be granted to a function that isn't a member of a structure.
- **`struct`**s and **`class`**es are very similar in meaning.
- How to reduce compilation: the handle trick.

Table of Contents

6 Initialization and Cleanup

Guaranteed initialization
Guaranteed cleanup
Aggregate initialization
Default constructor

Guaranteed initialization 6

- In C++ the concept of initialization and cleanup is essential for easy library use.
- The initialization can and should be guaranteed with the constructor.
- Initialization is too important to leave to the client programmer.
- Using provided constructors is performed by the compiler.

Example 6

```
// Class definition
class X {
  int i;
public:
  X();              // Constructor
};
```

```
// Declaring an object
void f() {
  X a;       // The compiler quietly inserts
             // the call X::X() for object a.
             // a is on the stack.
  // ...
}
```


Initialization cont'd 6

- Each constructor takes the name of its class. It has no return type.
- A constructor can have arguments.
- If you provide just one constructor the compiler won't let you create an object in a different way. It's always the compiler that makes the call.
- Multiple constructors can be provided.

Guaranteed cleanup 6

- Cleanup in C is easily forgotten. In C++ cleanup is guaranteed with the destructor.
- The destructor is preceded by a tilde to distinguish it from the constructor:

`X()`	constructor
`~X()`	destructor

Example 6

```
// Class definition
class Y {
  int i;
public:
  ~Y();              // Destructor
};
```


Cleanup cont'd 6

- The destructor is called automatically when the object goes out of scope.
  The only evidence for this is the closing curly bracket that terminates the scope.

Example 6

```
// constructor1.cpp
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Constructor
  ~Tree();                  // Destructor
  void grow(int years);
  void printsize();
};


Tree::Tree(int initialHeight) {
  height = initialHeight;
}
```

Example cont'd 6

```
Tree::~Tree() {
  cout << "inside Tree destructor" << endl;
  printsize();
}


void Tree::grow(int years) {
  height += years;
}


void Tree::printsize() {
  cout << "Tree height is " << height << endl;
}
```


Example cont'd 6

```
int main() {
  cout << "before opening brace" << endl;
  {
    Tree t(12);
    cout << "after Tree creation" << endl;
    t.printsize();
    t.grow(4);
    cout << "before closing brace" << endl;
  }
  cout << "after closing brace" << endl;
}
```


Output 6

```
before opening brace
after Tree creation
Tree height is 12
before closing brace
inside Tree destructor
Tree height is 16
after closing brace
```

Defining block elimination 6

```
// defineInitialize.cpp
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

// ...

int main() {
  cout << "initialization value? ";
  int retval = 0;       // Defined at point of use
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;   // Defined at point of use
  G g(y);
}
```


`Stash` rewritten 6

- The `Stash` example provides the functions `initialize()` and `cleanup()` which map to constructors and destructors.
- The following example rewritten using constructors and destructors:

Example 6

```
// Stash2.h - With constructors & destructors
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;             // Size of each space
  int quantity;         // Number of storage spaces
  int next;             // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void *element);
  void *fetch(int index);
  int count();
};
#endif
```

Example cont'd II 6

```
// Stash2.cpp - Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;

const int increment = 100;


Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}
```


Example cont'd III 6

```
int Stash::add(void *element) {
  if (next >= quantity) {   // Enough space left?
    inflate(increment);
  }
  // Copy element starting at next empty space:
  int startBytes = next * size;
  unsigned char *e = (unsigned char *)element;
  for (int i = 0; i < size; i++) {
    storage[startBytes + i] = e[i];
  }
  next++;
  return (next - 1);        // Index number
}

void *Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if (index >= next) { return 0; }
  // Produce pointer to desired element:
  return &(storage[index * size]);
}
```


Example cont'd IV 6

```
int Stash::count() {
  return next;     // Number of elements in CStash
}


void Stash::inflate(int increase) {
  require(increase > 0,
    "Stash::inflate zero or negative increase");
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char *b = new unsigned char[newBytes];
  for (int i = 0; i < oldBytes; i++) {
    b[i] = storage[i];     // Copy old to new
  }
  delete [](storage);      // Old storage
  storage = b;             // Point to new memory
  quantity = newQuantity;
}
```

Example cont'd V 6

```
Stash::~Stash() {
  if (storage != 0) {
    cout << "freeing storage" << endl;
    delete[] storage;
  }
}
```


Example cont'd VI 6

```
// Stash2Test.cpp - Constructors & destructors
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));  // Construction

  for (int i = 0; i < 100; i++) {
    intStash.add(&i);
  }
  for (int j = 0; j < intStash.count(); j++) {
    cout << "intStash.fetch(" << j << ") = "
         << *(int *)intStash.fetch(j)
         << endl;
  }
```


Example cont'd VII 6

```
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, "Stash2Test.cpp");
  string line;
  while (getline(in, line)) {
    stringStash.add((char *)line.c_str());
  }

  int k = 0;
  char *cp;
  while ((cp = (char *)stringStash.fetch(k++))
         != 0)
  {
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
  }
}      // Destruction
```

Attention 6

- One thing to be aware of:
  There are only built-in types used in the **`Stash`** example.
- Built-in types have no destructors.

Aggregate initialization 6

- Aggregate: a bunch of things clumped together.
- Aggregates may include mixed types, e.g. **`struct`**s and **`class`**es.
- An array is an aggregate of a single type.

Examples I 6

```
int a[5] = {1, 2, 3, 4, 5};

int b[6] = {0};   // Remaining elements are zeroed

int c[] = {0, 2, 4};         // Automatic counting

struct X {
  int i;
  float f;
  char c
};
X x1 = {1, 2.2, 'c'};
```

Examples II 6

```
// Must use a constructor here
// (even if all members are public):
struct Y {
  int i;
  float f;
  Y(int a);          // Constructor
};

Y y1[] = {Y(1), Y(2), Y(3)};
```


Next example 6

```
// multiarg.cpp -- Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};


Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}
```


Next example cont'd 6

```
void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}


int main() {
  Z z[] = {Z(1, 2), Z(3, 4), Z(5, 6), Z(7, 8)};

  for (int i = 0; i < sizeof z / sizeof *z; i++) {
    z[i].print();
  }
}
```

Default constructor 6

- A default constructor is one that takes no arguments and which is built by the compiler.
- A default constructor for a structure (**`struct`** or **`class`**) is automatically created by the compiler if and only if there are no other constructors provided.

Example 6

```
// autoDefaultConstructor.cpp
// Automatically-generated default constructor

class V {
  int i;                  // private
};                        // No constructor


int main() {
  V v, v2[10];            // Default constructors
                          // used
}
```


Summary 6

- Initialization and cleanup is essential for easy and safe library use.
- This is enforced by the compiler via constructors and destructors.
- The default constructor is automatically built by the compiler if no other constructor is provided.

Table of Contents

7 Function overloading etc.

Overloading
Default arguments
Placeholder arguments
Overloading versus default arguments

Overloading and Arguments 7

- Functions are distinguished by context. This makes name decoration obsolete.
- Instead of **`shirt_wash()`**, **`car_wash()`** ... we use **`wash(Shirt)`**, **`wash(Car)`** ...
- For constructors it's the only way: the constructors' names have to reflect the class name.

`return` & Overloading 7

- Overloading on return values is *not* allowed:
  `void f()` and `int f()` within the same scope would be problematic.
  While context like
  ```
  int i = f();
  ```
  would give a clue which `f()` is meant,
  ```
  f();
  ```
  does not.

Overloading example I 7

```
// Stash3.h - Function overloading
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;           // Size of each space
  int quantity;       // Number of storage spaces
  int next;           // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *storage;
  void inflate(int increase);
public:
  Stash(int size);    // Zero quantity
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void *element);
  void *fetch(int index);
  int count();
};
#endif
```


Overloading example II 7

```
// Stash3.cpp -- Function overloading

// ...

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

// ...
```

Default arguments I 7

- A default argument is a value given in the declaration that the compiler automatically inserts if no value is provided in the function call.
- The two functions

```
Stash(int size);
Stash(int size, int initQuantity);
```

can be replaced with the single function

```
Stash(int size, int initQuantity = 0);
```


Default arguments II 7

- The two object definitions

```
Stash a(100), b(100, 0);
```

will produce exactly the same results.

- For **`a`** the second argument is automatically substituted by the compiler.

Default arguments III 7

- There are two rules to be aware of:
  Rule one:
  Only trailing arguments may be defaulted.
  Rule two:
  Once you start using default args in a function call, all subsequent args in the function's arg list must be defaulted.
- Default args are only placed in declarations.

Code I 7

```
// Stash3a.h -- Default argument
#ifndef STASH3A_H
#define STASH3A_H

class Stash {
  int size;           // Size of each space
  int quantity;       // Number of storage spaces
  int next;           // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *storage;
  void inflate(int increase);
public:
  Stash(int size, int initQuantity = 0);
  ~Stash();
  int add(void *element);
  void *fetch(int index);
  int count();
};
#endif
```


Code II 7

```
// Stash3a.cpp -- Default argument

// ...

Stash::Stash(int sz, int initQuantity /* = 0 */) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

// ...
```

For documentation purposes

Placeholder arguments 7

- Arguments in a function declaration can be declared without identifiers.

```
void f(int x, int = 0, float = 1.1);
```

- Identifiers in function definitions aren't needed either.

```
void f(int x, int, float flt) {
  /* ... */
}
```

`x` and **`flt`** can be referenced in the code body, but not the middle argument.

Overloading vs default args 7

- Prefer overloading...
  if you have to *look* for the default rather than *treating* it as an ordinary value.
  Let the compiler do the selection in providing two functions.
- Maintainability is better especially with large functions.

Overloading vs default args 7

- Prefer default arguments...
  if the default is a value that is likely to be used and generally can be ignored by a client programmer.

  if you encounter cases where you'd use **`this()`** in Java.

Overloading vs default args 7

- Notice that the usage does not change regardless of whether a default constructor is used or overloading:
  No effect on the public interface.

Summary 7

- Overloading allows the use of identical function names distinguished by different parameter lists.
- C++ introduces the concept of default arguments.
- This raises the question whether to choose overloading or default arguments. The answer has no effect on the public interface.

Table of Contents

8 Constants

Value substitution
Aggregates
Pointers and `const`
Function arguments
Temporaries
Classes and `const`
Constructors
`const`ant objects

Concept 8

- The concept of **`const`** was created to allow the programmer distinguish what changes and what doesn't.
- This provides safety and control in C++ projects.

Value substitution 8

- Use a symbol (a constant) instead of a magical number:

```
const int bufsize = 100;
char[bufsize];
```

- **`const`** can be used for all built-in types.
- **`const`** can be used for all **`class`** objects.

`const` in header files 8

- A global **`const`** in a C++ file has file scope.
- You must always assign a value to **`const`**, except if declaring it **`extern`** :

```
extern const int bufsize;
```

- Normally the C++ compiler avoids creating storage for a constant. It then holds it in a symbol table. Not true however if **`extern`** is used.

`const` in header files 8

- Storage is also created for cases where the address of a `const` is taken.

Safety `const`s 8

- If you know that a variable won't change for the lifetime it is good practice to make it a `const`.
- Run-time `const`s have still to be initialized at the point of definition. Once they are initialized they can't be changed.

Example 8

```
// safecons.cpp -- Using const for safety
#include <iostream>
using namespace std;

const int i = 100;        // Typical constant
const int j = i + 10;     // Value from const expr
long address = (long)&j;  // Forces storage
char buf[j + 10];         // Still a const expr


int main() {
  cout << "type a character & CR:";
  const char c = cin.get();    // Run-time const
  const char c2 = c + 'a';     //    "       "
  cout << c2;
  // ...
}
```

Aggregates 8

- It is possible to use **const**ant aggregates.
- It is most likely that the compiler will allocate storage for the aggregate because of lacking sophistication.
- The compiler is *not* required to know the contents of the storage at compile-time.
  Cp illegal statements on next slide.

Example 8

```
// constag.cpp -- Constants and aggregates

const int i[] = {1, 2, 3, 4};

// float f[i[3]];         // Illegal


struct S {
  int i, j;
};

const s S[] = {{1, 2}, {3, 4}};

// double d[s[1].j];      // Illegal


int main() { }
```


extern const 8

- A C++ **const** always defaults to internal linkage, you can't define a **const** and reference it as an **extern** in another file.
- You must explicitely *define* it as **extern** :

```
extern const int x = 1;
```

- Declaring it in another file:

```
extern const int x;
```

Pointer to `const` 8

- **`const int *u;`**
  is read as "**`u`** is a pointer which points to a **`const`**ant **`int`**".
- No initialization is needed because **`u`** points to anything (i.e. it is not a **`const`** itself), but points to a **`const`**.
- Alternate expression with the same meaning (recommended to avoid):
  `int const *u;`

`const` pointer 8

`int d = 1;`
`int * const w = &d;`
is read as "**`w`** is a pointer which is **`const`** that points to an **`int`**".

`*w = d;`
is ok.

- No other address can however be assigned to **`w`**.

Example 8

```
// pointersAndConst.cpp
const int *u;                 // This syntax
                              //  preferred
int const *v;                 //  over this one

int d = 1;
int * const w = &d;

const int * const x = &d;     // This syntax
                              //  preferred
int const * const x2 = &d;    //  over this one


int main() { }
```

Assignment/type checking 8

- The strict type checking of C++ is extended to pointer assignments.
- An address of a non-**`const`** object can be assigned to a **`const`** pointer (You promise not to change something that you normally could).
- You can't assign the address of a **`const`** object to a non-**`const`** pointer (pretending to now be able to change this object).

Example 8

```
// pointerAssignment.cpp
int d = 1;
const int e = 2;
int *u = &d;          // OK -- d not const

// int *v = &e;       // Illegal -- e const

int *w = (int *)&e;   // Legal but bad practice:
                      //  breaking the safety
                      //  mechanism


int main() { }
```


Character array literals 8

- The correct and safe way to define array literals you don't want to be **`const`**ant, is:

```
char ca[] = "howdy";
```

- Compilers let you define the following line and some let you even change the literal (not correct):

```
char *cp = "howdy";
```

Function args & `return` 8

- Passing objects by value specifying `const` has no meaning to the client.
- Returning a value as a `const`ant means the returned value can't be modified.
- Passing/returning addresses as `const`s promises that the destination of the address will not be changed.

Passing by `const` value 8

```
void f1(const int i) {
  i++;         // Illegal
}
```

- As a copy of the original value is made the original will not be changed anyway.
- `const` is a tool for the creator rather than the caller.

Passing by `const` value 8

- If important to the creator a better style would be:

```
void f2(int ic) {
  const int& i = ic;
  i++;         // Illegal
}
```

- The caller is not being bothered with internal details.

Returning by `const` value 8

- Avoid using **`const`** when returning built-in types, it is meaningless because **`return`** values are copied.
- If a function returns a class object as a **`const`** the **`return`** value of that function can't be an lvalue. (lvalue: it can't be assigned to or otherwise modified).

Example 8

```
// constReturnValues.cpp
class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }


// Global functions:
X f5() { return X(); }

const X f6() { return X(); }
```


Example cont'd 8

```
void f7(X& x) {    // Pass by non-const reference
  x.modify();
}


int main() {
  f5() = X(1);     // OK - non-const return value,
                   //  but probably a prog bug
  f5().modify();   // OK, but probably a prog bug
// f7(f5());       // Causes warning or error

  // Causes compile-time errors:
// f6() = X(1);
// f6().modify();
// f7(f6());
}
```

Temporaries 8

- Sometimes during evaluation the compiler must create temporary objects:
  `f7(f5()); // cp example`
- Temporaries built by the compiler are always constant as the programmer can't see them anyway.
- Trying to change a temporary is almost certainly a mistake what the compiler informs you of.

Passing/returning addr 8

- If you pass or return an address (pointer or reference) you should pass it as a **`const`** if at all possible.
- If you don't you're excluding the possibility of using the function with anything that is a constant.
- The choice whether to return a **`const`** pointer or **`const`** reference depends on what you want the client programmer to do with it.

Example 8

```
// constPointer.cpp -- Constant pointer arg/return
void t(int *) { }

void u(const int *cip) {
// *cip = 2;          // Illegal -- modifies value
  int i = *cip;       // OK -- copies value
// int *ip2 = cip;    // Illegal -- non-const
}

const char *v() {
  // Returns address of static character array:
  return "result of function v()";
}

const int * const w() {
  static int i;
  return &i;
}
```

Example cont'd 8

```
int main() {
  int x = 0;
  int *ip = &x;
  const int *cip = &x;
  t(ip);              // OK
// t(cip);            // Not OK -- cip is a const
  u(cip);             // OK -- for const
  u(ip);              // Also OK -- for non-const
// char *cp = v();    // Not OK -- cp not a const

  const char *cp = v();          // OK
// int *ip2 = w();               // Not OK
  const int * const cip2 = w();  // OK -- copied
  const int *cip3 = w();         // OK
// *w() = 1;                     // Not OK
}
```


Standard arg passing 8

- The first choice when passing an argument is to pass it by reference.
- If ever possible (and appropriate) as a **`const`** reference.
- Passing an object as a **`const`**ant reference means that the function is not going to change the destination of the address.

Example 8

```
// constTemporary.cpp - Temporaries are const
class X { };

X f() { return X(); }   // Return by value

void g1(X&) { }         // Pass by non-const ref

void g2(const X&) { }   // Pass by const reference


int main() {
  // Error: const temporary created by f():
// g1(f());
  // OK: g2 takes a const reference:
  g2(f());
}
```

Classes 8

- **`const`** can be used inside classes. However, its meaning is different.
- Entire objects can be made **`const`**ant.
- To guarantee the **`const`**ness of a class object, the **`const`** member function was introduced: only a **`const`** member function may be called for a **`const`** object.

`const` in classes 8

- **`const`** can't be used inside classes the way it is used outside of classes.
- E.g. you can't **`const`** an array size in a class as you would expect it.
- Inside a class you cannot create an ordinary (non-**`static`**) **`const`** with an initial value.
- It must be done with the constructor *at a special place*.

Constructor initializer list 8

- The special initialization point is called the *constructor initializer list*.
- The list is evaluated before any of the code in the constructor; therefore its place between the argument list and the constructor body.

Example 8

```
// constInitialization.cpp - const in classes
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) { }

void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(); b.print(); c.print();
}
```


Built-in type "constructors" 8

- Built-in types in the constructor initializer list look as if they have constructors.
- This is essential when initializing **`const`** data members because they must be initialized before the function body is entered.

Example 8

```
// builtInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) { }
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);     // constructor-like init
  a.print(); b.print();
  cout << pi << endl;
}
```

`const`ants in classes 8

- A compile-time constant inside a class uses the keyword **`static`**.
- Its meaning is that of a **`const`**ant (= final) class instance in Java. A member that cannot change from object to object.
- Unlike other constants in classes a **`static const`** has to be defined at the point of its declaration.

Example 8

```
// StringStack.cpp
// Using static const to create a compile-time
// constant inside a class

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


class StringStack {
  static const int size = 100;
  const string *stack[size];
  int index;
public:
  StringStack();
  void push(const string *s);
  const string *pop();
};
```


Example cont'd 8

```
StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string *));
}


void StringStack::push(const string *s) {
  if (index < size) {
    stack[index++] = s;
  }
}


const string *StringStack::pop() {
  if (index > 0) {
    const string *rv = stack[--index];
    stack[index] = 0;
    return rv;
  }
  return 0;
}
```

Example cont'd 8

```
string iceCream[] = {
  "pralines & cream", "fudge ripple",
  "jamocha almond fudge", "deep chocolate fudge",
  "wild mountain blackberry",
  "raspberry sorbet", "lemon swirl", "rocky road",
};

const int ic_sz =
  sizeof iceCream / sizeof(*iceCream);

int main() {
  StringStack ss;
  for (int i = 0; i < ic_sz; i++) {
    ss.push(&iceCream[i]);
  }
  const string *cp;
  while ((cp = ss.pop()) != 0) {
    cout << *cp << endl;
  }
}
```


`const` objects 8

- A `const` object is defined the same way for user-defined type as for a built-in type:

```
const int i = 1;
const Blob b(2);
```

- For the compiler `const`ness means that no members of the object can be changed during the object's lifetime.

`const` member functions 8

- To ensure that a constant object is not changed by a member function only functions declared `const` can be used with constant objects.
- The `const`ness has to be known by the linker, too, to enforce that no modification of data members occur. So `const`ness is passed on to the linker as part of the (enhanced) signature.

Example 8

```
// constMember.cpp

class X {
  int i;
public:
  X(int ii);
  int f() const;        // Notice place behind
};                      //  function f()

X::X(int ii) : i(ii) { }

int X::f() const { return i; } // Repeated: part
                               // of the signature
int main() {
  X x1(10);
  const X x2(20);
  x1.f();               // OK -- with a const
  x2.f();               // Also OK
}
```


`const` member functions 8

- A `const` member function is the most general form of a member function.
- Any function that doesn't modify member data should be declared as `const`, so it can be used with `const` objects.
- Constructors and destructors can't be `const` member functions.

Example 8

```
// Quoter.cpp - Random quote selection
#include <iostream>
#include <cstdlib>   // Random number generator
#include <ctime>     // To seed random generator
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
  int lastQuote() const;
  const char *quote();
};

Quoter::Quoter() {
  lastquote = -1;
  srand(time(0));  // Seed random number generator
}
```

Example cont'd 8

```
int Quoter::lastQuote() const {
  return lastquote;
}


const char *Quoter::quote() {
  static const char *quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best.",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea that life is serious.",
    "Things that make us happy, make us wise.",
  };
```


Example cont'd 8

```
  const int qsize =
    sizeof quotes / sizeof(quotes *);
  int qnum = rand() % qsize;
  while (lastquote >= 0 && qnum == lastquote) {
    qnum = rand() % qsize;
  }
  return quotes[lastquote = qnum];
}


int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote();    // OK -- No modification
// cq.quote();       // Not OK; non-const function
  for (int i = 0; i < 20; i++) {
    cout << q.quote() << endl;
  }
}
```


Logical **`const`** 8

- You can perform a memberwise change even from within a **`const`** function.
- 1: You can cast away **`const`**ness.
- 2: Preferred technique:
  Use of the keyword **`mutable`** in the class declaration.

Example *casting* 8

```
// castaway.cpp -- "Casting away" constness
class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
// i++;                 // Error: const function
  ((Y *)this)->i++;    // OK: cast away const-ness
  // Better: use C++ explicit cast syntax:
  (const_cast<Y *>(this))->i++;
}

int main() {
  const Y YY;
  YY.f();              // Actually changes it!
}
```


Example *mutable* 8

```
// mutable.cpp -- The "mutable" keyword
class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
// i++;          // Error: const member function
  j++;          // OK: mutable
}


int main() {
  const Z zz;
  zz.f();       // Actually changes it!
}
```


`volatile` 8

- Let the compiler know that a variable can change outside of the compiler's knowledge (variable influenced by hardware, interrupts, other processes).
- The compiler is prevented from optimizing variable access in the program context.

Summary 8

- There are all different kinds of `const` to enhance program robustness.
- Use `const`ants wherever possible.
- Temporaries built by the compiler are always constant.
- `const` in classes are different.
- Argument passing: first choice is to pass arguments by reference.

Table of Contents

9 Inline Functions

Efficiency

Accessors and mutators

require.h

Efficiency 9

- In C efficiency is preserved through the use of macros. Macros are implemented with the preprocessor.
- C++ uses *inline functions* where efficiency is crucial.
- **`inline`** means that a function is expanded in place like a preprocessor macro but under the control of the compiler thus eliminating the overhead of the function call.

`inline` I 9

- Any implemented function within a class body is automatically **`inline`**.
- Any function can be made **`inline`** in preceding it with the keyword **`inline`**.
- For a function to be a macro you must provide the function body:

```
inline int plusOne(int x) {
  return x++;
}
```


`inline` II 9

- When the compiler sees an **`inline`** function signature and body are put into the symbol table.
- At the place of the function call the **`inline`** function is expanded and type checking etc. is performed.
- The **`inline`** code occupies space, but if the function is small enough this can take less space than with an ordinary call.

`inline` III

9

- An **`inline`** function in a header file has a special status:
  There are no multiple definition errors generated when an **`inline`** function is included in files including that header file.

Example

9

```
// inline.cpp -- inlines inside classes
#include <iostream>
#include <string>
using namespace std;

class Point {
  int i, j, k;
public:
  Point() : i(0), j(0), k(0) { }
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) { }
  void print(const string& msg = "") const {
    if (msg.size() != 0) {
      cout << msg << endl;
    }
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};
```


Example cont'd

9

```
int main() {
  // Use of constructors and functions is
  // transparent:
  Point p;
  Point q(1, 2, 3);

  p.print("value of p");
  q.print("value of q");
}
```

Access functions 9

- One of the most important uses of inlines inside a class are access functions (also called setter and getter functions).
- Access with these small functions is remarkably efficient.

Example 9

```
// access.cpp -- Inline access functions

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};


int main() {
  Access a;
  a.set(100);
  int x = a.read();
}
```


Accessors and mutators 9

- Accessor:
  Read the state information of an object.
- Mutator:
  Change the state of an object.

Example 9

```
// Rectangle.cpp -- Accessors & mutators
class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) { }
  int width() const { return wide; }   // Read
  void width(int w) { wide = w; }      // Set
  int height() const { return high; }  // Read
  void height(int h) { high = h; }     // Set
};

int main() {
  Rectangle r(19, 47);
  // Change width & height:
  r.height(2 * r.width());
  r.width(2 * r.height());
}
```


`Stash` with `inline`s 9

- The small functions are defined **`inline`**.
- The two largst functions are left as non-inline. Inlining wouldn't probably cause any performance gains.
- See next slides:

`Stash` I 9

```
// Stash4.h -- Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
  int size;         // Size of each space
  int quantity;     // Number of storage spaces
  int next;         // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char *stor;
  void inflate(int increase);
public:
  Stash(int sz)
    : size(sz), quantity(0), next(0), stor(0) {}
  Stash(int sz, int initQuantity)
    : size(sz), quantity(0), next(0), stor(0) {
    inflate(initQuantity);
  }
```

Stash II

```
  int add(void *element);

  void *fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if (index >= next) {
      return 0;       // To indicate the end
    }
    // Produce pointer to desired element:
    return &(stor[index * size]);
  }

  int count() const { return next; }
};

#endif
```


Stash III

```
// Stash4.cpp
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void *element) {
  if (next >= quantity) {   // Enough space left?
    inflate(increment);
  }
  // Copy element:
  int startBytes = next * size;
  unsigned char *e = (unsigned char *)element;
  for (int i = 0; i < size; i++) {
    stor[startBytes + i] = e[i];
  }
  next++;
  return(next - 1);         // Index number
}
```


Stash IV

```
void Stash::inflate(int increase) {
  assert(increase >= 0);

  if (increase == 0) {
    return;
  }

  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for (int i = 0; i < oldBytes; i++) {
    b[i] = stor[i];         // Copy old to new
  }
  delete [](stor);          // Release old storage
  stor = b;                 // Point to new memory
  quantity = newQuantity;   // Adjust the size
}
```

Limitations to inlines 9

- The compiler can't perform inlining if the function is too complicated. In general looping is considered too complicated.
- No inlining is possible if the address of the function is taken implicitly or explicitly.
- In such cases the compiler treats the function as non-inline. Multiple definition errors are suppressed.

Reducing clutter 9

- Cluttering in a class definition can be reduced in using the **`inline`** keyword in the *cpp*-file (shown on next slide).
- The class is easier to read because all definitions are placed outside of it.

Example 9

```
// noinsitu.cpp -- Removing in situ functions
class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};


inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) { }

inline int Rectangle::getWidth() const {
  return width;
}
```

Example cont'd 9

```
inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}

inline void Rectangle::setHeight(int h) {
  height = h;
}

int main() {
  Rectangle r(19, 47);
  // Transpose width & height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
}
```


Improved error checking 9

- The conditions that the file *require.h* handles end in an **`exit()`**. This is acceptable in cases where not enough command-line args are provided.
- Better error checking (and handling) is achieved with exceptions (cp later).

require.h I 9

```
// require.h - Test for error conditions in progs
// Local "using namespace std" for old compilers
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
  const std::string& msg = "Requirement failed") {
  using namespace std;
  if (!requirement) {
    fputs(msg.c_str(), stderr);
    fputs("\n", stderr);
    exit(1);
  }
}
```

require.h II

9

```
inline void requireArgs(int argc, int args,
  const std::string& msg =
    "Must use %d arguments") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg.c_str(), args);
     fputs("\n", stderr);
     exit(1);
   }
}

inline void requireMinArgs(int argc, int minArgs,
  const std::string& msg =
    "Must use at least %d arguments") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg.c_str(), minArgs);
    fputs("\n", stderr);
    exit(1);
  }
}
```


require.h III

9

```
inline void assure(std::ifstream& in,
  const std::string& filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}

inline void assure(std::ofstream& out,
  const std::string& filename = "") {
  using namespace std;
  if(!out) {
    fprintf(stderr, "Could not open file %s\n",
      filename.c_str());
    exit(1);
  }
}
#endif
```


Summary

9

- In C++ use **`inline`** functions if efficiency is an issue.
- But the compiler decides if a function will be expanded or not.
- In most cases **`inline`** is used with getter and setter member functions.