

Introduction to C & C++

Søren Debois

Department of Theoretical Computer Science
IT University of Copenhagen

September 5th, 2005

Outline

- 1 "Hello, World!"
- 2 Program Structure
- 3 Memory Management

“Hello, World!” in C++

Example (hello.cpp)

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

int main(int, char*[]) {
    cout << "Hello, World!" << endl;
}
```

To compile and run (on unix):

Example

```
> g++ hello.cpp -o hello
> ./hello
Hello, World!

>
```

C++ vs. Java

- Basic control structures are shared among (C, C++ and Java): `if .. else`, `while`, `do .. while`, `for`, `switch`.
- Statements are terminated by `;` and grouped using `{, }`.
- Variables must be declared before use, and must have a type. Some basic types: `void`, `int`, `short`, `long`.
- Exception handling (`try`, `catch` in C++), but no “finally”.
- Same syntax for comments as in Java.
- Programs are compiled and linked.

Program Structure

A C++-program consists of

- implementation files, typically named `.C`, `.cpp`, `.Cpp`, `.cc`,
- and header files, typically named `.h`.

Header files declare the existence of classes. Implementation files implement their methods.

Program Structure (continued)

Here is a header-file for last weeks stack-example.

stack.h

```
#define SIZE 100

class stack {
    int stck[SIZE];
    int tos;
public:
    stack(); //constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};
```

We get some separation of interface and implementation, but not enough for purists: We can see the private members.

Program Structure (continued)

Here is a header-file for last weeks stack-example.

stack.h

```
#define SIZE 100

class stack {
    int stck[SIZE];
    int tos;
public:
    stack(); //constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};
```

We get some separation of interface and implementation, but not enough for purists: We can see the private members.

Program Structure (continued)

And here is the corresponding implementation file.

`stack.cc`

```
#include "stack.h"

stack::stack() { // Constructor
    tos=0;
    std::cout << "stack initialized\n";
}

stack::~~stack() { // Destructor
    std::cout << "stack destroyed\n";
}

...
```

Note the `#include "stack.h"`.

Program Structure (continued)

And here is the corresponding implementation file.

```
stack.cc
```

```
...

void stack::push(int i)
{
    if(tos==SIZE) {
        std::cout << "stack is full\n";
        return;
    }
    stck[tos] = i;
    tos++;
}
int stack::pop()
{
    if(tos==0) {
        std::cout << "stack underflow\n";
        return 0;
    }
}
```

The Preprocessor

Lines beginning with '#' are **preprocessor directives**. `#include "<somefile>"` means simply "read the file "somefile"" as if it was included verbatim at this point.

Besides including files as above, the preprocessor is most commonly used to conditionally compile code.

Example

```
#define INCLUDE_DEBUGGING_CODE 1

int f(int x) {
#ifdef INCLUDE_DEBUGGING_CODE
    cerr << "in f, x=" << x << endl;
#endif
    ...
}
```

Manual Memory Management

C++ has no garbage collector. You have to deallocate memory yourself.

C++ has only very limited automatic allocation. In many cases, you have to allocate memory yourself.

Pointers

A **pointer value** is the memory address of some object.

Example (Pointer values)

```
0, 1, 2, ..., 0xffffffff
```

A **pointer variable** is a variable declared to contain a pointer value.

Example (Pointer variables)

```
int* p; char* q; int** z
```

This is the same distinction as the one between integers and integer variables.

Pointer Operators

There are five basic operations related to pointers.

Operator(s)	Meaning(s)
*	dereference
new, delete	allocation and deallocation
&	address-of
[]	compound dereference
+, -, ...	pointer arithmetics

Dereference

If p is a pointer variable of, say, type `int`, the expression `*p` is the integer pointed to by p . Here is a function and part of the store.

```
int f (int* p) {  
    int k = *p;  
    return k+2;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa08)`? Well, p is `0x2e01fa08`, so `*p` is 289. Add 2 to get 291.

Dereference

If p is a pointer variable of, say, type `int`, the expression $*p$ is the integer pointed to by p . Here is a function and part of the store.

```
int f (int* p) {  
    int k = *p;  
    return k+2;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa08)`? Well, p is `0x2e01fa08`, so $*p$ is 289. Add 2 to get 291.

Dereference

If p is a pointer variable of, say, type `int`, the expression `*p` is the integer pointed to by p . Here is a function and part of the store.

```
int f (int* p) {  
    int k = *p;  
    return k+2;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa08)`? Well, p is `0x2e01fa08`, so `*p` is 289. Add 2 to get 291.

Dereference

If p is a pointer variable of, say, type `int`, the expression $*p$ is the integer pointed to by p . Here is a function and part of the store.

```
int f (int* p) {  
    int k = *p;  
    return k+2;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa08)`? Well, p is `0x2e01fa08`, so $*p$ is 289. Add 2 to get 291.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b (*p)`.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Read.

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b (*p)`.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Write.

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b (*p)`.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b (*p)`.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b` (`*p`).

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is **45678**. Add 1 to get 45679, then write back to `0x2e01fa0b` (`*p`).

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45678

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get **45679**, then write back to `0x2e01fa0b (*p)`.

Dereference (continued)

We can write using dereference.

```
f (int* p) {  
    *p = *p + 1;  
}
```

Address	Value
0x2e01fa04	387
0x2e01fa08	289
0x2e01fa0b	45679

What is the result of the function call `f(0x2e01fa0b)`? Well, `p` is `0x2e01fa0b`, so `*p` is 45678. Add 1 to get 45679, then write back to `0x2e01fa0b` (`*p`).

Allocation & Deallocation

Declaring an object within a method or function allocates that object on the stack.

```
int f(int x) {  
    int y = 5;  
    return x+y;  
    // y is automatically deallocated  
}
```

Stack-variables are also called *automatic* variables, because they are automatically allocated and deallocated.

Allocation & Deallocation (continued)

The same goes for class objects.

```
int f(int x) {  
    Stack s();  
    ...  
    // Compute something using s.  
    ...  
    return s.pop();  
    // s is automatically deallocated  
}
```

When we reach this line, the `Stack` object is allocated on the program's run-time stack (the overlap in names is incidental!), and the constructor of `Stack` is called.

Just before `s` goes out of scope, its destructor is called.

Allocation & Deallocation (continued)

The same goes for class objects.

```
int f(int x) {  
    Stack s();  
    ...  
    // Compute something using s.  
    ...  
    return s.pop();  
    // s is automatically deallocated  
}
```

When we reach this line, the `Stack` object is allocated on the program's run-time stack (the overlap in names is incidental!), and the constructor of `Stack` is called.

Just before `s` goes out of scope, its destructor is called.

Allocation & Deallocation (continued)

The same goes for class objects.

```
int f(int x) {
    Stack s();
    ...
    // Compute something using s.
    ...
    return s.pop();
    // s is automatically deallocated
}
```

When we reach this line, the `Stack` object is allocated on the program's run-time stack (the overlap in names is incidental!), and the constructor of `Stack` is called.

Just before `s` goes out of scope, its destructor is called.

Allocation & Deallocation (continued)

If we want an object to survive beyond the scope of its containing method, we must allocate on the heap.

```
Stack* f(int n) {  
    Stack* p = new Stack();  
    return p;  
    // p is automatically deallocated,  
    // but *p is not.  
}
```

The `new` construct allocates space for a `Stack`-object, then calls the appropriate constructor. Objects constructed using `new` are called dynamically allocated objects.

Allocation & Deallocation (continued)

If we want an object to survive beyond the scope of its containing method, we must allocate on the heap.

```
Stack* f(int n) {  
    Stack* p = new Stack();  
    return p;  
    // p is automatically deallocated,  
    // but *p is not.  
}
```

The `new` construct allocates space for a `Stack`-object, then calls the appropriate constructor. Objects constructed using `new` are called dynamically allocated objects.

Allocation & Deallocation (continued)

To avoid memory leaks, we must deallocate dynamically allocated objects.

```
main () {  
    Stack* s = f(100); // ... f from previous slide.  
    ...  
    // Do some computation using f  
    delete s;  
}
```

The `delete` construct calls the appropriate destructor, then deallocates the space previously used by `s`.

Allocation & Deallocation (continued)

To avoid memory leaks, we must deallocate dynamically allocated objects.

```
main () {  
    Stack* s = f(100); // ... f from previous slide.  
    ...  
    // Do some computation using f  
    delete s;  
}
```

The `delete` construct calls the appropriate destructor, then deallocates the space previously used by `s`.

Address-of

If x is a variable, $\&x$ is the address of that variable. We use this to call functions that modify their arguments.

```
f (int* p) {  
    *p = *p + 1;  
}  
  
main() {  
    int x = 6;  
    f (&x);  
    printf ("%d\n", p);  
}
```

(There are better ways to achieve this in C++, but in C it is the only way. And a number OS-interfaces are written in C.)

Address-of

If x is a variable, $\&x$ is the address of that variable. We use this to call functions that modify their arguments.

```
f (int* p) {
    *p = *p + 1;
}

main() {
    int x = 6;
    f (&x);
    printf ("%d\n", p);
}
```

(There are better ways to achieve this in C++, but in C it is the only way. And a number OS-interfaces are written in C.)

Arrays

We can dynamically allocate arrays of things using brackets. Let's allocate and de-allocate one-hundred bytes contiguous.

```
char* p = new char[100];  
...  
delete [] p;
```

(The `[]` are important! Otherwise, you delete only the first element!)

Q: But `*p` just gives us the first element. How do we get to the rest?

A: `p[50]` accesses element 51. (We're indexing from 0.)

Arrays

We can dynamically allocate arrays of things using brackets. Let's allocate and de-allocate one-hundred bytes contiguous.

```
char* p = new char[100];  
...  
delete [] p;
```

(The `[]` are important! Otherwise, you delete only the first element!)

Q: But `*p` just gives us the first element. How do we get to the rest?

A: `p[50]` accesses element 51. (We're indexing from 0.)

Arrays

We can dynamically allocate arrays of things using brackets. Let's allocate and de-allocate one-hundred bytes contiguous.

```
char* p = new char[100];  
...  
delete [] p;
```

(The `[]` are important! Otherwise, you delete only the first element!)

Q: But `*p` just gives us the first element. How do we get to the rest?

A: `p[50]` accesses element 51. (We're indexing from 0.)

Pointer arithmetic

More generally, pointer values are just integers, so if we know that a `char` takes up only one byte, we just add 50 to `p` to get to the 51st element:

```
* (p+50)
```

(This is actually the meaning of `p[50]`.) In particular, we can use the `++`-operators on pointers. What does `*++p` mean? `*p++`? `*p += 8`? `++*p`?

Pointer arithmetic

More generally, pointer values are just integers, so if we know that a `char` takes up only one byte, we just add 50 to `p` to get to the 51st element:

```
*(p+50)
```

(This is actually the meaning of `p[50]`.) In particular, we can use the `++`-operators on pointers. What does `*++p` mean? `*p++`? `*p += 8`? `++*p`?

C-Style Strings

In C, a zero-terminated string (or simply string) is simply an array of `chars`. By convention, the last element in the array must have value 0 (the null character).

So, the representation of the string "Hello, world" is actually a sequence of `chars` `'h', 'e', ..., 'l', 'd', 0`.

C-Style Strings

In C, a zero-terminated string (or simply string) is simply an array of `chars`. By convention, the last element in the array must have value `0` (the null character).

So, the representation of the string "Hello, world" is actually a sequence of `chars` `'h', 'e', ..., 'l', 'd', 0`.

C-Style Strings (continued)

Here is the `strlen` function; it computes the length of a string.

Example

```
int strlen(char* p) {
    int i=0;
    for (; *p; ++p, ++i)
        /* Do nothing */ ;
}
```

Freely available compilers:

- **GCC** <http://gcc.gnu.org>
- **Microsoft Visual C++**
<http://msdn.microsoft.com/visualc/vctoolkit2003/>